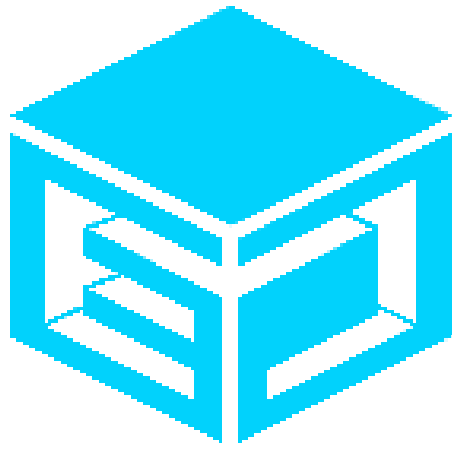




EcoCtrl

能源与 IoT 控制平台

3D 建筑可视化 · 实时监控 · 统一管理后台 — 一个 monorepo
全部集成。

[什么是 EcoCtrl ?](#)

[用户手册 📖](#)



3D 建筑可视化

基于 Babylon.js 的交互式 3D 场景，支持热点标签、辉光效果和可配置的自动旋转相机。



实时监控

通过 SSE 实时推送能耗、故障和 IoT 设备数据，配合 Recharts 可视化分析。



统一管理后台



现代化 Monorepo

React 19 Tab 式控制面板，覆盖用户、模型、能源、故障、维保、报表与平台配置。

pnpm 工作区，包含源码分发的 UI 组件库、Rolldown 打包的服务端、统一工具链。



生产级认证

自动轮转 JWT + Refresh Token、OAuth 登录、SMTP 邮件验证码、基于角色的权限模型。



一键部署

一条 docker compose up 启动整套服务，支持离线部署包与无外网环境。

基于 MIT 协议发布

Copyright © 2026 EcoCtrl

平台概述

1. 什么是 EcoCtrl

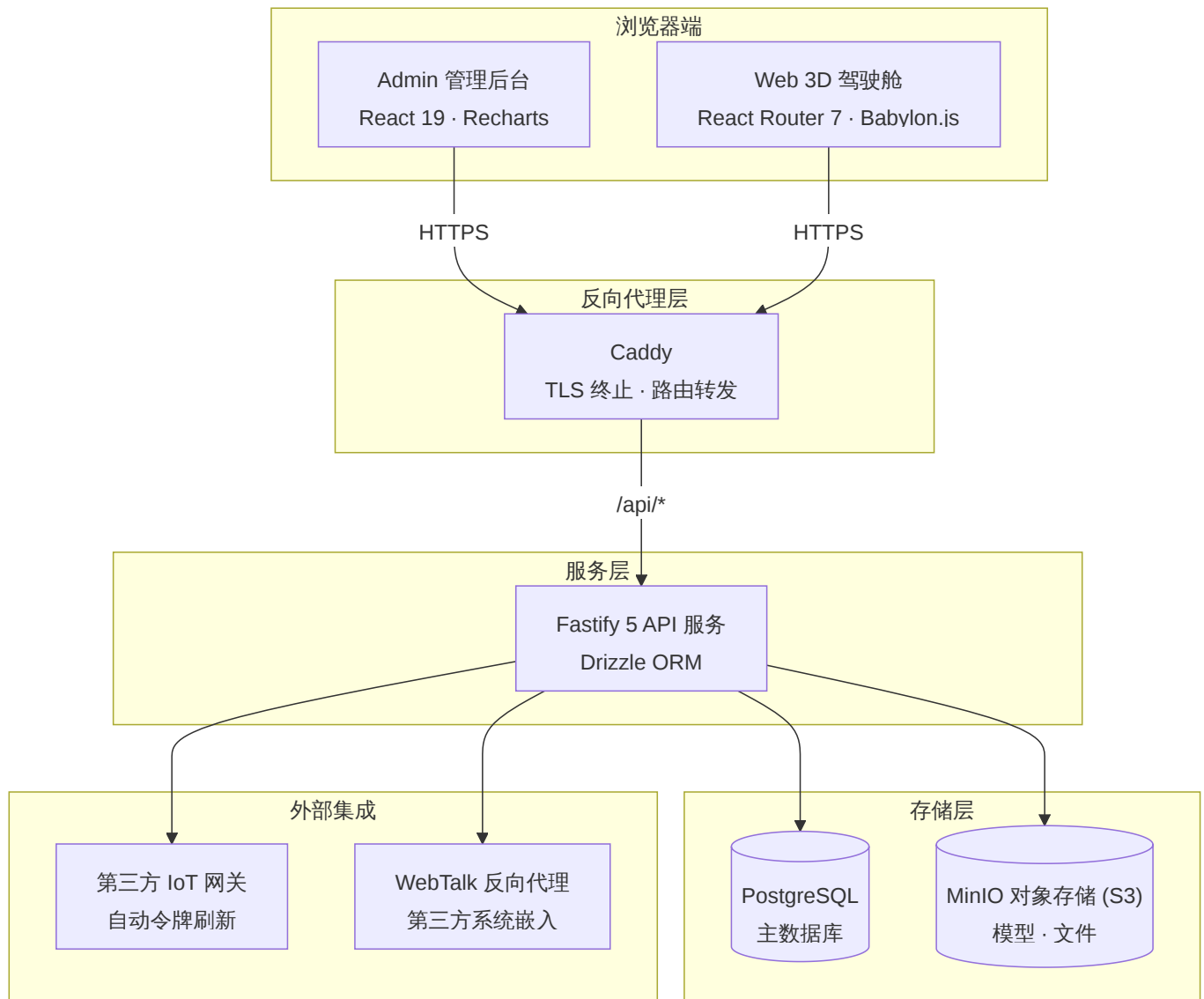
EcoCtrl 是一个能源与 IoT 控制平台，将 3D 建筑可视化门户、实时监控管理后台、REST API 后端以及工作流自动化引擎整合在统一的 pnpm monorepo 中。

平台包含两大前端入口，分别面向不同使用场景：

入口	技术栈	定位
Admin 管理后台	React 19 + Recharts + TailwindCSS v4	功能全面的 Tab 式管理面板，面向运营团队和管理员，覆盖用户、模型、能源、故障、维保、报表与系统配置
Web 3D 可视化驾驶舱	React Router 7 + Babylon.js + TailwindCSS v4	以 3D 建筑场景为核心的公共门户，面向访客和操作员，支持楼层漫游、热点标签与实时数据叠加

2. 高层架构

以下架构图展示了平台的整体拓扑与数据流向：



关键设计原则：

- 所有前端应用统一使用 `/api` 前缀访问后端。实际后端主机地址由开发模式下的 Vite 代理或生产环境中的 Caddy 在运行时改写，修改 API 地址无需重新构建前端。
- 存储于 MinIO 的对象存储文件（3D 模型、上传文件、宠物精灵图等）均通过 API 服务端 `streamFile()` 反向代理流式传输至前端，前端不直接访问 MinIO。这一设计避免了独立 MinIO 实例的跨域（CORS）问题，同时简化了鉴权与缓存控制（支持 304 条件请求与 `Cache-Control` 头部）。

3. 核心能力全景

能力领域	说明	面向用户
3D 建筑可视化	Babylon.js 驱动的交互式 3D 场景，支持多楼层漫游、热点标签、辉光效果与可配置自动旋转相机	所有用户
实时监控	通过 Server-Sent Events (SSE) 推送能耗数据、设备故障与 IoT 状态变化，结合 Recharts 实现可视化分析	运营操作员、维保人员
统一管理后台	Tab 式导航面板，覆盖用户管理、数据模型、能源管理、故障管理、维保管理、报表系统与平台配置	系统管理员、运营操作员
工作流自动化	可视化 DAG 工作流编辑器，支持定时触发、Webhook 触发与事件触发，内置插件系统支持自定义节点	运营操作员、开发者
IoT 设备集成	透明网关代理模式，自动管理第三方 IoT 接口的令牌刷新与重连，提供类型化的设备数据接入	开发者、系统管理员
AI 智能助手	Screen Pet 对话式助手，支持自然语言交互与语音输入，可辅助查询能耗、故障与设备状态	所有用户
第三方系统集成	WebTalk 反向代理模块，支持将第三方 Web 系统以 iframe 方式嵌入 EcoCtrl 界面，实现统一工作台	系统管理员、开发者
报表与数据导出	定时报表计划、多维度能耗统计报表、碳排放因子管理、支持 PDF / Excel 导出	能源管理员、运营操作员

4. 适用角色

不同角色的典型工作场景如下：

4.1. 系统管理员

管理平台的用户体系、访问权限与全局配置。负责用户创建与角色分配、系统参数维护、OAuth / SMTP 等服务配置，以及安全审计日志的审查。

4.2. 运营操作员

日常使用最频繁的角色。通过 Admin 后台监控仪表盘查看能耗趋势与实时告警；通过工作流编辑器配置自动化任务（如定时报表、设备联动）；处理故障告警并进行初步处置。

4.3. 能源管理员

关注建筑各分区的能耗情况与碳排放指标。使用能源管理模块查看分区能耗趋势、管理碳排放因子、生成对标分析报表，为节能优化提供数据支撑。

4.4. 维保人员

负责设备的实时故障监控与维保管理工作。通过实时告警面板及时发现异常，调取维保说明书进行现场处置，记录维修历史。

4.5. 开发者

通过 REST API 与平台集成，扩展现有功能或在 WebTalk 框架下开发子系统插件。API 覆盖认证、数据模型、能源、IoT 事件、报表导出等全部业务领域。

5. 接下来去哪里

根据你的角色和需求，选择以下入口开始使用：

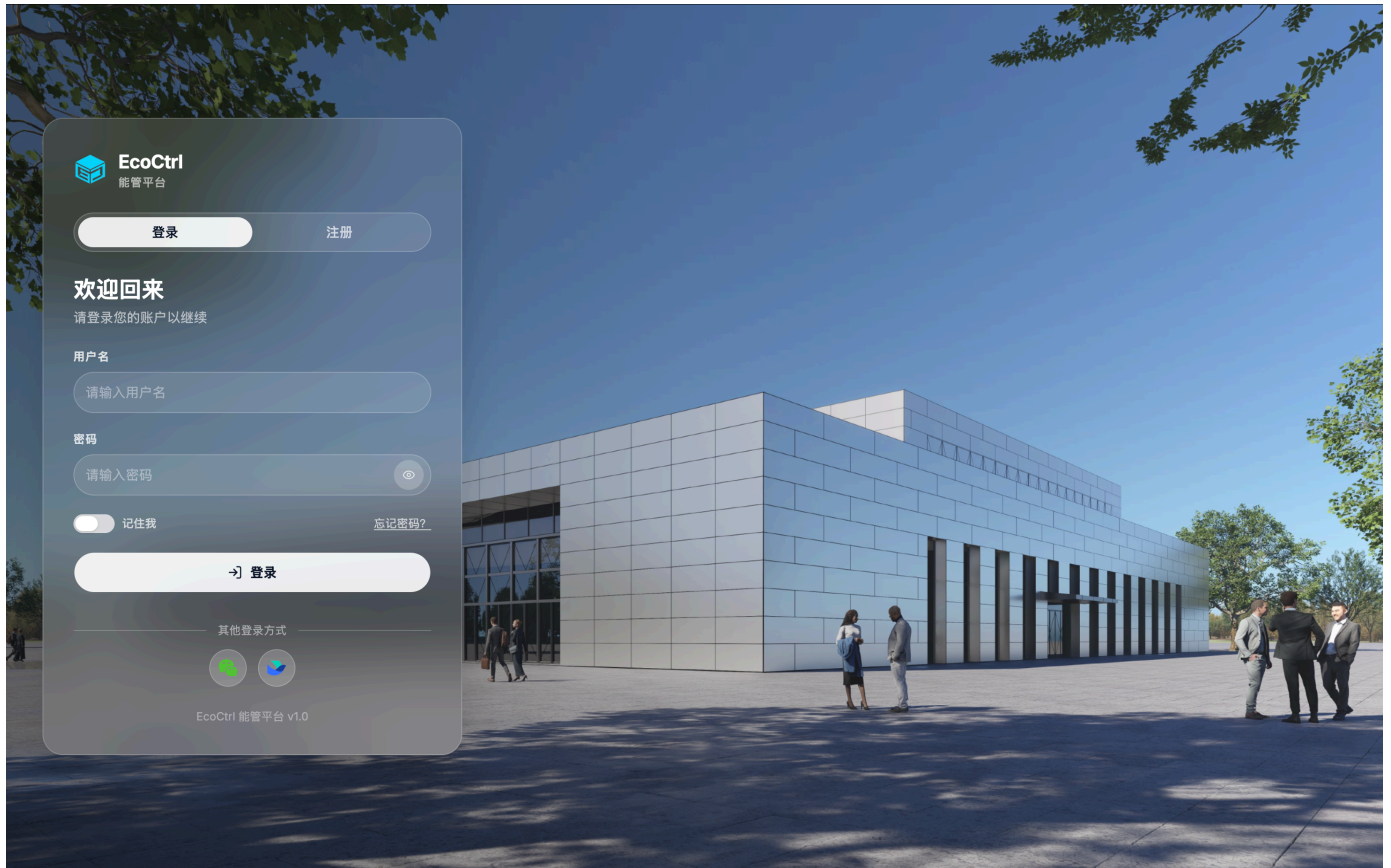
如果你是...	建议从这里开始
第一次访问平台的用户	→ 登录与账户访问 了解如何登录、注册与绑定 OAuth
运营操作员 / 管理员	→ Admin 管理总览 查看仪表盘功能概览
想体验 3D 可视化	→ Web 3D 建筑驾驶舱 了解 3D 场景操作
需要配置工作流	→ 工作流管理概览 入门工作流自动化
开发者 / 系统集成	→ 整体架构概览 了解技术架构与 API 路由
准备部署上线	→ 系统要求与前提条件 查看部署指南

6. 技术栈一览

层级	技术栈	用途
apps/web	React Router 7 + Babylon.js + TailwindCSS v4	公共 3D 门户 — 楼宇视图、楼层、系统、分析
apps/admin	React 19 + Recharts + TailwindCSS v4	内部管理后台 — 账号、设备、模型、报表、设置
apps/docs	VitePress 2	当前文档站点
packages/server	Fastify 5 + Drizzle ORM + PostgreSQL + Rolldown	REST API 与 IoT 网关代理
packages/ui	React + Base UI + TailwindCSS v4	共享组件库（shadcn/ui 风格）
packages/shared	Zod + TypeScript	共享 schema、类型与 Vite 工具

还在寻找更多信息？ 查阅完整的 用户指南 与 技术参考 文档，或通过搜索功能查找关键词。

登录与账户访问



EcoCtrl 平台提供多种登录方式，支持账号密码、邮箱验证码、OAuth 第三方登录。本文将逐一介绍每种方式的操作步骤及注意事项。

1. 登录

访问 Admin 后台地址（默认 <http://localhost:4173>），即可看到登录页面。

操作步骤：

1. 在浏览器中打开 <http://localhost:4173>。
2. 输入用户名和密码。
3. 根据需要开启「记住我」开关（详见下文）。
4. 点击「登录」按钮。
5. 登录成功后自动跳转到管理总览页面。

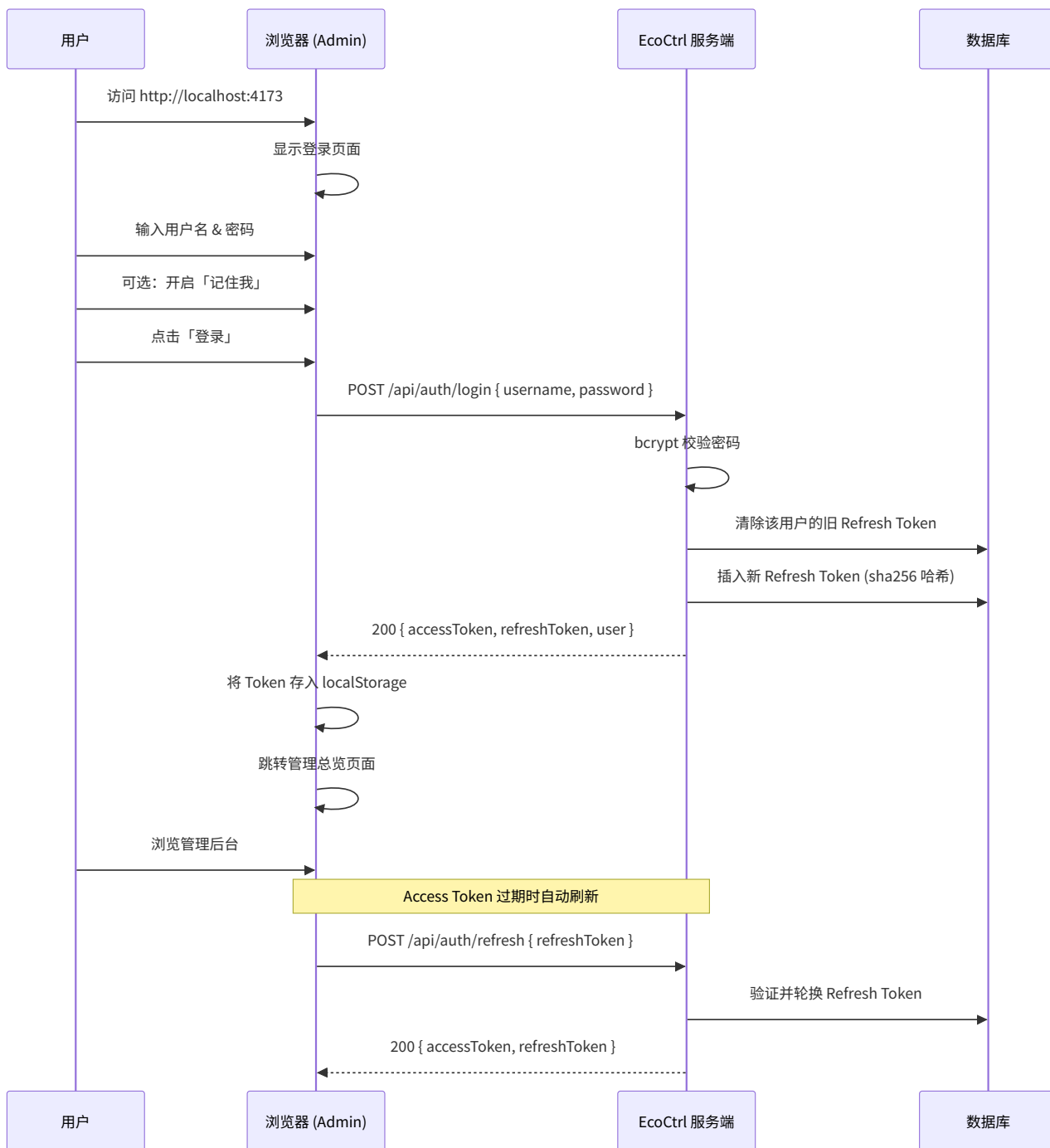
首次部署：

- 默认管理员账号： `admin`
- 初始密码由以下两种方式之一决定（优先级从上到下）：
 - `INIT_ADMIN_PASSWORD` 环境变量指定
 - 运行 `pnpm db:seed` 时输出的初始密码
- 首次登录后建议立即修改密码。

记住我：

「记住我」开关控制会话的持久化时长。开启后，Refresh Token 的有效期将延长，减少因 Token 过期而需要频繁重新登录的情况。关闭时，会话在浏览器关闭或 Token 过期后自动失效。具体有效期配置可参考服务端运维文档。

登录流程：



2. 注册新账户

登录页面底部提供「注册新账户」切换链接，点击后进入注册模式。

操作步骤：

1. 在登录页底部点击「注册新账户」。
2. 填写以下信息：

- 用户名：登录时使用的账号名称。
- 邮箱：用于接收验证码和后续找回密码。
- 验证码：点击「发送验证码」，6 位数字码将发送至填写的邮箱。
- 密码：设置登录密码。
- 确认密码：再次输入密码以确认。

3. 点击「注册」按钮。

注意事项：

- 验证码通过 SMTP 邮件发送，有效期为 5 分钟。请在收到后尽快填写。
- 未收到验证码时，可检查邮箱垃圾箱或稍后重试发送。
- 每个验证码只能使用一次，使用后立即失效。
- 注册成功后自动登录，无需再次输入密码。
- 新账户的默认角色为 **viewer**（只读权限）。如需提权，请联系管理员在 Admin 后台用户管理页面操作。

3. 忘记密码

如果忘记登录密码，可通过邮箱验证码重置密码。

操作步骤：

1. 在登录页面点击「忘记密码」链接。
2. 输入注册时使用的邮箱地址。
3. 点击「发送验证码」，6 位数字码将发送至该邮箱。
4. 在页面中输入收到的验证码。
5. 输入新密码并确认。
6. 点击「重置密码」完成操作。

关于会话：

- 密码重置成功后，当前设备上的登录状态在 Access Token 有效期内仍然可用。
- Access Token 过期后，由于密码已变更，系统将无法签发新的 Token，届时需要使用新密码重新登录。
- 如需在所有设备上立即下线，管理员可在服务端清除该用户的 Refresh Token 记录。

4. OAuth 登录

当服务端已配置 OAuth Provider 时（当前支持 微信 和 飞书），登录页面会增加对应的第三方登录按钮。

首次使用 OAuth：

首次使用 OAuth 登录时，需要将第三方身份与 EcoCtrl 账户绑定。绑定流程分为两种情况：

情况	操作
已有 EcoCtrl 账户	登录后选择「绑定 OAuth」，将第三方账号与现有账户关联
尚无 EcoCtrl 账户	在 OAuth 授权后引导注册新账户，同时完成绑定

绑定后的效果：

- 一个 EcoCtrl 账户可以同时绑定多个 OAuth Provider。
- 绑定后，用户既可以使用 账号密码 登录，也可以使用 OAuth 登录。
- 无论通过哪种方式登录，最终访问的都是同一个用户账户和数据。

操作步骤：

1. 在登录页面点击对应的 OAuth 按钮（如「微信登录」或「飞书登录」）。
2. 浏览器弹出 Provider 授权窗口。
3. 在 Provider 页面完成身份确认。
4. 授权窗口自动关闭，页面完成登录。

配置要求：

OAuth Provider 需要在服务端配置凭据后方可使用。具体配置项（`WECHAT_APPID`、`WECHAT_SECRET`、`FEISHU_APPID`、`FEISHU_SECRET`）在 `packages/server/.env.local` 中设置。未配置的 Provider 不会在登录页显示。

5. 三种界面模式

EcoCtrl 提供三种不同的界面模式，分别面向管理操作、数据可视化和公开访问。

5.1. Admin 管理后台

- 访问地址：<http://localhost:4173>
- 登录后进入 Tab 式控制面板。
- 包含用户管理、设备管理、数据查询、系统设置等核心管理功能。
- 需要登录才能访问。

5.2. Web 3D 驾驶舱

- 独立入口，提供能源数据的 3D 可视化展示。
- 支持 匿名访问：公开仪表盘无需登录即可查看。
- 如需要查看更详细的数据或操作，也可使用账户登录。

5.3. 登录前可访问的页面

以下页面在未登录状态下即可访问：

页面	说明
Web 公开仪表盘	无需登录查看公开能源数据
登录页	账号密码 / OAuth 登录入口
注册页	创建新账户
忘记密码	密码重置流程

6. 常见问题

问：登录后无法跳转？

检查浏览器是否禁用了 `localStorage`。系统需要 `localStorage` 存放 Token。

问：注册时收不到验证码？

- 确认邮箱地址填写正确。
- 检查邮箱垃圾箱。
- 确认服务端已正确配置 SMTP 发送服务。
- 单个邮箱发送频率可能受限，等待 1 分钟后重试。

问：OAuth 按钮没有显示？

说明服务端未配置对应 OAuth Provider 的凭据，请联系系统管理员配置。

问：密码重置后旧设备还能登录吗？

旧设备上的当前会话在 Access Token 有效期内仍然可用。Access Token 过期后需要重新登录。如需强制所有设备下线，请联系管理员。

导航与界面布局



EcoCtrl 提供两个前端入口——Admin 管理后台和 Web 3D 驾驶舱——它们的导航逻辑和页面布局各有不同。本文将系统性地介绍这两个入口的界面结构，帮助你快速找到所需功能。

1. Admin 导航架构

Admin 管理后台采用 左侧可折叠侧边栏 + 内容区 的两栏布局。侧边栏是主要的导航工具，所有顶级功能模块都在这里以 Tab 形式列出。

1.1. 侧边栏布局

侧边栏有两种状态：

状态	宽度	说明
展开	240px	显示图标 + 文字标签，适合大屏幕操作
折叠	64px	仅显示图标，鼠标悬停时弹出文字提示（Tooltip）

侧边栏底部有一个折叠/展开切换按钮：

- 展开状态下，按钮显示为向左箭头（ [ChevronLeft](#) ），点击后折叠。
- 折叠状态下，按钮显示为向右箭头（ [ChevronRight](#) ），悬停在底部区域时浮现，点击后展开。
也可直接点击顶部品牌区域展开。

侧边栏折叠状态会自动保存，刷新页面后保持之前的折叠/展开设置。

1.2. 顶级导航 Tab

侧边栏共列出 11 个顶级 Tab，按功能领域组织如下（列出顺序即侧边栏显示顺序）：

Tab	图标	说明
管理总览	LayoutDashboard	平台核心仪表盘，展示关键指标和实时状态
账户控制	Users	用户管理、角色权限、OAuth 绑定配置
数据模型	Server	管理数据模型（Schema）、业务对象和点位信息
大屏模型	Box	配置 3D 大屏的热点区域和标签（全屏沉浸模式）
逻辑控制	Workflow	可视化工作流编辑器与执行记录
报表管理	FileBox	报表模板、生成与导出管理
维保管理	Wrench	设备维保计划、工单与历史记录
故障管理	AlertTriangle	实时故障列表与故障统计分析
能耗管理	Zap	分区能耗概览、详细数据、统计报表与碳排放因子配置
高级管理	Shield	高级平台管理功能
系统配置	Settings	系统全局设置入口，内部包含个人信息与偏好设置子页面

1.3. 当前 Tab 高亮

侧边栏中当前激活的 Tab 会有明显的视觉区分：

- 左侧显示一条彩色竖条指示线。
- 背景和文字颜色变化（半透明主色背景加主色文字）。
- 图标略微放大，线条加粗。
- 切换 Tab 时，指示线会以平滑动画过渡到新的位置。

1.4. 状态持久化

当前激活的 Tab 通过全局状态管理（Zustand）持久化到浏览器的 `localStorage`，刷新页面或关闭浏览器后重新打开，都会停留在上次访问的 Tab 上，无需重新定位。

2. 页面内子 Tab 切换

部分功能模块内部包含多个子页面。Admin 后台通过 子 Tab 栏 实现页面内的切换，子 Tab 栏位于页面顶部、面包屑/标题的右侧。

2.1. 子 Tab 示例

父模块	子 Tab
能耗管理	分区总览 / 详细数据 / 统计报表 / 碳排放因子
逻辑控制	工作流 / 执行记录
故障管理	实时故障列表 / 故障统计分析
数据模型	数据模型 / 业务对象 / 点位信息
大屏模型	热点区域配置 / 标签配置

2.2. 子 Tab 的持久化

每个模块的子 Tab 状态独立存储、互不干扰。切换模块后，子 Tab 的选择会被记住，再次回到该模块时，会自动定位到最后一次查看的子页面。例如：

1. 在「能耗管理」中切换到「详细数据」子页。

2. 切换到「故障管理」查看故障列表。

3. 点击「能耗管理」返回，页面直接显示「详细数据」而不是默认的「分区总览」。

这种持久化同样通过 Zustand 存储到 `localStorage`，刷新或关闭浏览器后仍然保持。

3. 界面布局特性

3.1. 面包屑导航

页面顶部的面包屑组件显示当前所在的位置层级。以「个人信息」页面为例：

首页 / 系统配置 / 个人信息

plaintext

面包屑的特性：

- 每个分段（除「首页」外）如果存在同级页面，会显示为下拉菜单，点击可快速跳转到同级页面。
- 左侧扁平布局（手机等小屏幕）时，面包屑隐藏，改用页面标题展示。
- 面包屑的显示/隐藏可以在「偏好设置」中切换。

3.2. 页面头部

每个功能页面的头部区域包含：

- 左侧：面包屑导航（大屏幕）或当前页面标题（小屏幕）。
- 中侧：子 Tab 切换栏（如该模块有子页面）。
- 右侧：全局搜索、主题切换（浅色/深色/跟随系统）、通知铃铛、用户头像与下拉菜单。

用户头像下拉菜单提供三个入口：

- 个人信息：修改头像、用户名、邮箱等资料。
- 偏好设置：调整主题、布局密度、字体大小、通知方式等。
- 退出登录：清除当前会话。

3.3. 全屏沉浸式页面

以下三个页面采用 全屏沉浸式布局，进入后会隐藏侧边栏和页面头部，独占整个浏览器视口，以获得最大的操作空间：

页面	入口位置	说明
大屏模型	侧边栏「大屏模型」Tab	全屏配置 3D 热点区域和标签，预览大屏效果
workflow 画布	逻辑控制 > workflow 详情	可视化 DAG workflow 编辑器的全屏操作界面
日志查看器	逻辑控制 > 执行记录 > 详情	全屏查看 workflow 执行日志，支持日志流式滚动

在沉浸式页面的右上角通常有「退出全屏」或「返回」按钮，点击后恢复侧边栏和头部。

3.4. 系统配置嵌套页面

「系统配置」不包含子 Tab，而是在其下以 独立页面 的方式管理两个子页面：

- 个人信息（ </profile> ）：修改用户头像、用户名、邮箱等账户资料。
- 偏好设置（ </preferences> ）：调整界面主题（浅色/深色/跟随系统）、布局密度（舒适/紧凑）、字体大小、桌面通知、告警音效、邮箱通知、侧边栏默认折叠状态、面包屑显示开关。

这两个页面不出现在侧边栏中，而是通过用户头像下拉菜单或系统配置内的链接跳转到达。

3.5. 内容区布局分类

Admin 后台的内容区根据页面类型有两种布局方式：

布局模式	适用的页面	说明
居中定宽	管理总览、账户控制、报表管理、维保管理	内容在 1440px 最大宽度内居中显示，两侧留白，适合阅读型页面
全宽流动	系统配置、偏好设置、个人信息、逻辑控制、数据模型、故障管理、能耗管理、高级管理	内容区域填满屏幕宽度，适合数据密集和操作密集型页面

4. Web 3D 驾驶舱界面

Web 门户（ / ）以 3D 建筑场景为核心，提供沉浸式的可视化体验，界面构成与 Admin 后台有较大差异。

4.1. 整体页面结构

Web 驾驶舱是一个 单页面应用，页面由三个视觉层级叠加构成：

1. 全屏背景层： Babylon.js 3D 建筑场景，占据整个视口
2. 覆盖层： Bento Grid 仪表盘组件（统计卡片、图表、列表等），悬浮在 3D 场景之上
3. 浮动控件： 导航栏（顶部 SSE 状态 + 时间）、相机控制按钮（右下侧浮动）、区域标签（3D 场景中悬浮）

4.2. 顶部导航栏

导航栏固定在页面顶部（高度 56px），包含三个区域：

左侧： Logo 按钮。包含 EcoCtrl 品牌图标和文字，点击后跳转到 </settings> 设置页。

中央： 状态胶囊。显示以下信息：

- SSE 连接状态指示器： 一个彩色圆点，绿色（已连接） / 琥珀色（重连中） / 红色（已断开），旁边显示实时的连接状态。
- 场地标识文字（如 "ECCTRL LOCATION"）及 "MONITORING" 标签。

右侧： 当前时间。24 小时制时间显示，下方配日期，每秒自动更新。

4.3. 3D 场景与浮动相机控制

3D 建筑场景以全屏背景方式渲染（ Babylon.js 引擎），覆盖整个浏览器视口。

场景上方浮动一组相机控制按钮，位于页面右下侧，纵向排列：

- 全屏切换（最大化/最小化图标）： 进入或退出沉浸式全屏模式。
- 放大（ + ）： 推进相机，拉近视角。
- 缩小（ - ）： 拉远相机，扩大视野。
- 复位： 将相机恢复到默认位置和角度，同时清除选中的热点标签、退出全屏模式。

点击 3D 场景中的热点标签时，相机自动聚焦到该区域，左侧弹出 照明控制面板（宽度 320px），用于调整该区域的照明组状态。此时仪表盘组件自动淡出，相机控制按钮隐藏。

4.4. Bento Grid 仪表盘覆盖层

3D 场景之上覆盖有一层 Bento Grid 布局的仪表盘，以 16 列 x 8 行的 CSS 网格排列。每个网格单元放置一个数据可视化组件（能耗趋势图、设备状态、告警列表等）。

查看模式：组件按预设位置排列，鼠标悬停可查看详情。

编辑模式：点击页面底部的编辑工具栏可进入布局编辑：

- 每个组件左上角出现拖拽手柄，可以自由拖动到其他位置。
- 组件右下角可调整尺寸（宽度/高度）。
- 点击组件上的眼睛图标可隐藏该组件，隐藏后的组件在编辑模式下显示为占位符。
- 将组件拖到其他组件上方可交换二者位置。
- 编辑完成后点击「完成」保存布局，或点击「取消」回退到之前的状态。

布局编辑支持自动保存到后端，每次修改会以去抖方式同步到服务端。

4.5. 设置页

点击顶部导航栏的 Logo 或使用底部导航 Pill 中的「设置」按钮，可进入全屏设置页面（</settings>），覆盖在 3D 场景之上。

设置页左侧是导航选项卡，右侧是滚动内容区，使用 Scroll Spy 技术自动高亮当前浏览的章节：

选项卡	配置项
General	语言、减少动效、加载动画开关、Screen Pet 主题、语音配置
Appearance	自动旋转、热点标签显示、辉光效果、环境预设、相机设置
Layout	Bento Grid 编辑入口、编辑超时自动退出延迟
Data	数据刷新闻隔、导航隐藏延迟、同步去抖间隔
Account	登录/注销、用户信息展示

设置页右上角有「保存」和「重置」按钮，修改内容以去抖方式自动同步到服务端。

4.6. 浮动 AI 助手

Screen Pet 是一个可拖拽的浮动角色，在所有页面或场景中都可见。点击它可以展开对话气泡，通过自然语言与平台 AI 交互，查询能耗数据、设备状态或执行简单的控制操作。

5. 常见问题

问：侧边栏折叠后找不到某个功能？

鼠标悬停到对应的图标上，会弹出文字提示显示功能名称。如果仍不确定，可以点击底部的展开按钮将侧边栏恢复为完整显示。

问：子 Tab 的位置为什么会记住？

子 Tab 的选择会自动保存到浏览器本地存储。这是为了方便多模块之间的频繁切换，不需要每次进入模块后重新选择子页面。如需重置，清除浏览器站点数据即可。

问：为什么大屏模型页面没有侧边栏？

大屏模型、工作流画布和日志查看器是全屏沉浸式页面，为了最大化操作空间会自动隐藏侧边栏和顶部导航。使用页面内的返回按钮即可退出沉浸模式。

问：Web 驾驶舱可以匿名访问吗？

可以。Web 3D 驾驶舱支持匿名访问，公开仪表盘无需登录即可查看。如需使用更多功能或查看详细数据，可在设置页登录账户。

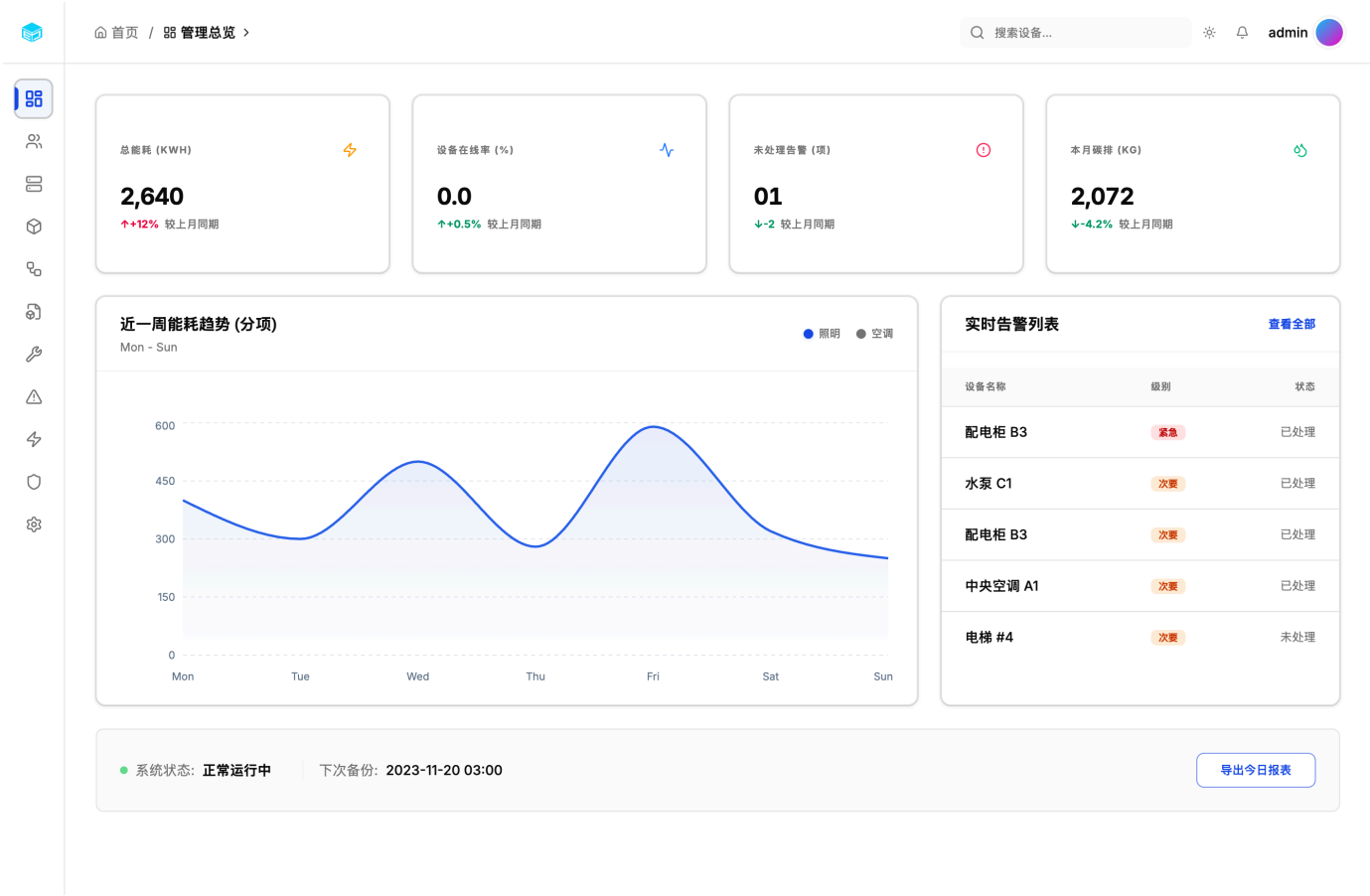
问：Bento Grid 布局能恢复默认吗？

可以。在编辑模式下点击「重置」按钮，即可将布局恢复到系统默认配置。

问：Web 驾驶舱的 SSE 连接断开怎么办？

当 SSE 连接断开时，顶部导航栏的状态指示器会变为红色。系统会自动尝试重连，状态指示器会变为琥珀色（重连中）。连接恢复后自动转为绿色。如长时间无法连接，请检查网络或联系管理员确认服务端状态。

Admin 管理总览



登录 Admin 后台后，默认进入管理总览页面。这里是平台的指挥中心，集中展示核心运行指标。

1. 指标卡片

页面顶部是四张指标卡片，每张显示当前数值与环比变化趋势：

卡片	指标	数据来源
⚡ 总能耗	当前周期总耗电量 (kWh)	<code>energy_readings</code> 聚合
📶 设备在线率	在线设备数 / 总设备数	IoT 心跳检测
🔴 未处理告警	当前未处理的故障/告警数	<code>alerts</code> + <code>faults</code> 表
🌿 本月碳排	本月碳排放总量 (tCO ₂)	能耗 × 碳排放因子计算

每张卡片右侧显示趋势箭头（↑ 上升 / ↓ 下降）和百分比变化，反映与上一周期的对比。

2. 能耗趋势图

页面中部是近一周能耗趋势面积图：

- X 轴：日期（最近 7 天）
- Y 轴：能耗值（kWh）
- 填充区域表示逐日变化趋势
- 支持切换显示粒度（日 / 时）

3. 实时告警列表

右侧面板列出最近的未处理告警：

字段	说明
设备名	触发告警的设备名称
级别	🔴 紧急 / 🟡 警告 / 🔵 信息
时间	告警触发时间
状态	待处理 / 处理中 / 已解决

点击「查看全部」跳转到 [实时故障监控](#) 页面。

4. 系统状态

页面底部系统状态区域显示：

- API 服务状态：在线 / 离线
- 数据库连接：正常 / 异常
- IoT 网关：已连接 / 断开（含图标指示）

- 下次自动备份时间：根据 [backupSchedule](#) 配置显示

提示：页面数据通过 REST API 定时刷新。刷新间隔可在 [系统配置](#) 中调整（10s / 30s / 1m / 5m）。

Web 3D 建筑驾驶舱



Web 公共门户（[apps/web](#)）在首页渲染一个全屏 Babylon.js 3D 建筑场景，叠加悬浮控制面板和仪表盘小组件。

1. 3D 场景

场景以一个 glTF/gLB 格式的三维建筑模型为基础，支持：

- 热点标签（Hotspot）：在建筑特定位置悬浮显示区域名称，点击触发交互
- 辉光效果（Glow Layer）：突出显示当前选中的区域
- 自动旋转相机：可配置旋转速度和初始角度
- 环境预设：切换不同 HDR 环境贴图（工作室 / 公园 / 黄昏等）

2. 相机控制

场景右上角浮动控制按钮：

操作	说明
 放大	拉近相机视角
 缩小	拉远相机视角
 复位	恢复默认相机位置和角度
 全屏	切换全屏模式（隐藏仪表盘覆盖层）

全屏模式下仪表盘小组件淡出，3D 场景占据整个视口，适合沉浸式浏览。

3. SSE 连接状态

页面左上角显示 SSE（Server-Sent Events）连接指示器：

-  绿色：已连接，实时数据正常推送
-  黄色：正在重新连接
-  红色：连接断开

连接使用短期 JWT 令牌（有效期 30 秒），由 `/api/events/token` 动态签发，心跳间隔 30 秒。

4. 区域导航

左侧面板列出建筑内的所有区域/楼层。点击某个区域：

- 相机平滑飞至该区域位置
- 该区域热点标签高亮显示
- 区域关联的数据在右侧面板更新

5. 时间与日期

页面顶部右侧显示当前日期和时间（使用浏览器本地时区）。

仪表盘小组件参考



Web 门户的仪表盘由多种类型的小组件组成，每个组件展示不同维度的数据。

1. StatCard（统计卡片）

核心 KPI 展示组件：

- 显示一个大号数值和单位
- 可选的 sparkline 迷你趋势图（显示最近一段时间的变化）
- 趋势方向指示 $\uparrow$ / $\downarrow$ / $\rightarrow$
- 点击卡片弹出详情对话框，展示更细粒度的数据

2. EnergyTrendChart（能耗趋势图）

支持三种图表类型：

类型	适用场景
 面积图	展示能耗的累计变化趋势
 折线图	展示精确的数值变化曲线
 柱状图	按时间段对比能耗数据

数据源： [energy_readings](#) 表按时间范围聚合。

3. EnergyBreakdownChart（能耗构成图）

饼图展示各分类能耗占比：

- 🌡 HVAC（暖通空调）
- 💡 照明
- ⚙ 设备
- 🔌 其他

颜色编码与 Admin 后台一致，便于跨系统对照。

4. 列表组件

组件	展示内容
DeviceList	设备列表：名称、状态（在线/离线）、最近活动时间
AlertList	告警列表：等级、设备、时间、状态图标
SuggestionList	优化建议：节能建议、运维提示

每种列表均支持点击展开查看详情。

5. WeatherWidget（天气组件）

展示当前位置的实时天气信息：

- 当前温度与天气图标
- 最高 / 最低温
- 湿度与风速

配置需求：[OPENWEATHER_API_KEY](#) 环境变量。未配置时该组件自动隐藏。

6. 数据源绑定

每个小组件的 `dataType` 字段决定其数据来源：

dataType	数据源	更新方式
stat	dashboard_stats 聚合	REST + SSE 推送
chart	energy_readings	REST + SSE 推送
list	alerts / devices / 自定义	REST
weather	OpenWeather API	REST 定时轮询
iot	IoT 点位值	REST 定时轮询

编辑仪表盘布局



Web 门户的仪表盘使用 Bento Grid 布局引擎，支持自由调整小组件的位置和尺寸。

1. 进入编辑模式

通过以下方式进入编辑模式：

1. 导航到 设置 → 布局，点击「编辑仪表盘布局」
2. 编辑模式下组件的边框变为虚线，左上角出现操作手柄

2. 拖拽重排

进入编辑模式后：

- 拖拽标题区域：移动组件到网格中的其他位置

- 其他组件会自动让位并重新排列
- 释放后布局立即生效

3. 调整尺寸

选中组件后，右下角出现尺寸调节手柄：

- 拖拽手柄调整组件的宽高
- 网格基于 CSS Grid 实现，尺寸单位是网格单元格
- 组件宽度范围：1-4 网格单元

4. 隐藏/显示组件

编辑模式下每个组件右上角有隐藏按钮：

- 点击后组件折叠为一个占位条
- 再次点击恢复显示
- 隐藏的组件在刷新后保持隐藏状态

5. 自动退出超时

编辑模式空闲一段时间后自动退出（防止误操作）：

可选超时	适用场景
15 秒	快速调整
30 秒	常规编辑
1 分钟	仔细排版
5 分钟	复杂布局
永不过期	长时间编辑（需手动退出）

可在 设置 → 布局 → 自动退出编辑延迟 中配置。

6. 恢复默认布局

点击「恢复默认布局」按钮：

1. 弹出确认对话框
 2. 确认后所有组件回到初始位置和尺寸
 3. 隐藏的组件重新显示
 4. 操作不可撤销
-

7. 布局持久化

当前布局自动保存到服务器（`dashboard_widgets` 表），与用户账户绑定。切换设备后同步加载，无需重新配置。

3D 区域标签与照明控制



1. 区域标签

Web 3D 场景中，建筑的不同区域上方悬浮着区域标签：

- 标签显示区域名称（如「A 区大堂」「B 区办公室」等）
- 标签位置由 Admin 后台的大屏模型 → 标签配置页面设置
- 鼠标悬停显示标签文字
- 点击标签触发交互

2. 照明控制面板

点击区域标签后，右侧弹出 LightingSheet 侧边面板（320px 宽）：

2.1. 面板内容

- 区域选择器：顶部下拉框，可切换查看不同区域的照明状态
- 照明分组：面板列出该区域下的所有照明分组

2.2. 每组的控制

每个照明分组显示：

- 组名称（如「A1 区-照明 1」「A1 区-照明 2」）
- 状态指示： 开 /  关
- 开关按钮：点击切换该组的开/关状态
- 支持三种状态：`on`（开） / `off`（关） / `half`（半开）

2.3. 批量操作

面板底部提供批量控制按钮：

按钮	操作
全部打开	该区域下所有照明组设为 <code>on</code>
全部关闭	该区域下所有照明组设为 <code>off</code>

2.4. 实时同步

技术细节：SSE 实时通信架构见 [实时通信 \(SSE\)](#)，IoT 集成见 [物联网集成](#)

照明状态变更通过 SSE 实时同步到其他设备：

- 在一个浏览器中开关照明，另一个浏览器中状态自动更新
- 更新通过 `lighting_update` 事件推送
- 状态变更经过服务端验证（REST API 调用，非本地乐观更新）

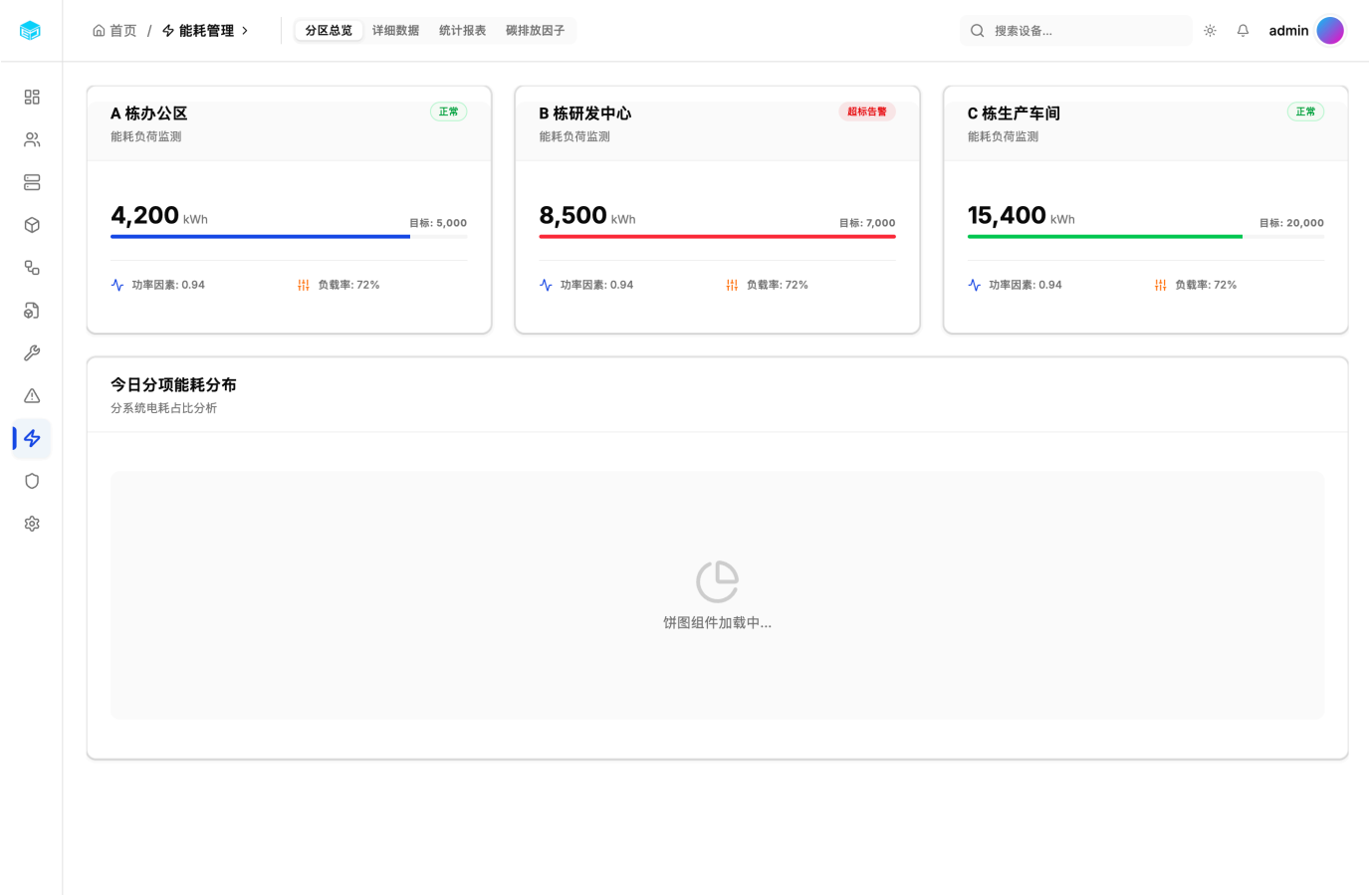
2.5. IoT 离线降级

当 IoT 网关不可用时（连接断开），照明控制自动降级为Mock 模式：

- 所有开关状态在服务端内存中模拟

- 开关功能仍然可用，但无法控制真实设备
- 页面顶部 Mock 模式指示条提醒操作员

分区域能耗总览



能耗管理页面的第一个子 Tab 展示建筑各区域的能耗概览。

1. 区域能耗卡片

页面上以卡片网格形式展示各区域能耗数据，每张卡片包含：

字段	说明
 区域名称	区域标题，如「A 区」「B 区」「C 区」
 当前值	当前周期的总能耗（kWh）
 目标值	该区域的能耗目标（kWh）
 进度条	当前值 / 目标值的百分比进度，颜色渐变（绿 → 黄 → 红）
 功率因素	cosφ 实时值，反映电能利用效率

字段	说明
⚙️ 负载率	当前负载 / 额定负载百分比

1.1. 超目标告警

当区域能耗超过目标值时，卡片右上角出现告警 Badge：

- 红色：超过目标值 110% 以上
- 黄色：超过目标值但未达 110%
- 绿色：在目标范围内

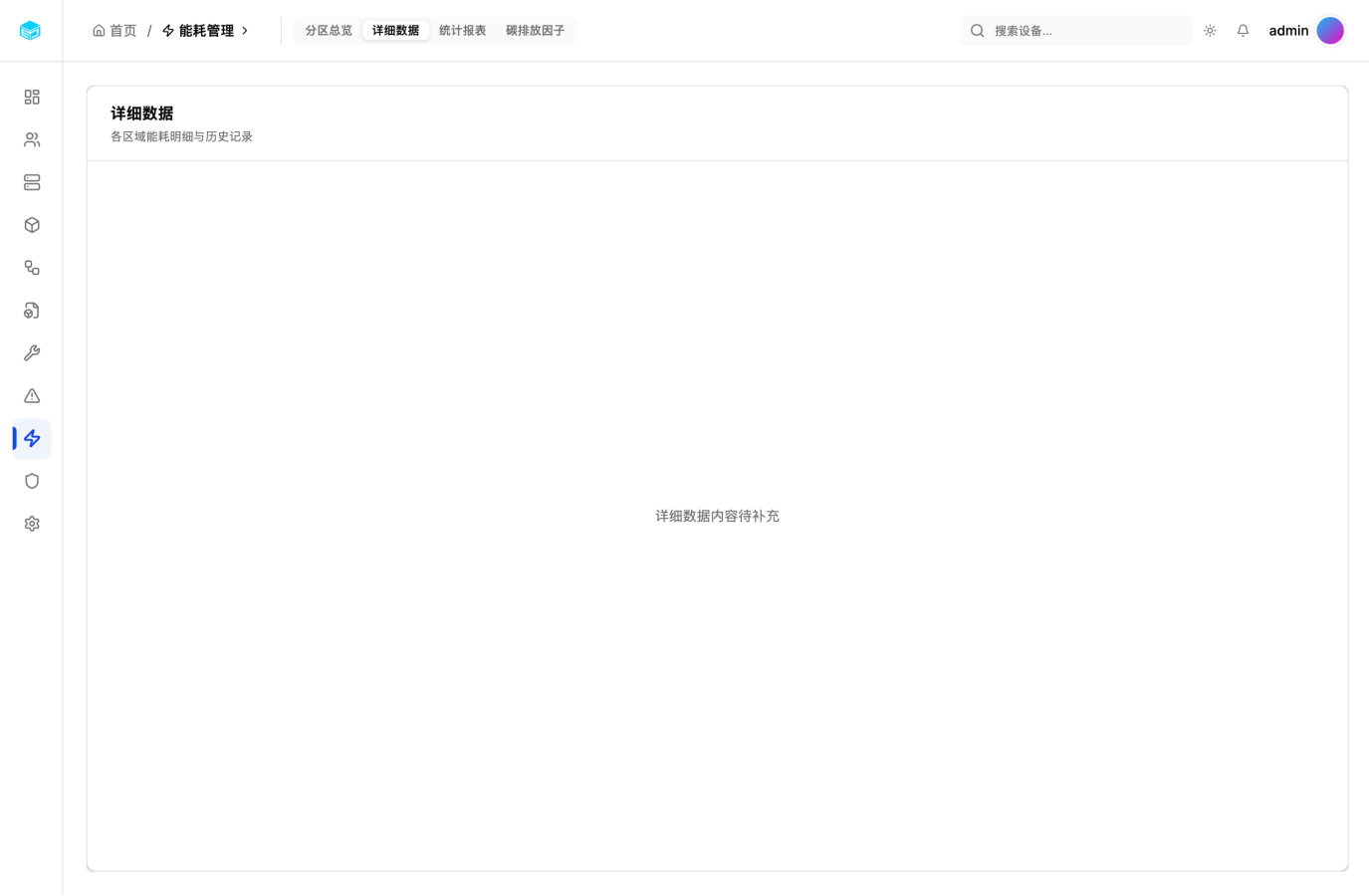
点击卡片进入该区域的详细数据页。

2. 数据源

区域数据来自 `energy_areas` 表的配置（目标值、功率因素基准等）和 `energy_readings` 表的实时聚合。

- 区域配置在 Admin 后台通过数据库初始化（当前无单独管理界面）
- 目标值通过 seed 数据或 API 更新

详细能耗数据



1. 数据视图

能耗管理的第二个子 Tab 提供细粒度的能耗数据表格视图。

1.1. 数据列

列	说明
时间	数据记录的时间戳
区域	所属区域
能耗值 (kWh)	该时间点的读数值
功率因素	该时间点的 cosφ

列	说明
负载率	该时间点的负载百分比

1.2. 筛选条件

页面顶部提供筛选栏：

- 时间范围：支持预设（今天 / 本周 / 本月 / 自定义）
- 区域：按区域名称筛选
- 粒度：按时间聚合（5 分钟 / 15 分钟 / 1 小时 / 1 天）

1.3. 数据导出

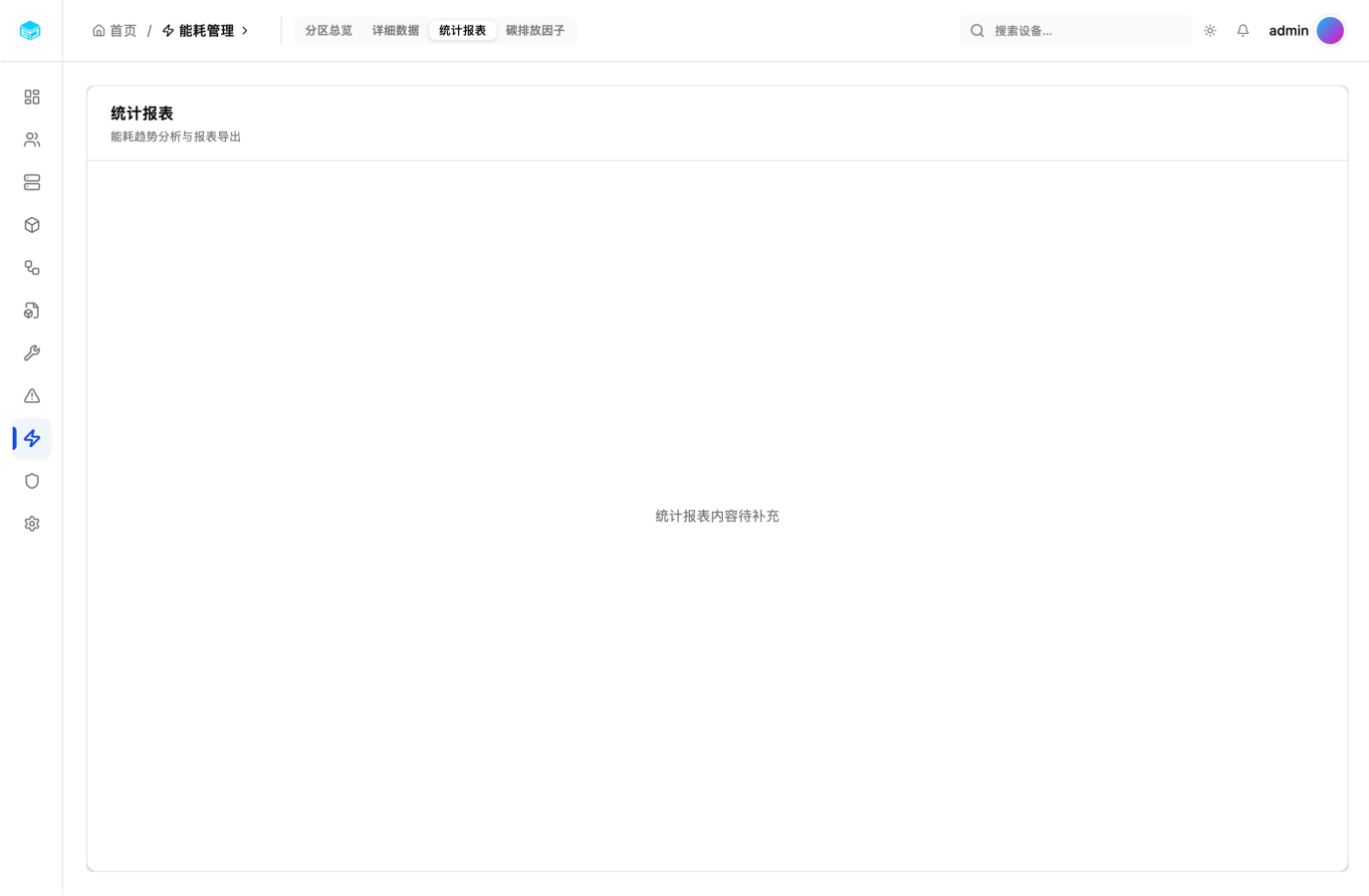
点击「导出当前数据」按钮，将当前筛选条件下的数据导出为 CSV 格式：

- 文件名自动包含时间范围（如 [energy-data_2026-07-01_2026-07-07.csv](#) ）
- 编码为 UTF-8 with BOM，兼容 Excel 中文打开
- 导出内容包含筛选条件下的完整数据列

1.4. 排序与分页

- 默认按时间降序排列（最新数据在前）
- 支持点击列头切换排序方向
- 后端分页，每页 50 条记录

能耗统计报表



1. 报表功能

能耗管理第三个子 Tab 提供自定义统计报表。

1.1. 报表参数

参数	选项
时间范围	自定义起止日期
聚合粒度	按日 / 按周 / 按月
区域	全部区域或指定区域
指标类型	能耗(kWh) / 碳排放(tCO ₂) / 功率因素 / 负载率

1.2. 报表展示

生成的报表以图表和表格两种方式展示：

- 趋势折线图：显示指标在时间范围内的变化趋势
- 数据表格：详细数值（日期、值、环比变化率）
- 汇总行：合计值、平均值、最大值、最小值

1.3. 报表解读

- 环比分析：表格中自动计算每行数据的环比变化百分比
- 异常标记：超过阈值的数据点高亮显示
- 空值处理：无数据的日期自动跳过，不中断趋势线

1.4. 导出

报表数据支持导出为 CSV 格式（与详细数据导出机制相同）。

碳排放因子管理



能耗管理的第四个（最后一个）子 Tab 提供碳排放因子的管理功能。

1. 界面布局

页面分为左右两个面板：

1.1. 左面板：因子分类树

- 可展开/折叠的分类树结构，按能源类型组织
- 支持搜索筛选（按名称或分类）
- 分类节点包括：电力、天然气、水、蒸汽等
- 点击分类节点，右面板显示该分类下的因子详情

1.2. 右面板：因子详情表格

每行因子包含：

字段	说明
因子名称	如「华东区域电网排放因子」
分类	所属分类
因子值	排放系数（如 kgCO ₂ /kWh）
单位	如 kgCO ₂ /kWh、tCO ₂ /万Nm ³
来源	数据来源说明

技术细节：碳排放因子系统设计见 [碳排放因子系统](#)

2. 数据刷新

页面提供「从国家温室气体排放因子库刷新」按钮：

- 1. 点击后系统请求最新的排放因子数据
- 2. 增量同步：新增的因子自动插入，已有的因子覆盖更新
- 3. 同步完成后页面自动刷新

2.1. 与能耗模块的联动

碳排放因子是能耗数据计算碳排放的基础：

$$\text{碳排放量} = \text{能耗值} \times \text{碳排放因子（按区域/按能源类型）}$$

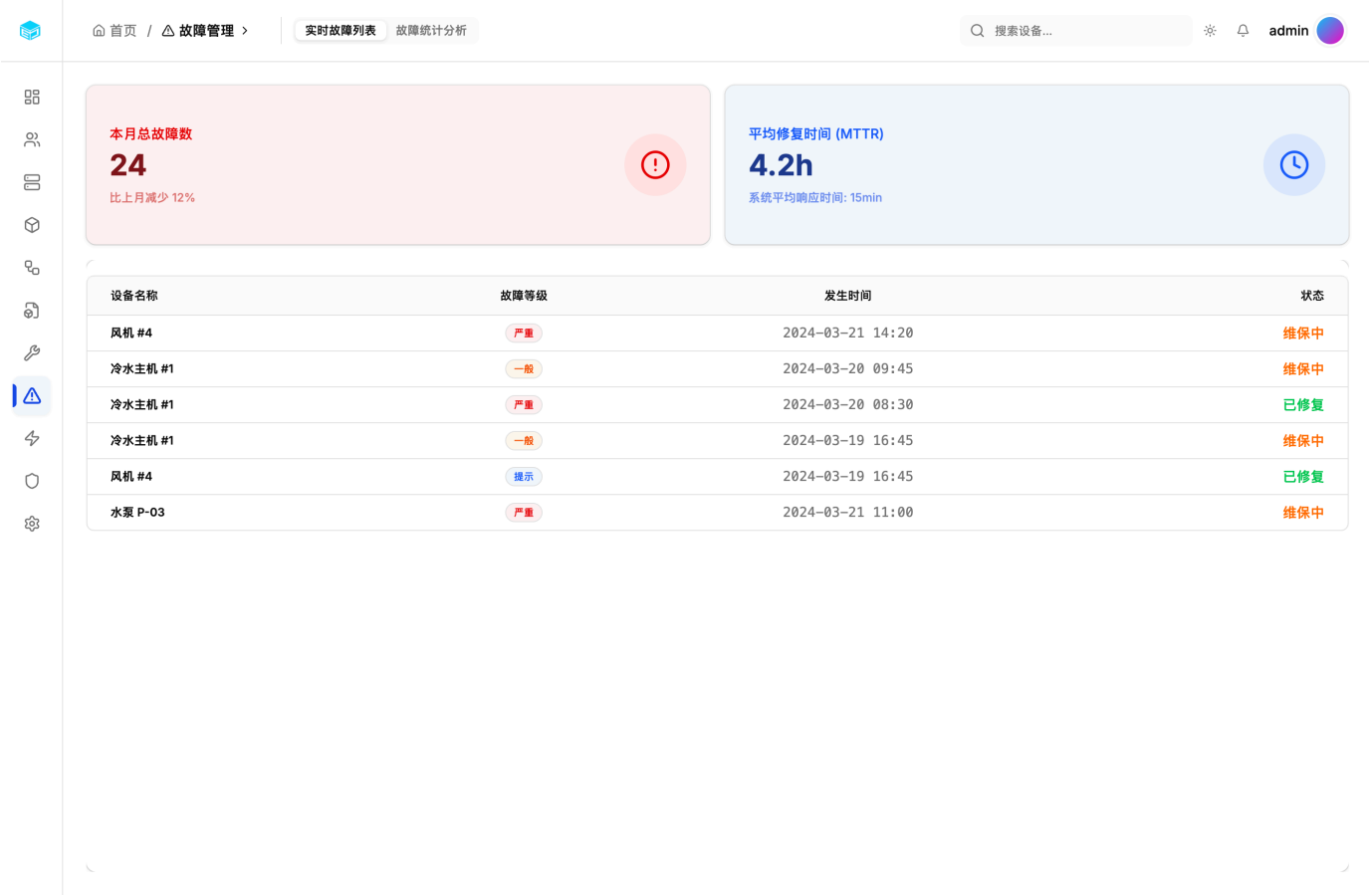
更新因子后，历史碳排放数据会自动重新计算，管理总览页面的碳排卡片会反映最新值。

3. 技术说明

- 因子数据存储在 `carbonFactorNodes`（分类树）和 `carbonFactors`（因子值）两张表中
- 分类树支持多级嵌套

- 系统通过 API 与外部排放因子数据库通信（需网络连通）

实时故障监控



1. 故障列表

页面以表格形式展示所有故障/告警记录：

字段	说明
设备名	触发故障的设备
故障等级	🔴 紧急 / 🟡 警告 / 🔵 信息
故障时间	触发时间戳
状态	待处理 / 处理中 / 已解决

1.1. 筛选

页面顶部提供筛选栏：

- 时间范围：最近 1 小时 / 今天 / 本周 / 本月 / 自定义
- 故障等级：多选筛选项
- 设备名称：文本搜索

1.2. 快速跳转

从 [管理总览](#) 顶部的「未处理告警」区域点击「查看全部」可直接跳转到此页，并自动应用当前筛选。

2. 操作

列表中每条记录支持：

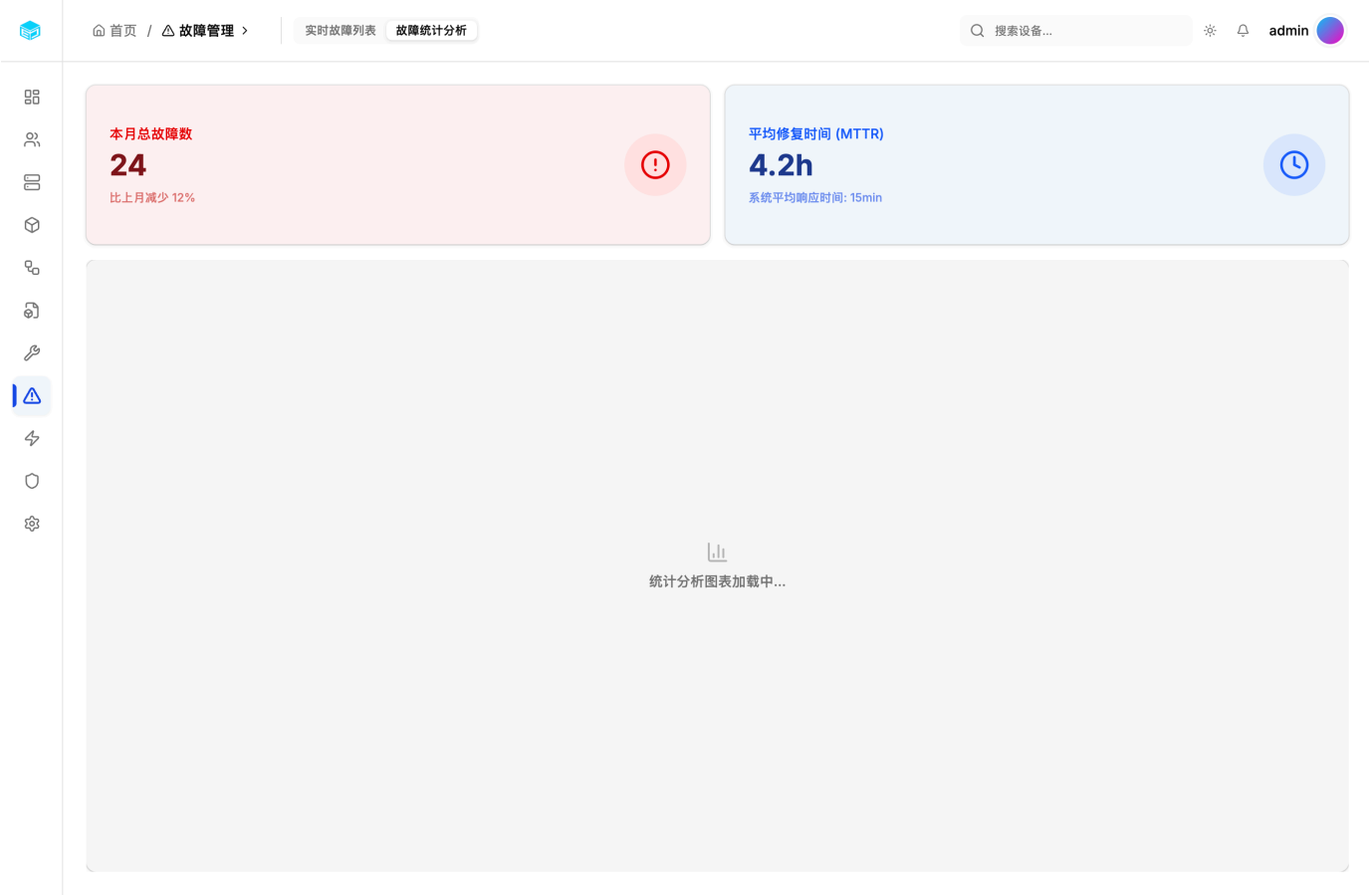
- 点击行：展开故障详情（设备信息、故障描述、建议处理方案）
- 状态变更：处理中 → 已解决（确认处理后更新状态）

3. 数据源

故障数据来自 `faults` 表和 `alerts` 表的联合查询：

- `faults`：设备故障记录
- `alerts`：系统告警规则触发的告警

故障统计与分析



1. 统计卡片

页面顶部显示统计摘要：

指标	说明
 本月总故障数	当前月份累计故障数量
 平均修复时间 (MTTR)	从「处理中」到「已解决」的平均耗时
 同比变化	与上月相比的变化百分比

2. 故障趋势图

页面中部展示故障数量趋势：

- X 轴：日期（最近 30 天）
- Y 轴：故障数量
- 折线图显示每日新增故障数
- 可切换时间维度（按日 / 按周）

3. 按等级分布

饼图展示本月故障的等级分布：

- ● 紧急故障占比
- ● 告警占比
- ● 信息类占比

各等级带颜色编码，便于直观识别。

4. 热门故障设备

底部表格列出本月故障次数最多的设备 Top 10：

设备名	故障次数	主要故障类型	平均修复时间
...	...	...	...

技术细节：实时数据推送基于 SSE，架构见 [实时通信\(SSE\)](#)

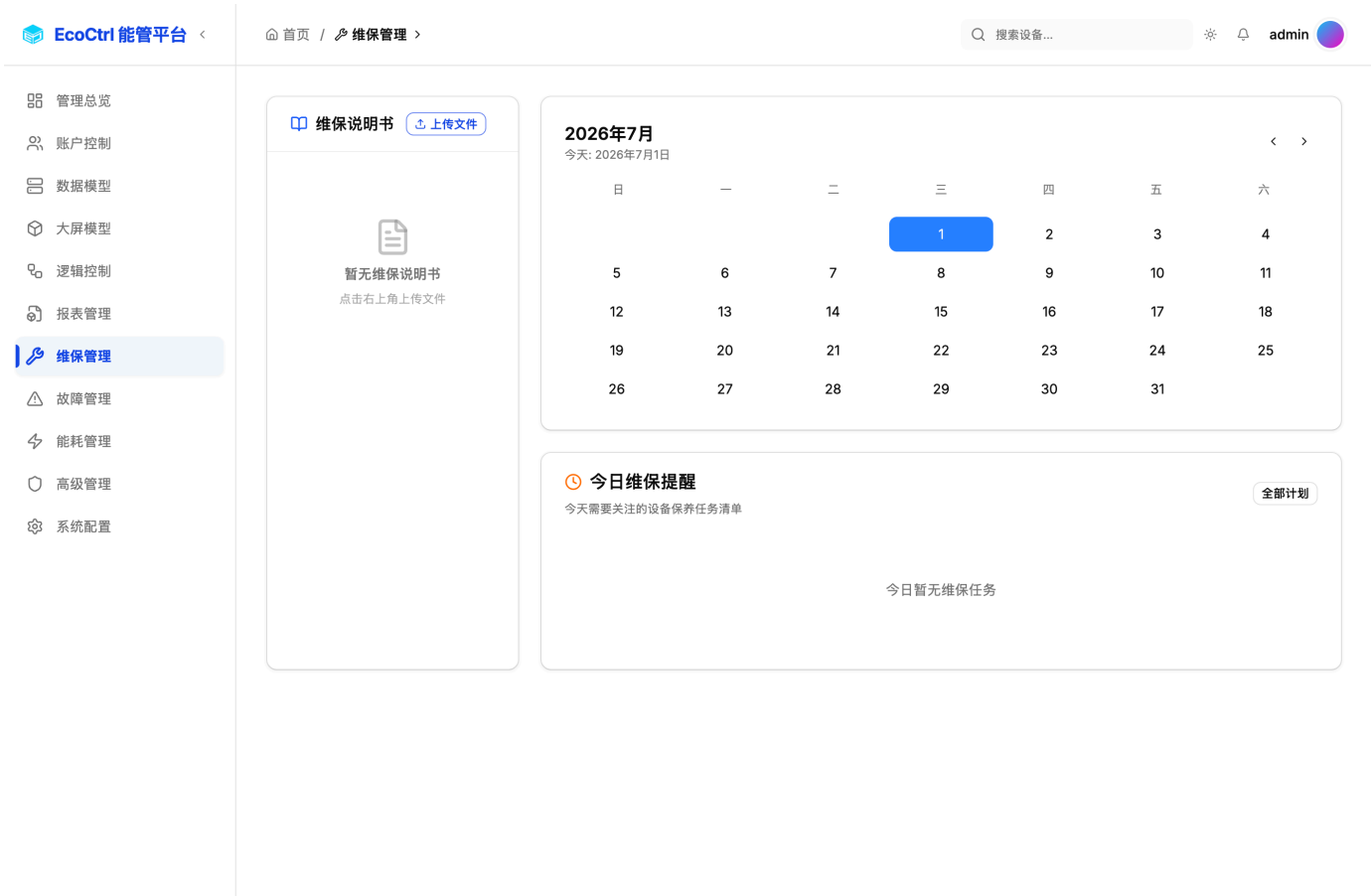
5. 数据源

统计数据来自 `fault_stats` 表的聚合查询：

- 故障数量通过 `faults` 表实时统计
- MTTR 通过各故障记录的处理时间差值计算

- 趋势数据按时间范围聚合

维保说明书管理



1. 界面布局

页面分为左右两个面板：

1.1. 左面板：说明书文件列表

- 列出已上传的维保说明书文件
- 每个文件项显示文件名和上传日期
- 支持操作：
 - 点击文件：在新标签页中预览 PDF
 - 双击文件名：进入内联重命名模式
 - Enter 确认重命名
 - Escape 取消

- 删除按钮：弹出确认对话框后删除文件
- 上传按钮：选择本地 PDF 文件上传

1.2. 上传多份 PDF

- 支持一次上传多份 PDF 文件
- 限制：单文件不超过 100MB
- 上传进度条显示
- 上传完成后文件列表自动刷新

1.3. 右面板：日历月视图

- 显示当前月份的日历
- 日期上标记当天关联的维护事件（如有）
- 点击日期切换查看当天的维护记录
- 使用 `react-day-picker` 实现

2. 数据存储

- 文件存储在后端配置的存储后端中（开发环境为本地磁盘，生产环境为 MinIO S3 对象存储）
- 元数据存储在 `files` 表（文件名、路径、大小、上传时间）
- 维护记录存储在 `maintenance` 表

1. 工作流列表

字段	说明
名称 + 标签	工作流名称和自定义标签
触发器类型	 手动 /  定时 /  Webhook /  事件
启用状态	Switch 开关切换启用/禁用
版本	当前 DSL 版本号
更新时间	最后修改时间

1.1. 启用/禁用

- 使用行内 Switch 开关直接切换工作流的启用状态
- 禁用的工作流不会响应触发器，不会被执行

1.2. 手动触发

- 点击行右侧的「触发」按钮立即执行一次工作流
- 适用于测试工作流逻辑
- 执行记录会出现在 [执行记录](#) 中

技术细节：工作流引擎架构与 DSL 参考见 [工作流引擎](#)

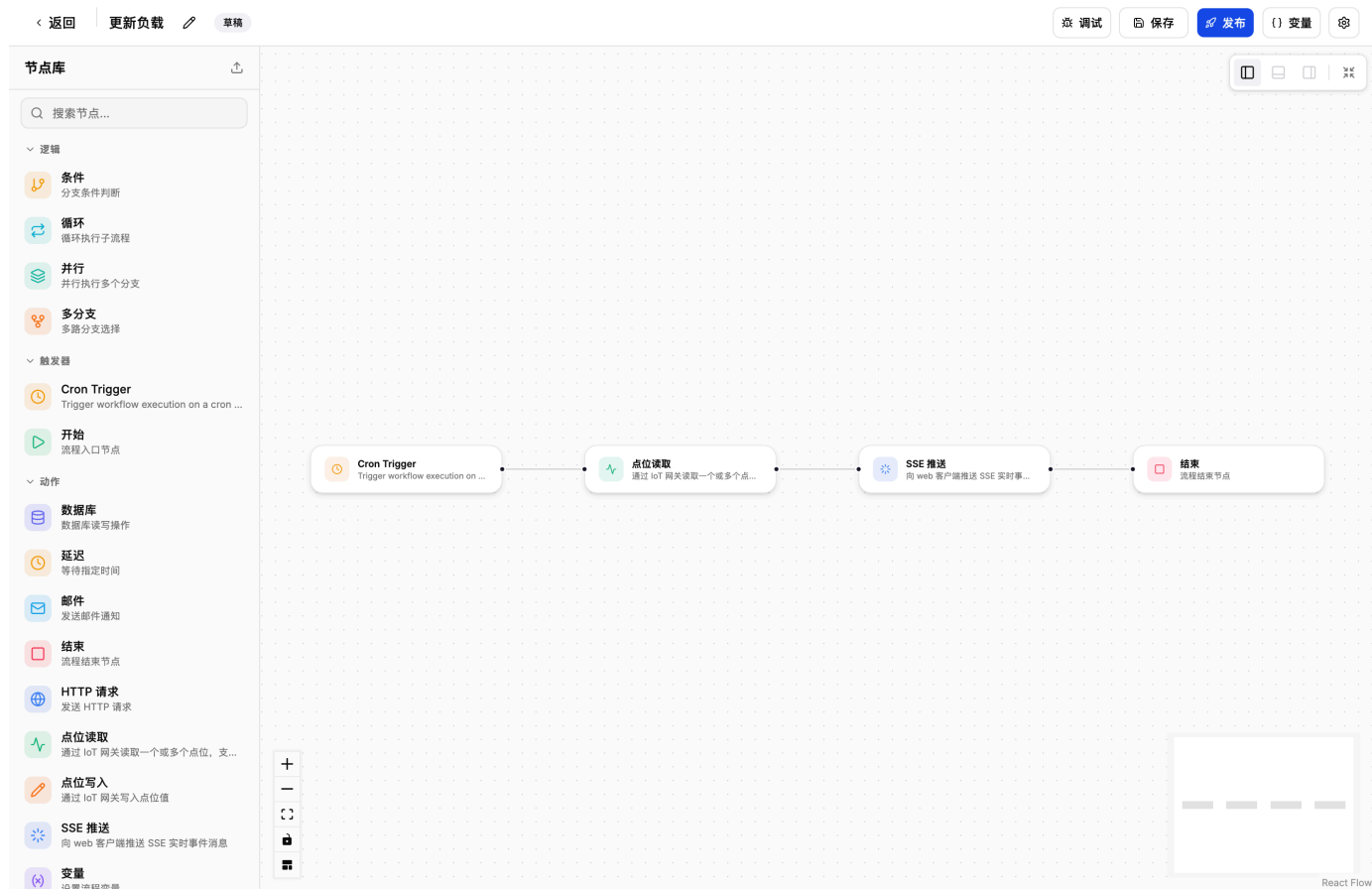
2. 触发器类型

类型	说明
manual	手动触发，用户点击按钮执行
schedule	定时触发，基于 cron 表达式配置
webhook	外部系统通过 HTTP 请求触发（POST /api/webhook/:slug）
event	系统内部事件触发（如设备告警、能耗阈值超限）

3. 新建工作流

点击「新建工作流」按钮，创建工作流后进入可视化编辑器。

可视化工作流编辑器



点击工作流或新建工作流后，进入全屏沉浸式编辑器，基于 ReactFlow（XYFlow）实现。

1. 界面布局

- 全屏模式：隐藏侧边栏和头部，最大画布空间
- 左上角：返回按钮 + 工作流名称（可内联编辑）
- 右上角：保存/另存为按钮
- 左侧面板：节点库 — 可拖拽的节点类型列表
- 中央：空白画布

2. 创建流程

1. 从左侧节点库拖拽节点到画布

2. 从一个节点的输出手柄拖拽到另一个节点的输入手柄，创建连线
 3. 唯一约束：必须有且仅有一个 Start 节点和一个 End 节点
 4. 连线不允许形成闭环（DAG 约束）
-

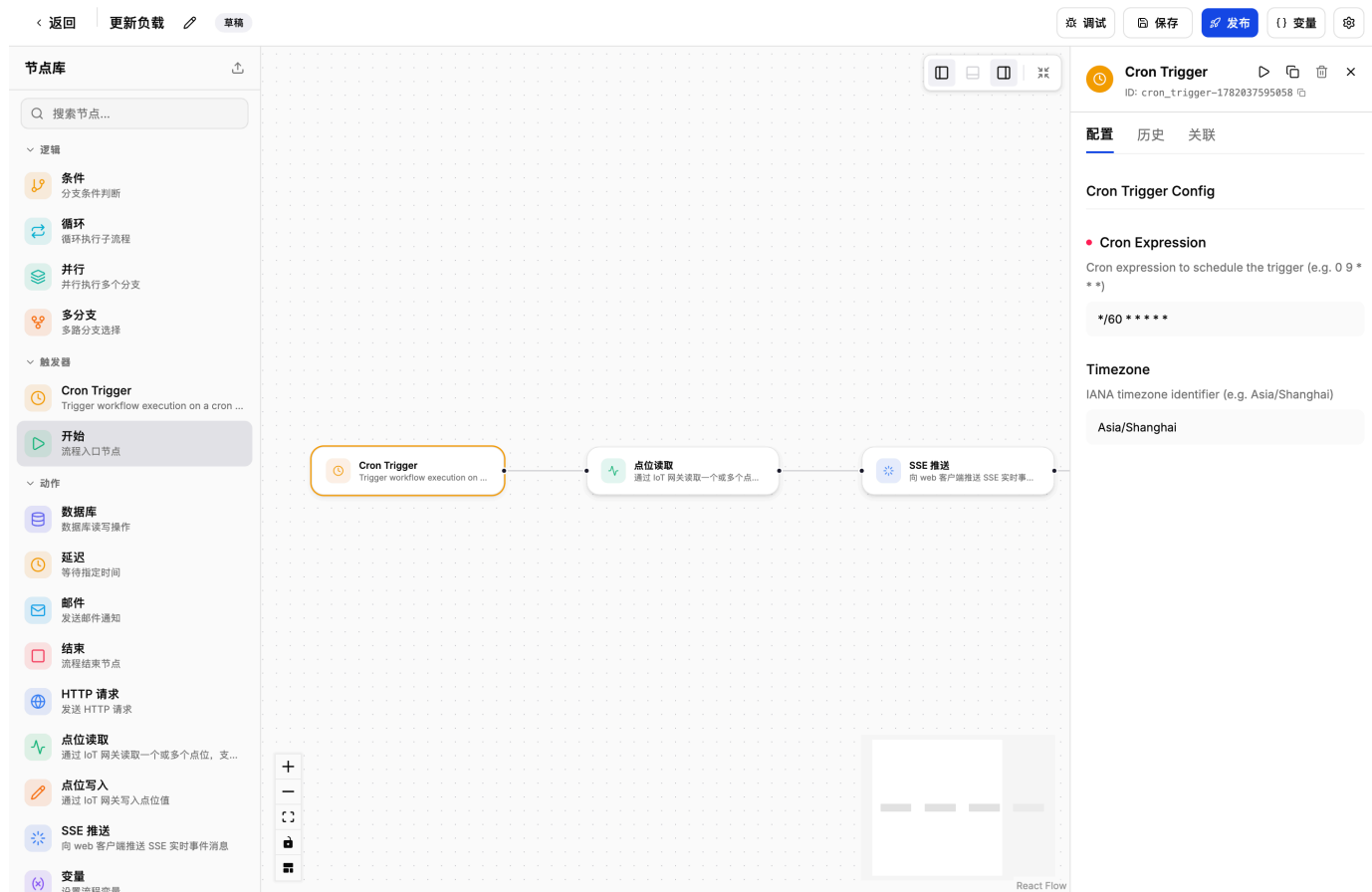
3. 编辑工作流名称

- 点击工作流名称进入内联编辑模式
 - Enter 确认修改
 - Escape 取消修改
-

4. 保存

- 点击保存按钮将当前工作流 DSL 持久化到 `workflows` 表
- 保存后版本号递增
- 未保存的更改有离开提示

配置 workflow 节点



1. 节点配置面板

点击画布上的节点，右侧弹出 NodeConfigPanel。

1.1. 通用配置

所有节点共享的配置：

- 节点名称：自定义显示名称
- 描述：节点功能的文字说明

1.2. 节点类型

workflow 引擎基于插件架构，节点类型由已安装的插件动态决定。核心节点类型：

控制节点

节点	功能
Start	工作流入口，仅一个
End	工作流出口，仅一个
Condition	条件分支（if/else）
Switch	多路分支（switch/case）
Loop	循环执行子流程
Parallel	并行执行多个分支
Delay	等待指定时间后继续

动作节点

节点	功能
HTTP Request	发送 HTTP 请求到外部系统
Database	执行数据库查询
Email	发送邮件
Variable	设置或修改变量

1.3. 错误处理

每个节点可配置 onError 策略：

策略	行为
retry	重试（可配置重试次数和间隔）
skip	跳过当前节点继续执行
abort	中止整个工作流
goto	跳转到指定节点继续执行

1. 执行记录列表

字段	说明
工作流 ID	关联的工作流标识
执行 ID	本次执行的唯一标识
状态	 成功 /  失败 /  运行中
耗时	执行总耗时 (ms)
时间	执行开始时间

工作流的第二个子 Tab 展示最近 50 条执行记录。

[illegible]

1.1. 实时追加

- 有新的执行开始时，通过 SSE 实时推送到列表顶部
- 无需手动刷新页面
- 正在进行的执行实时更新状态

1.2. 手动刷新

列表顶部提供「刷新」按钮。

2. 执行详情

点击某条记录的「详情」按钮，打开全屏 LogViewer：

- 显示 workflow 执行时的完整节点执行日志
 - 每个节点的执行状态（运行中 / 已完成 / 失败 / 跳过）
 - 节点的输入参数和输出结果
 - 错误堆栈（如有）
 - 执行时间线
-

3. 删除记录

- 单条删除：点击删除按钮 → 确认对话框
 - 批量删除：勾选多条记录 → 点击「批量删除」按钮
 - 删除操作不可撤销
-

4. 数据源

执行记录来自 `workflow_executions` 表：

- 执行开始 → 插入记录，状态为 `running`
- 执行完成 → 更新状态和耗时

- 节点日志存储在执行上下文中
-

5. 常见错误排查

workflow 执行失败时，请检查 [监控与日志](#) 中的 pg-boss 队列状态和 SSE 连接健康。

数据模型管理

☰

🔍

📁

🔗

📄

🔧

⚠️

⚡

🛡️

⚙️

🏠 首页 / 📁 数据模型 >

数据模型 业务对象 点位信息

🔍 搜索设备...

⚙️ 🔔 admin

📄 点位信息

查看所有数据模型中的点位定义信息。

🔍 筛选点位...

+ 新建点位

📶 导入点位

☐	点位类型	编码	点位名称	分区	系统	属性数量	关联对象	关联模型	操作
☐	BV	0002	C_5001_BV_0002	南序厅	照明分组	8	5001	C	
☐	BV	0006	C_5001_BV_0006	南序厅	照明分组	8	5001	C	
☐	BV	0007	C_5001_BV_0007	南序厅	照明分组	8	5001	C	
☐	BV	0055	C_5001_BV_0055	南序厅	照明分组	8	5001	C	
☐	BV	0037	C_5001_BV_0037	南序厅	照明分组	8	5001	C	
☐	BV	0022	C_5001_BV_0022	南序厅	照明分组	8	5001	C	
☐	BV	0023	C_5001_BV_0023	南序厅	照明分组	8	5001	C	
☐	BV	0039	C_5001_BV_0039	南序厅	照明分组	8	5001	C	
☐	BV	0053	C_5001_BV_0053	南序厅	照明分组	8	5001	C	
☐	BV	0005	C_5001_BV_0005	南序厅	照明分组	8	5001	C	

批量删除

共 317 条

10

<

1

2

3

4

5

...

32

>

数据模型页面的第一个子 Tab，管理 3D 模型文件。

1. 模型卡片网格

页面以卡片网格形式展示所有上传的 3D 模型：

- 每张卡片显示模型名称和缩略图（如果没有缩略图则显示模型图标）
- 卡片右上角显示版本号标签

1.1. 操作

操作	说明
点击卡片	跳转到关联的 业务对象 页面（该模型下的对象列表）

操作	说明
预览按钮	打开 ModelViewer 弹窗，在浏览器中渲染 3D 模型预览
编辑按钮	编辑模型信息（名称、版本、编码、描述）
删除按钮	弹出确认对话框后删除模型

1.2. ModelViewer 弹窗

- 全屏弹窗使用 Babylon.js 渲染
- 支持旋转、缩放查看模型
- 自动适配模型尺寸到视口

2. 模型数据

每个模型包含：名称、版本、编码、文件路径、描述

数据来源：`models` 表。

业务对象管理

☰

🔍

📄

🔗

📁

🔧

⚠️

⚡

🔄

⚙️

🏠 首页 / 📁 数据模型 >

数据模型 业务对象 点位信息

🔍 搜索设备...

admin

业务对象

基于模型创建业务对象实例，为点位属性赋值。

🔍 筛选对象...

+ 新增对象

☐	编码 ↑↓	对象名称 ↑↓	关联模型 ↑↓	操作
☐	5001		C	
☐	5930-5		C	
☐	5930-6		C	
☐	5930-7		C	
☐	5930-8		C	
☐	5930-9		C	
☐	5931-12		C	
☐	5931-13		C	
☐	5931-14		C	
☐	5931-15		C	

批量删除

共 39 条

10

<

1

2

3

4

>

数据模型页面的第二个子 Tab，管理业务对象（Business Object）。

1. 对象列表

以表格形式展示业务对象：

字段	说明
编码	业务对象编码（全局唯一）
名称	对象名称
关联模型	所属数据模型，显示模型名称和版本

1.1. 搜索筛选

搜索框支持按编码或名称模糊匹配。

2. 新增业务对象

点击「新增对象」按钮打开对话框：

字段	必填	说明
选择模型	是	从已有数据模型下拉选择
编码	是	唯一编码，不可重复
名称	是	对象名称
描述	否	对象说明

3. 编辑与删除

- 编辑：点击行右侧编辑按钮，弹窗修改
- 删除：点击删除按钮 → 确认对话框「此操作不可撤销，关联的业务对象将被清除」

4. 业务对象与模型的关系

一个数据模型可以关联多个业务对象。例如：

- 模型：「AHU-01 空调机组」
- 业务对象：「A区 AHU-01」「B区 AHU-02」等

来自 `objects` 表，通过 `modelId` 关联到 `models` 表。

点位管理

☰

🔍

📁

🔗

📄

🔧

⚠️

⚡

🛡️

⚙️

🏠 首页 / 📁 数据模型 >

数据模型 业务对象 点位信息

🔍 搜索设备...

🌟 🔔 admin

📄 点位信息

查看所有数据模型中的点位定义信息。

🔍 筛选点位...

+ 新建点位

📶 导入点位

☐	点位类型	编码	点位名称	分区	系统	属性数量	关联对象	关联模型	操作
☐	BV	0002	C_5001_BV_0002	南序厅	照明分组	8	5001	C	
☐	BV	0006	C_5001_BV_0006	南序厅	照明分组	8	5001	C	
☐	BV	0007	C_5001_BV_0007	南序厅	照明分组	8	5001	C	
☐	BV	0055	C_5001_BV_0055	南序厅	照明分组	8	5001	C	
☐	BV	0037	C_5001_BV_0037	南序厅	照明分组	8	5001	C	
☐	BV	0022	C_5001_BV_0022	南序厅	照明分组	8	5001	C	
☐	BV	0023	C_5001_BV_0023	南序厅	照明分组	8	5001	C	
☐	BV	0039	C_5001_BV_0039	南序厅	照明分组	8	5001	C	
☐	BV	0053	C_5001_BV_0053	南序厅	照明分组	8	5001	C	
☐	BV	0005	C_5001_BV_0005	南序厅	照明分组	8	5001	C	

批量删除

共 317 条

10

<

1

2

3

4

5

...

32

>

数据模型页面第三个子 Tab，管理 IoT 数据点位（Point）。

1. 点位列表

以表格形式展示所有点位：

字段	说明
类型	点位类型（sensor / actuator / meter 等）
编码	点位唯一编码
名称	点位中文名称
分区	点位所在分区
系统	点位所属系统

字段	说明
属性数	该点位关联的属性数量
关联对象	关联的业务对象名称
关联模型	关联的数据模型名称

1.1. 搜索筛选

搜索框支持按编码、名称、分区、系统多维度筛选。

2. 新建点位

点击「新建点位」打开对话框：

字段	必填	说明
选择模型对象	是	先选择模型，再选择该模型下的对象
类型	是	sensor / actuator / meter 等
编码	是	唯一编码
名称	是	点位名称
描述	否	点位说明
分区	是	点位所在区域
系统	是	所属子系统

3. 批量导入

支持通过文件批量导入点位数据：

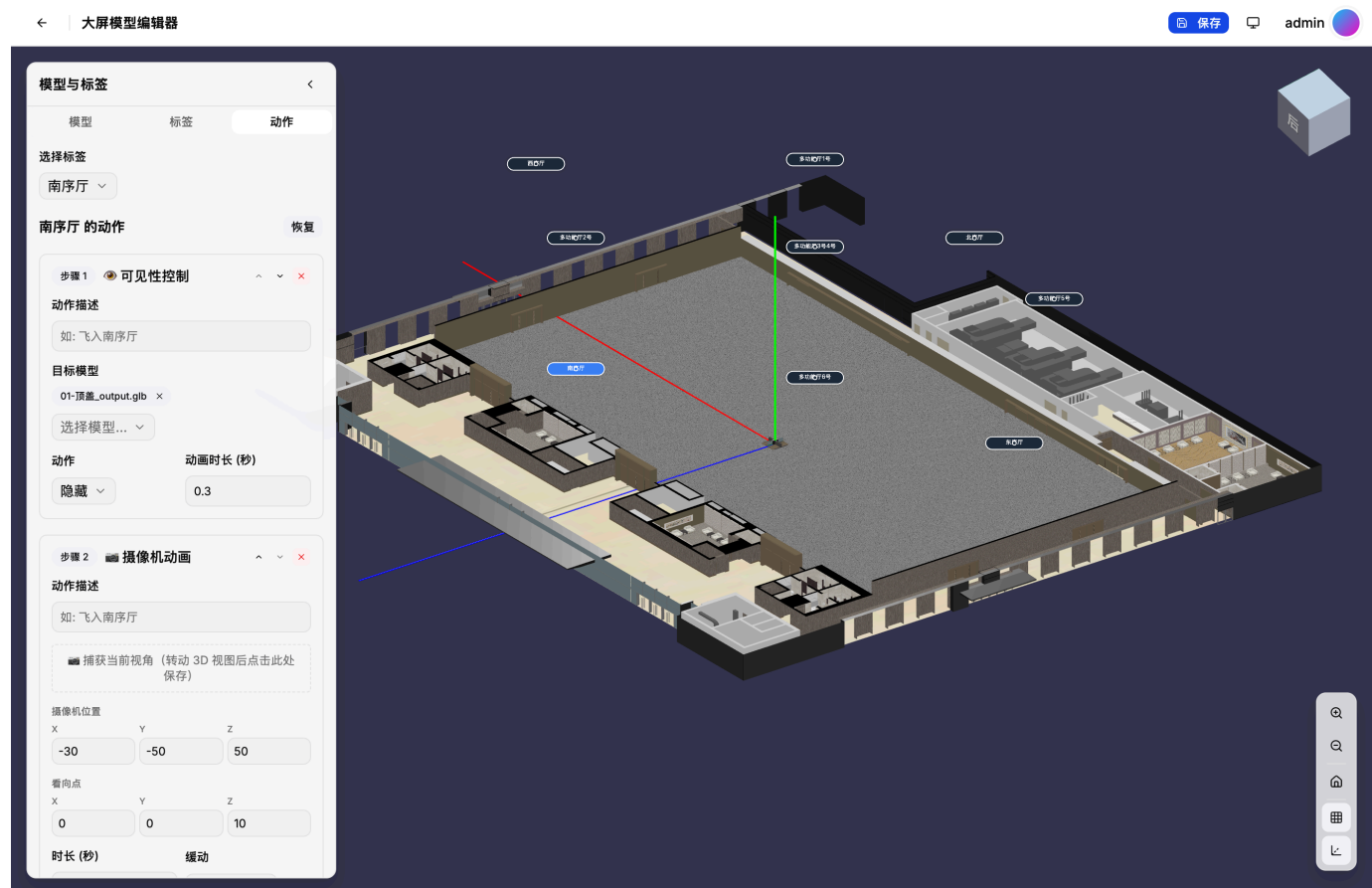
- 格式：.json / .csv / .xlsx
- 点击「批量导入」选择文件

- 系统校验字段完整性，自动跳过重复数据
 - 导入完成后显示成功/失败计数
-

4. 数据源

点位数据来自 `points` 表，通过 `objectId` 关联 `objects` 表，再通过 `modelId` 关联 `models` 表。

3D 模型编辑器



「大屏模型」Tab (DashboardModel) 是一个全屏沉浸式页面入口，点击后进入全屏场景编辑器。

1. 编辑模式

页面进入全屏模式（隐藏侧边栏和头部），右上角提供「返回」按钮回到数据模型页。

编辑器有三种模式，通过模式切换器选择：

1.1. 🏗️ 模型模式

- 管理已上传 3D 模型文件的排列
- 使用拖拽排序功能调整模型在场景中的加载顺序
- 支持上传新的模型文件

1.2. 标签模式

- 在 3D 场景中配置热点标签
- 每个标签绑定到一个位置 (x, y, z 坐标)
- 标签名称、颜色、关联的区域/数据可编辑

1.3. 动作模式

- 配置 3D 场景中的交互动作
- 点击热点标签时的行为 (飞至、弹窗、数据面板)
- 相机运动的路径和速度

2. 上传模型

设置项	值
支持格式	.glb / .gltf / .obj
每模型文件数上限	最多 10 个文件
单文件限制	100MB
上传区域	支持拖拽文件到上传区 + 点击选择文件

上传完成后模型出现在文件列表中，可拖拽排序。

3. 文件管理

- 拖拽排序：调整模型文件在场景中的加载顺序
- 删除文件：移除不需要的模型
- 自动保存：修改后配置自动保存到 `dashboard_models` 表

定时报表计划

EcoCtrl 能管平台

管理总览

账户控制

数据模型

大屏模型

逻辑控制

报表管理

维保管理

故障管理

能耗管理

高级管理

系统配置

首页 / 报表管理

搜索设备...

admin

管理模板 新建定时任务 导出当前数据

定时发送计划

配置自动生成的周期性数据概览并发送至指定邮箱。

报表名称	收件人	发送频率	状态控制
季度审计报表	sustainability@energy.com	每季度末	☒
设备故障率分析	audit@energy.com	每月 1 日	☒
季度审计报表	maintenance@energy.com	每周一	☐
季度审计报表	sustainability@energy.com	每周一	☒

能耗日报
1,245 份

月度汇总
48 份

异常分析
12 份

1. 计划列表

报表管理页面以列表形式展示所有定时报表计划：

字段	说明
报表名称	计划名称，自定义
收件人	报表发送的目标邮箱
发送频率	每天 / 每周 / 每月
状态开关	Switch 切换启用/禁用

2. 新建报表计划

点击「新建计划」按钮打开对话框：

字段	必填	说明
报表名称	是	计划名称
收件人邮箱	是	CSV 格式支持多个邮箱
发送频率	是	每天 / 每周（指定星期几） / 每月（指定日期）
状态	否	新建后默认启用

3. 模板管理

点击「管理报表模板」打开 TemplateDialog：

- 模板列表：展示已创建的模板（名称、格式、最后修改时间）
- 新建模板：定义报表的内容结构、数据源和布局
- 模板格式：支持表格和图表两种布局
- 模板与计划关联：一个模板可被多个计划使用

4. 技术说明

- 报表生成通过 pg-boss 队列异步执行
- 定时发送依赖队列调度器
- 生成完成后自动通过 SMTP 发送到目标邮箱

数据导出



1. 导出当前数据

在能耗管理的 [详细数据](#) 和 [统计报表](#) 页面中，点击「导出当前数据」按钮。

1.1. 导出格式

- 格式：CSV（逗号分隔值）
- 编码：UTF-8 with BOM（兼容 Excel 中文打开）
- 文件名：自动包含数据范围（如 `energy-data_2026-07-01_2026-07-07.csv`）

1.2. 导出内容

导出数据包含当前筛选条件下所有可见数据列：

列	说明
时间	数据记录时间戳
区域	所属区域
能耗值	耗电量（kWh）
功率因素	实时功率因素
负载率	负载百分比

2. 数据完整性

- 导出数据不包含分页限制（当前页 + 所有页）
- 时间范围限制仍然生效
- 导出前可预览数据范围

用户账户管理

☰

👤

📁

🏠

🔍

🔗

🔒

⚡

🔧

🏠 首页 / 👤 账户控制 >

🔍 搜索设备...

⚙️ 🔔 admin

+ 新增用户

📄 导出表格

🔍 搜索用户...

用户列表

姓名	邮箱	角色	状态	操作
张三	horned_gianncarlo@gmail.com	分析师	禁用	✎ 🗑
周八	reputed_renard@free.fr	管理员	繁忙	✎ 🗑
赵六	nefarious_abir@msn.com	管理员	离线	✎ 🗑
李四	prim_nala@ymail.com	超级管理员	禁用	✎ 🗑
吴九	religious_jessey@gmail.com	分析师	离线	✎ 🗑
ecoctrl	ecoctrl@example.com	超级管理员	在线	✎ 🗑
admin	admin@example.com	超级管理员	在线	✎ 🗑

账户控制页面（Accounts）用于管理平台所有用户。

1. 用户列表

以表格形式展示用户数据：

字段	说明
用户名	登录使用的用户名
邮箱	用户绑定的邮箱地址
角色	角色标签（admin / manager / operator / viewer），颜色区分
状态	● 在线 / ○ 离线（基于活跃会话检测）

1.1. 搜索筛选

页面顶部搜索框支持按以下维度筛选：

- 用户名：模糊匹配
- 邮箱：模糊匹配
- 角色：下拉选择角色类型

2. 新增用户

点击「新增用户」按钮打开弹窗：

字段	必填	说明
用户名	是	登录用户名，全局唯一
邮箱	是	用户邮箱，用于验证码发送
密码	是	登录密码
角色	是	从角色列表中选择

3. 编辑用户

点击用户行打开 UserSheet 滑出面板（右侧滑入 400px）：

- 用户名：可编辑
- 邮箱：可编辑
- 角色：下拉选择（admin / manager / operator / viewer）
- 状态：启用/禁用切换（禁用后用户无法登录）
- 修改后自动保存

4. 删除用户

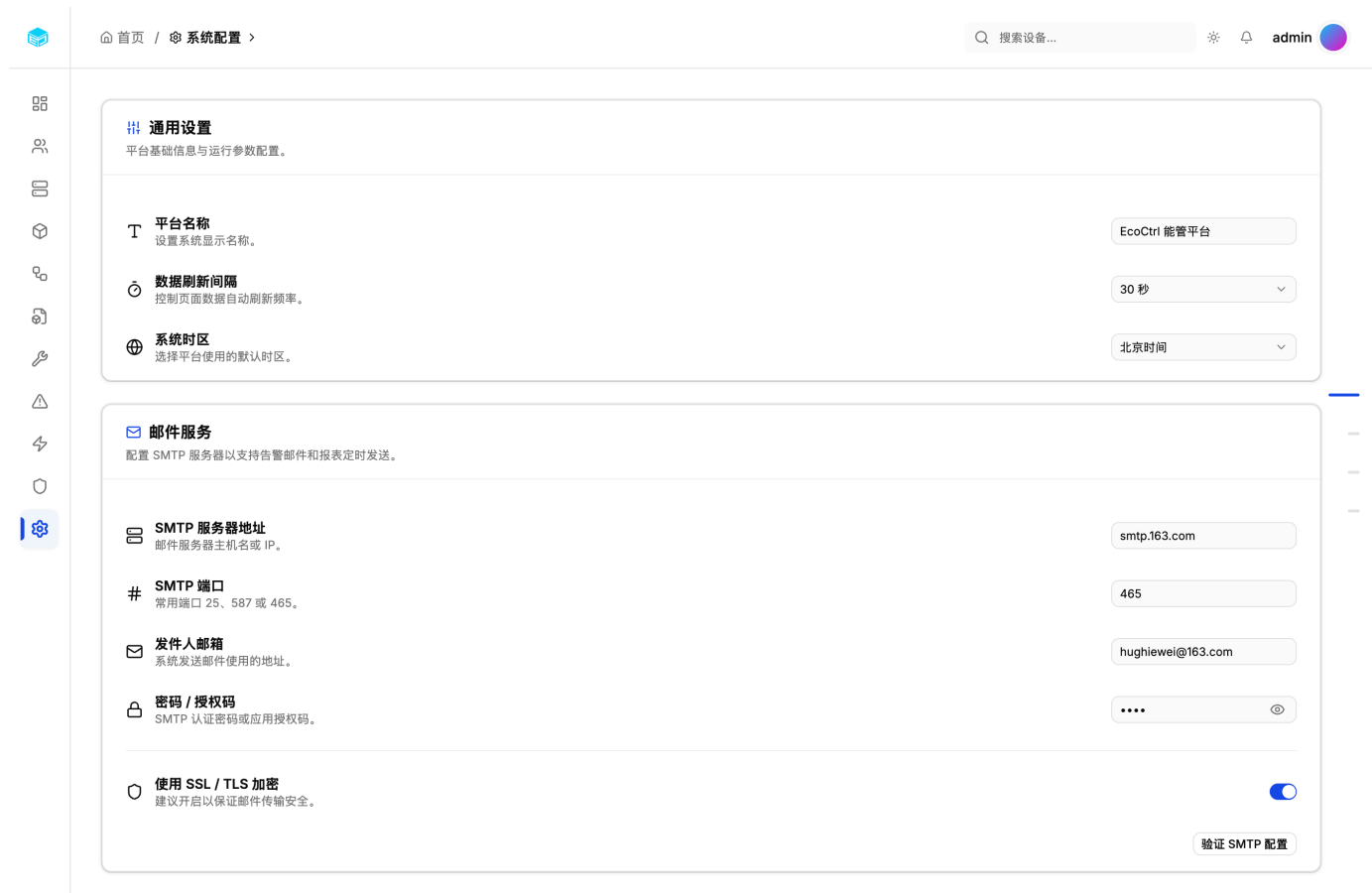
点击删除按钮：

- 1. 弹出确认对话框「确定要删除用户 [用户名] 吗？此操作不可撤销。」
- 2. 确认后执行删除
- 3. 关联的 Refresh Tokens 和 OAuth 绑定同时清除

5. 角色说明

角色	权限范围
admin	全部功能，包括用户管理、系统配置
manager	运维管理功能，不包含用户管理和系统配置
operator	日常操作功能，故障处理、报表查看
viewer	只读查看仪表盘和基础数据

个人信息与安全



系统配置下的「个人信息」页面，每个用户可在此管理自己的账户。

1. 头像

- 默认显示用户名首字母圆形头像
- 点击头像上的📷相机按钮，选择本地图片上传
- 上传后头像立即更新，保存到服务器（`files` 表）
- 支持格式：PNG / JPG / WebP

2. 修改用户名与邮箱

- 用户名：输入新用户名，点击保存
- 邮箱：输入新邮箱，系统发送验证邮件 → 输入验证码确认

3. 修改密码

需要输入：

1. 原密码：验证身份
2. 新密码：至少 8 位
3. 确认新密码：防止输入错误

提交后密码通过 bcrypt 哈希存储。修改成功后现有 Token 继续有效直至过期。

4. 账户信息（只读）

页面底部展示用户的只读信息：

- 用户 ID：系统唯一标识
 - 角色：当前角色（admin / manager / operator / viewer）
 - 状态：正常 / 已禁用
 - 最后登录时间：上次成功登录时间
-

5. OAuth 邮箱绑定

对于通过 OAuth（微信 / 飞书）注册但没有邮箱的用户，页面提供邮箱绑定入口：

1. 输入邮箱地址
2. 接收验证码
3. 输入验证码并设置密码（可选）
4. 绑定成功后同时支持 OAuth 和邮箱密码两种方式登录

偏好设置



1. 外观

设置项	选项	说明
主题	浅色 / 深色 / 跟随系统	基于 <code>useColorMode</code> 实现，选择跟随系统时自动匹配 OS 设置
界面密度	紧凑 / 舒适 / 宽松	控制表格行高、间距和元素密度
字体大小	小 / 中 / 大	整体界面字号缩放

2. 语言

- 简体中文：默认语言
- English：开发中（Web 门户已完成，Admin 后台部分文案仍为中文）

3. 通知

设置项	说明
桌面通知	开启后浏览器推送系统通知
告警提示音	新告警时播放提示音
邮件通知	通过 SMTP 发送通知邮件

4. 界面行为

设置项	说明
默认折叠侧边栏	登录后侧边栏默认折叠为图标模式
面包屑导航	开启后页面顶部显示面包屑路径

5. 保存机制

- 所有偏好设置更改 500ms 防抖自动保存
- 无需手动点击保存按钮
- 设置存储在两个位置：
 - `localStorage`：立即生效
 - `userSettings` API 端点：持久化到服务器

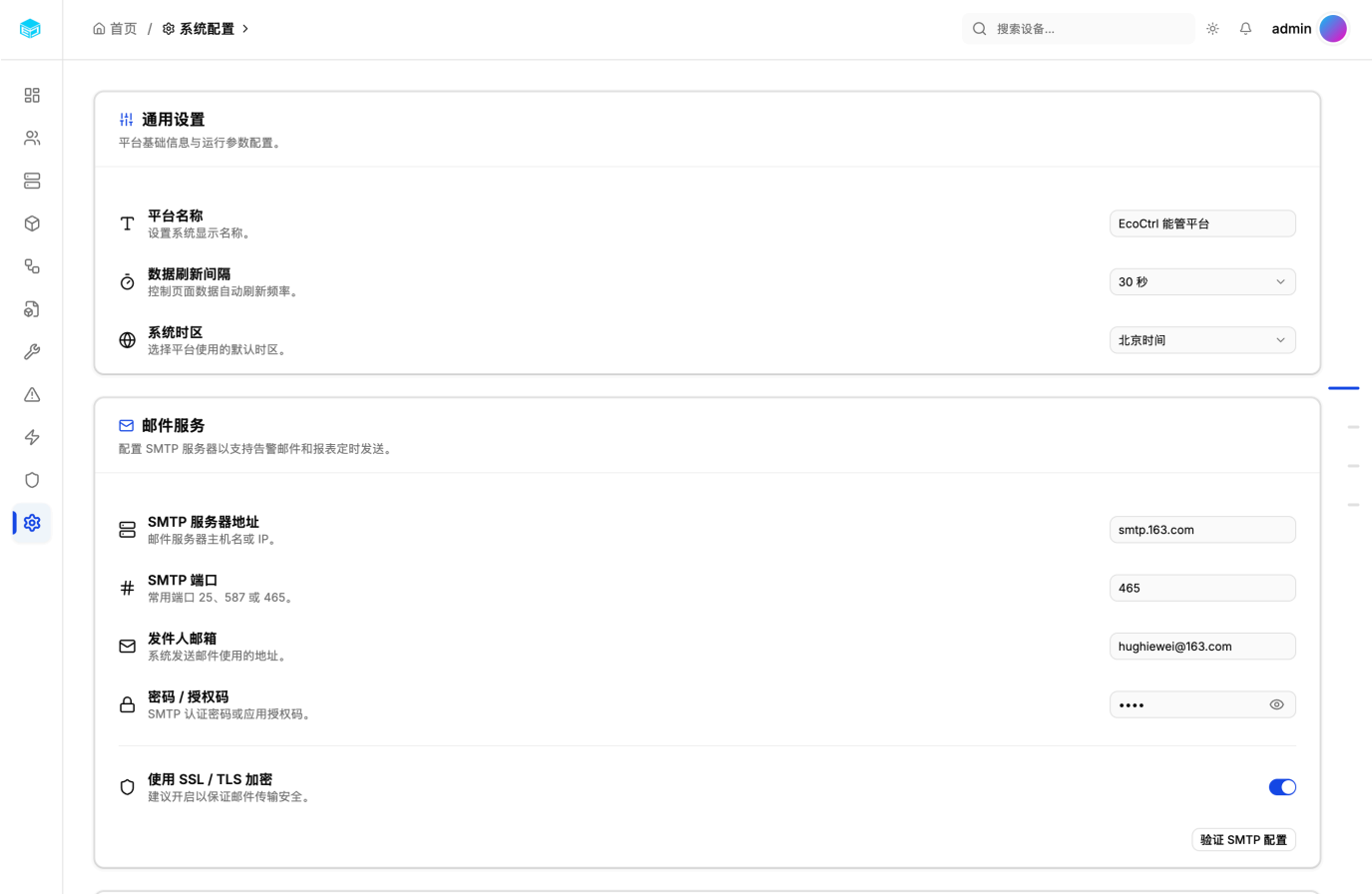
6. 恢复默认

点击「恢复默认」按钮：

1. 弹出确认对话框
2. 确认后所有偏好设置重置为系统默认值

3. 页面自动刷新

系统全局配置



系统配置页面（Config）提供平台级别的全局设置。

1. 通用设置

设置项	默认值	说明
平台名称	EcoCtrl	显示在 Web 门户导航栏和 Admin 后台浏览器的标签标题
数据刷新闻隔	30s	可选：10s / 30s / 1m / 5m = 管理总览的自动刷新周期
系统时区	Asia/Shanghai	影响日志时间戳和定时任务的执行时间

2. 邮件服务

配置 SMTP 用于发送验证码和通知邮件。

设置项	必填	说明
SMTP 主机	是	邮件服务器地址
SMTP 端口	是	一般为 465（SSL）或 587（TLS）
发件人地址	是	发送邮件的邮箱地址
密码/授权码	是	邮箱密码或 SMTP 授权码（支持切换可见性  ）
SSL/TLS	否	加密方式开关

运维参考：SMTP 详细配置见 [SMTP 邮件服务配置](#)，完整环境变量见 [环境变量参考](#)

2.1. 验证配置

配置完成后，点击「验证 SMTP 配置」：

- 1. 系统使用当前配置发送测试邮件到管理员邮箱
- 2. 成功收到测试邮件 → 配置可用
- 3. 发送失败 → 显示错误信息

注意：首次部署时，系统会从 `packages/server/.env.local` 的 `SMTP_*` 环境变量自动同步配置。如需修改请通过 Admin 后台 UI 保存，保存后值写入 `platform_configs` 表，不再依赖环境变量。

3. 数据与安全

设置项	说明
自动备份	开关：定时自动备份数据库
备份保留天数	自动备份文件的保留天数
会话超时时间	用户无操作后自动登出的时间（分钟）

3.1. 自动备份机制

- 基于 `backupSchedule` 配置和 pg-boss 定时任务
- 备份文件存储在服务端磁盘或文件存储中
- 超过保留天数的备份自动清理

第三方系统集成



高级管理（Advanced Management）页面提供第三方系统集成功能。

1. WebTalk 集成

WebTalk 是一个第三方 IoT 设备管理平台，EcoCtrl 通过反向代理将其管理界面嵌入到 Admin 后台中。

#> 技术细节：反向代理架构见 [WebTalk 反向代理](#)

2. 配置步骤

1. 在第三方系统集成页面输入 WebTalk 系统的用户名和密码
2. 点击「提交认证」

3. 认证成功后，页面下方以 iframe 方式内嵌 WebTalk 管理界面

2.1. 技术原理

- 认证过程由 EcoCtrl 服务端代为完成（/api/webtalk/* 路由）
 - 后续所有 WebTalk 请求通过 EcoCtrl 服务端代理转发
 - iframe 保持同源，解决了跨域 Cookie 问题
-

3. 配置重置

- 重新输入凭据可更新认证信息
- 退出登录时清除缓存的认证凭据

AI 智能助手



Web 门户的屏幕宠物（Screen Pet）是一个可爱又实用的 AI 助手。

技术细节：AI 提供者架构与工具调用机制见 [AI 与智能助手](#)

1. Screen Pet

1.1. 显示与交互

- 一个浮动的小宠物显示在 Web 门户页面右下角
- 可拖拽：拖拽宠物头像可在屏幕上任意移动位置（位置通过 `localStorage` 持久化）
- 主题切换：可从服务器获取宠物列表，切换不同外观主题
 - 宠物以 sprite sheet（精灵图）zip 包形式上传
 - 由 `pets-api.ts` 从 `/api/pets` 端点获取

1.2. AI 对话

点击宠物弹出聊天界面：

- 文字输入：在聊天输入框中输入消息
- 语音输入：支持语音输入（通过 `useVoiceInput` hook）
- 语音输出：AI 回复支持 TTS 朗读（通过 `useVoiceOutput` hook）
- 流式响应：AI 回复通过 SSE 流式输出，逐 token 显示
- 对话历史：同一 Session 内的对话保持上下文

1.3. AI 能力

- 基于 Anthropic Claude 或 OpenAI 提供 AI 能力（由服务端 `AI_PROVIDER` 环境变量决定）
- 支持工具调用：AI 可以查询设备状态、能耗数据等
- 回复内容结合当前页面上下文（`context: { currentPage, currentRouteData }`）

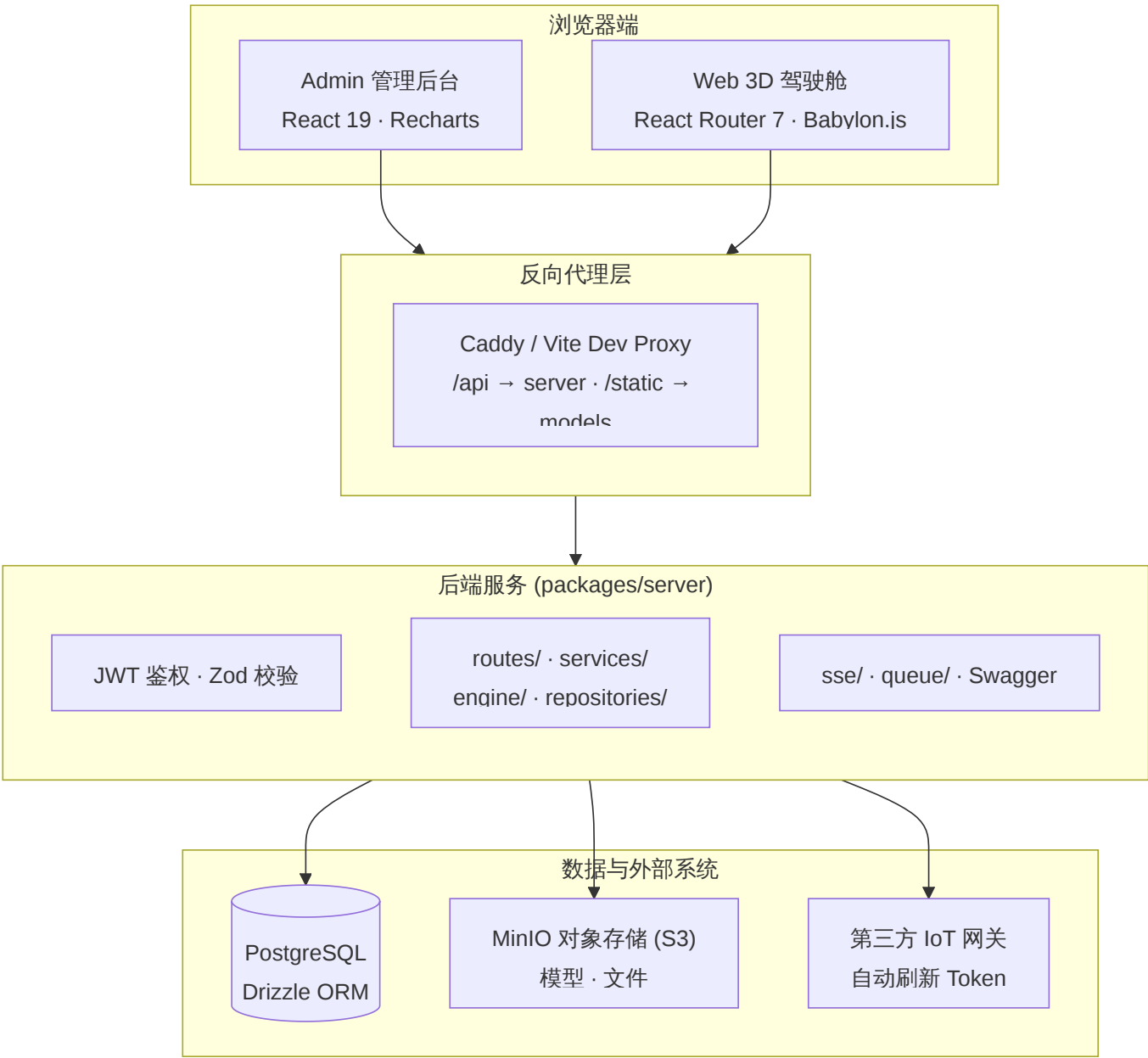
2. 偏好设置

在 Web 设置页的「通用」分区可管理 AI 助手偏好：

设置	说明
宠物主题	切换宠物的外观
启用语音	开关语音输入/输出
语速	0.5x - 2.0x 语速调节

整体架构概览

1. 高层架构图



2. Monorepo 结构

EcoCtrl 采用 pnpm workspace monorepo，根目录下包含 `apps/` 和 `packages/`：


```
ecoctrl/
├─ apps/
│   ├─ admin/          # React 19 SPA 管理后台
│   ├─ web/            # React Router 7 + BabylonJS 3D 公共门户
│   └─ docs/           # VitePress 2 文档站点
├─ packages/
│   ├─ server/         # Fastify 5 后端 API 服务
│   ├─ ui/             # shadcn/ui 组件库（源码分发）
│   └─ shared/         # 共享 Zod Schema、Vite 配置、工具函数
├─ docker/            # Docker Compose 部署配置
└─ ...
```

3. 技术栈

层级	技术	说明
前端 Admin	React 19 + TypeScript + TailwindCSS v4	内部管理界面
前端 Web	React Router 7 + BabylonJS 9 + TypeScript	公共 3D 门户
后端	Fastify 5 + TypeScript	RESTful API 服务
数据库	PostgreSQL 16 + Drizzle ORM	关系型数据库
实时通信	Server-Sent Events (SSE)	服务端推送到客户端
任务队列	pg-boss	PostgreSQL 驱动的后台任务
AI	Anthropic Claude / OpenAI	AI 对话与工具调用
IoT	BACnet 网关代理	第三方物联网设备接入
缓存	无独立缓存层（直接查询 PostgreSQL）	—

4. 请求生命周期

一次典型的 API 请求经过以下路径：

1. 浏览器发送请求到 `/api/xxx`
2. Caddy / Vite Dev Proxy 根据前缀 `/api` 改写目标地址
3. Fastify 接收请求
 - ├─ onRequest 钩子: 校验 JWT (public routes 白名单放行)
 - ├─ Zod 校验: body / querystring / params
 - ├─ 路由处理器:
 - │ ── 调用 repository 层 (数据库操作)
 - │ ── 或调用 service 层 (IoT 代理、邮件发送等)
 - │ ── 或调用 engine 层 (工作流执行)
 - └─ 响应返回
4. Caddy 将响应转发回浏览器

5. 部署模式

5.1. 本地开发

```
node tsx --watch    → Fastify (localhost:3000)
vite-plus dev       → Admin (localhost:5173)
vite-plus dev       → Web   (localhost:8080)
vitepress dev       → Docs  (localhost:5174)
```

Vite Dev Proxy 自动将 `/api` 和 `/static` 前缀转发到后端。

5.2. Docker Compose

```
postgres:16-alpine      :5432
ecoctrl-server (Node)    :3000
ecoctrl-admin (Caddy)    :4173 → /api /static 重写至 http://server:3000
ecoctrl-web   (Caddy)    :8081 → /api /static 重写至 http://server:3000
```

前端 bundle 始终请求字面量 `/api` 和 `/static` 前缀。代理层负责改写目标地址。修改后端主机或前缀属于运行时配置变更，无需重新构建。

6. 构建流水线

包	工具	输出
apps/web 、 apps/admin	vp build (Rolldown)	静态 SPA bundle
packages/server	rolldown	dist/index.mjs + 自动生成 dist/package.json
apps/docs	vitepress build	.vitepress/dist/ 静态站点
packages/ui 、 packages/shared	源码分发	不适用

Admin 前端架构

1. 概述

Admin 后台是一个单页应用（SPA），基于 React 19 + TypeScript + TailwindCSS v4。与多数后台应用不同，Admin 未使用 React Router，而是通过 Zustand store 中的 `activeTab` 状态实现 Tab 式路由。这种设计简化了渲染逻辑，无需 URL 路由匹配即可在多个页面间切换。

2. Tab 路由系统

2.1. 核心机制

路由状态存储在 `apps/admin/src/store/appStore.ts` 中的 `useAppStore`：

```
interface AppState {  
  activeTab: string; // 当前活动 Tab，决定渲染哪个页面组件  
  setActiveTab: (tab: string) => void;  
  
  // 子 Tab（页面内部导航），每个页面可维护独立的子 Tab 状态  
  energyTab: string; // "overview" | "charts" | "areas"  
  faultsTab: string; // "list" | "stats"  
  dashboardModelTab: string; // "hotspots" | "labels" | "config"  
  modelsTab: string; // "models" | "upload"  
  workflowsTab: string; // "workflows" | "templates"  
  // 其他子 Tab...  
}
```

typescript

Tab 状态通过 `zustand/middleware` 的 `persist` 插件持久化到 `localStorage`（storage key 为 `ecoctrl-admin-storage`）。页面刷新后恢复上次的 Tab 和子 Tab 状态。

2.2. 16 个页面组件

App.tsx 通过 `switch(activeTab)` 渲染对应页面组件：

Tab Key	组件	说明
<code>overview</code>	Overview	仪表盘总览，显示 KPI 卡片和趋势图表
<code>config</code>	Config	平台全局配置
<code>accounts</code>	Accounts	用户账户管理（CRUD + 角色分配）
<code>models</code>	Models	3D 模型文件管理（上传、列表、删除）
<code>settingsGroup</code>	DashboardModel	全屏沉浸 — 3D 场景编辑器，配置热点、标签、相机的可视化工具
<code>workflows</code>	Workflows	工作流列表与模板管理
<code>workflowCanvas</code>	WorkflowCanvasPage	全屏沉浸 — 工作流可视编辑画布，支持节点拖拽与连线
<code>logViewer</code>	LogViewerPage	全屏沉浸 — 工作流执行日志查看器，支持按时间/级别过滤
<code>reports</code>	Reports	报表生成与管理
<code>maintenance</code>	Maintenance	维护任务管理与调度
<code>faults</code>	Faults	故障记录与统计
<code>energy</code>	Energy	能耗数据分析
<code>advancedManagement</code>	AdvancedManagement	高级管理功能
<code>profile</code>	Profile	当前用户个人信息编辑
<code>preferences</code>	Preferences	偏好设置（主题、密度、通知、布局等）
<code>Login</code> （独立）	Login	登录页面，仅未认证时展示；不包含主布局组件

2.3. 子 Tab 嵌套模式（tabStoreKey pattern）

多个页面组件内部维护子 Tab 导航。子 Tab 状态同样存储在 `useAppStore` 中，每个页面有自己的子 Tab Key：

typescript

```
// appStore.ts 中的子 Tab 字段定义
energyTab: "overview"; // Energy 页面: 总览 / 图表 / 区域
faultsTab: "list"; // Faults 页面: 列表 / 统计
dashboardModelTab: "hotspots"; // DashboardModel: 热点 / 标签 / 配置
modelsTab: "models"; // Models: 模型列表 / 上传
workflowsTab: "workflows"; // Workflows: 工作流 / 模板
```

页面组件内通过以下方式读写子 Tab:

typescript

```
function EnergyPage() {
  const subTab = useAppStore((s) => s.energyTab);
  const setSubTab = useAppStore((s) => s.setEnergyTab);

  switch (subTab) {
    case "overview": return <EnergyOverview />;
    case "charts":   return <EnergyCharts />;
    case "areas":    return <EnergyAreas />;
  }
}
```

这种模式避免了 URL 参数管理, 子 Tab 状态天然持久化且全局可访问。

3. 认证守卫

认证逻辑实现在 `App.tsx` 顶层:

text

应用启动流程:

1. 检查 localStorage 中的 Access Token
2. 有 Token:
 - a. 调用 GET /api/auth/me 验证有效性
 - b. 有效 → 加载用户偏好 → 渲染主界面 (侧边栏 + 内容区)
 - c. 无效或 401 → 清空 Token → 渲染 Login 页面
3. 无 Token → 直接渲染 Login 页面

3.1. 登录流程

text

Login 页面：

1. 用户输入用户名 + 密码 → 提交
2. POST /api/auth/login → 返回 { accessToken, refreshToken, user }
3. onLogin 回调：
 - a. 存储 Token 到 localStorage
 - b. 设置 useAppStore 中的用户状态
 - c. 调用 GET /api/auth/me/state 加载用户偏好
 - d. activeTab 切换到上次退出时的 Tab (从 localStorage 恢复)
 - e. 渲染主布局

3.2. 登出流程

text

1. 用户点击登出（侧边栏底部或 Header 中的退出按钮）
2. ClearAuth: 清除 Token、用户数据、子 Tab 状态
3. activeTab 置为 "Login"
4. 可选：调用 POST /api/auth/logout 通知服务端作废 Refresh Token

4. 布局组件体系

4.1. LayoutShell

[App.tsx](#) 中定义的主布局有两种模式，根据当前 activeTab 动态切换。

标准布局（适用于 13 个非沉浸式页面）：

tsx

```
<div className="bg-background text-foreground flex h-screen">
  <SidebarNav activeTab={activeTab} onTabChange={setActiveTab} />
  <div className="flex h-full min-w-0 flex-1 flex-col">
    <Header activeTab={activeTab} showBreadcrumb={effectivePrefs.showBreadcrumb} />
    <main className="flex-1 overflow-hidden">
      <ScrollArea className="h-full">{pageContent}</ScrollArea>
    </main>
  </div>
</div>
```

沉浸式布局（适用于 `settingsGroup` / `workflowCanvas` / `logViewer` 三个页面）：

```
<div className="h-screen w-screen overflow-hidden">{fullScreenPageContent}</div>
```

tsx

沉浸式布局隐藏侧边栏和顶栏，将全屏空间留给 3D 场景编辑器、可视工作流画布、日志面板等需要大量操作空间的页面。

4.2. SidebarNav

侧边栏组件渲染导航项列表。每项包含图标（`lucide-react`）和文本标签。点击触发 `setActiveTab(tabKey)`。侧边栏宽度固定，可折叠（收窄为仅图标模式），折叠状态存储在 `user preferences` 中。

4.3. Header

顶栏组件包含：

- 标题面包屑：显示当前 Tab 名称和子 Tab 路径层级，由 `showBreadcrumb` 偏好控制显隐
- 全局操作区：通知铃铛、搜索快捷键入口、用户头像下拉菜单（个人信息 / 偏好设置 / 登出）
- 时间显示：当前日期和时间（国际化格式）

5. 三个全屏沉浸式页面

5.1. DashboardModel (settingsGroup)

3D 场景配置工具。界面分为左侧场景树面板、中央 3D 渲染区域、右侧属性面板。支持以下编辑功能：

- 热点标记放置（`placeLabel` 模式）
- 标签文字编辑
- 场景配置（环境光、相机预设、背景色）
- 模型文件选择与可见性切换
- 剪切预览模式（`clipPreview`）— 用于查看模型剖面

5.2. WorkflowCanvas (workflowCanvas)

工作流可视编辑画布。基于 React Flow 库，支持：

- 节点拖拽添加与删除
- 节点间连线（handles 输入/输出端口）
- 工作流参数配置（侧边面板）
- 保存 / 发布工作流定义

5.3. LogViewer (logViewer)

工作流执行日志查看器。功能包括：

- 按时间范围过滤日志
- 按日志级别（info / warn / error）过滤
- 日志全文搜索
- 自动滚动（新日志到达时）
- 日志导出（复制 / 下载）

6. 用户偏好系统

偏好设置分三层合并覆盖：

typescript

```
const effectivePrefs = {
  ...DEFAULT_PREFERENCES, // 内置默认值（硬编码）
  ...userPrefs, // 数据库持久化偏好（GET /api/auth/me/state 获取）
  ...preferencesOverride, // 本地会话覆盖（未保存到数据库的临时变更）
};
```

6.1. 配置项清单

配置项	类型	默认值	说明
theme	"system" / "light" / "dark"	"system"	主题模式

配置项	类型	默认值	说明
language	"zh-CN" / "en-US"	"zh-CN"	界面语言
density	"comfortable" / "compact"	"comfortable"	布局密度
fontSize	"normal" / "large"	"normal"	字体大小
desktopNotification	boolean	false	桌面通知开关
alertSound	boolean	false	告警声音开关
showBreadcrumb	boolean	true	面包屑显隐
sidebarCollapsed	boolean	false	侧边栏折叠状态

7. SSE 集成

Admin 前端通过 `SseProvider` 组件集成 Server-Sent Events，实现实时数据推送。

7.1. SseProvider

`SseProvider` 在 `App.tsx` 中作为顶层组件包裹主界面，管理 SSE 连接的完整生命周期：

- 连接建立：认证成功后自动发起 `/api/events` 连接
- 心跳维持：后端每隔 30 秒发送一次 `ping` 事件，超过 90 秒未收到心跳触发重连
- 自动重连：指数退避策略（1s / 2s / 4s / 8s / 16s，上限 30s）
- 事件分发：按事件类型分发到对应 handler，当前处理的事件类型包括：

事件类型	说明	更新目标
lighting	照明设备状态变更	照明显示面板
alert	新告警事件	告警通知列表 + 桌面通知
maintenance	维护任务状态变更	维护看板
energy	能耗读数更新	能耗图表自动刷新

事件类型	说明	更新目标
fault	故障记录变更	故障列表

7.2. sseStore

sseStore 是 SSE 连接状态管理的 Zustand store，提供连接生命周期的钩子方法和事件分发注册接口：

typescript

```
interface SSEState {
  status: "disconnected" | "connecting" | "connected";
  reconnectAttempts: number;
  connect: () => void;
  disconnect: () => void;
  onMessage: (event: string, handler: (data: unknown) => void) => void;
}
```

8. 其他 Store

8.1. modelEditorStore

3D 场景编辑器的状态管理，位于 `apps/admin/src/store/modelEditorStore.ts`：

typescript

```
interface ModelEditorState {
  config: DashboardModelConfig | null; // 当前编辑的场景配置
  labels: DashboardModelLabel[]; // 标签列表
  mode: "select" | "placeLabel" | "clipPreview"; // 编辑模式
  selectedLabelId: string | null; // 当前选中的标签
  modelFileVisible: boolean; // 模型文件可见性
  // 操作方法
  loadConfig: (id: string) => Promise<void>;
  addLabel: (label: DashboardModelLabel) => void;
  updateLabel: (id: string, changes: Partial<DashboardModelLabel>) => void;
  deleteLabel: (id: string) => void;
}
```

```
setMode: (mode: ModelEditorState["mode"]) => void;  
}
```

8.2. useStore

全局通用状态 store，位于 [apps/admin/src/store/](#)，管理 FOUC（闪白）状态和应用级标志：

```
interface AppFlags {  
  themeLoaded: boolean; // 主题样式已加载  
  userLoaded: boolean; // 用户数据已加载  
}
```

typescript

Web 前端架构

1. 概述

Web 前端是面向公众的 3D 门户，基于 React 19 + React Router 7（framework 模式）+ TypeScript + TailwindCSS v4。核心能力是 BabylonJS 3D 场景渲染和可拖拽的 Bento Grid 仪表盘。前端仅做 CSR（客户端渲染），不涉及 SSR，构建产物为静态 SPA bundle。

2. 路由结构

路由定义在 `apps/web/app/routes.ts`，共 3 个入口：

```
export default [
  layout("routes/dashboard-layout.tsx", [
    index("routes/home.tsx"), // / → 仪表盘主页
    route("settings", "routes/settings.tsx"), // /settings → 设置页
  ]),
  route(".well-known/*", "routes/well-known.tsx"), // 开发工具探测
] satisfies RouteConfig;
```

typescript

- /（首页）：Bento Grid 仪表盘 + BabylonJS 3D 场景视图。用户登录后可编辑仪表盘布局。
- /settings（设置页）：用户偏好设置，包含主题切换、语言选择、密码修改等。
- /.well-known/（开发工具探测）：浏览器扩展或开发工具通过 Well-Known URI 机制探测应用信息。

使用 React Router 7 的 `layout route` 模式，`home` 和 `settings` 共享 `dashboard-layout.tsx` 布局组件。

3. BabylonJS 3D 场景集成

核心 3D 场景组件位于 `apps/web/app/components/dashboard/building-view.tsx`，加载流程如下：

1. 组件挂载后调用 `/api/public/model` 获取场景配置（模型 URL、相机预设、热点列表、标签注释^{text}）
2. 创建 Babylon.js Engine 实例（Canvas 尺寸自适应容器大小）
3. 创建 Scene 实例，配置环境贴图、光照和雾效
4. 通过 `SceneLoader.ImportMeshAsync` 加载 glTF / glB 模型
5. 根据配置应用相机预设（预设位置 + 目标点）
6. 渲染热点标记（HTML overlay 定位到 3D 坐标）和标签注释
7. 监听用户交互：点击热点触发面板展开、鼠标拖拽旋转场景、滚轮缩放
8. 窗口 `resize` 时自动调整 Engine 尺寸
9. 组件卸载时调用 `engine.dispose()` 清理资源

支持的模型格式为 glTF 2.0（.gltf + .bin 分离文件）和 glB 二进制格式。模型文件通过 `/api/uploads` 上传管理。

3.1. 场景性能优化

- 模型加载期间显示骨架屏占位
- 使用 `engine.setHardwareScalingLevel()` 控制渲染分辨率平衡性能
- 非可见标签延迟渲染（IntersectionObserver）
- Camera 惯性控制在低帧率时自动降级

4. Bento Grid 仪表盘布局系统

主页使用 12 列 CSS Grid 排版组件网格。每个组件的网格坐标（`layoutX` / `layoutY` / `layoutW` / `layoutH`）存储在数据库 `dashboard_widgets` 表中。

4.1. 组件类型与数据源

组件类型	数据源	说明
统计卡片	<code>dashboard_stats</code> / IoT 点位	带趋势指示器的大型数字 KPI
图表	<code>energy_readings</code> 、 <code>fault_stats</code>	折线图、柱状图、饼图

组件类型	数据源	说明
列表	alerts 、 maintenance_reminders	可滚动告警 / 任务列表
天气	OpenWeatherMap API	当前天气卡片（API Key 缺失时自动隐藏）
能耗图表	energy_readings + energy_areas	多系列能耗分析图

4.2. 布局数据结构

typescript

```
// dashboard_widgets 表字段（简化）
interface DashboardWidget {
  id: string;
  type: "stats" | "chart" | "list" | "weather" | "energy";
  layoutX: number; // 网格 X 坐标 (0-11)
  layoutY: number; // 网格 Y 坐标
  layoutW: number; // 宽度 (占几列, 1-12)
  layoutH: number; // 高度 (占几行)
  config: Record<string, unknown>; // 组件专属配置
}
```

布局数据通过 `/api/dashboard` 端点查询， `/api/dashboard/settings` 端点保存编辑结果。

5. Edit Mode 状态管理

仪表盘支持编辑模式，用户可自定义组件排布。状态由 `widgetData` store 中的 `isEditing` 标志控制。

text

- Edit Mode 生命周期：
- 1. 用户点击「编辑」按钮 → `isEditing = true`
 - 2. 组件进入可拖拽状态（拖拽手柄、尺寸控制点可见）
 - 3. 用户操作：
 - a. 拖拽组件到新位置 → 更新 `layoutX` / `layoutY`
 - b. 拖动调整组件尺寸 → 更新 `layoutW` / `layoutH`
 - c. 从类型面板拖入新组件 → `POST /api/dashboard`
 - d. 点击组件删除按钮 → `DELETE /api/dashboard/{id}`
 - 4. 用户点击「保存」→ `PUT /api/dashboard/settings`

5. 保存成功 → `isEditing = false`, `widgetData` 刷新
6. 用户点击「取消」→ `isEditing = false`, 恢复原始布局

暂存状态仅在内存中维护，未保存前刷新页面将丢失编辑内容。

6.5 个 Zustand Store

6.1. auth — 认证状态

typescript

```
interface AuthState {
  accessToken: string | null;
  refreshToken: string | null;
  user: User | null;
  setAuth: (access: string, refresh: string) => void;
  clearAuth: () => void;
}
```

Token 存储于 `localStorage` (key: `ecoctrl-auth`)。所有 API 请求通过 `axios interceptor` 自动附加 `Authorization: Bearer` 头，遇到 401 时自动尝试 Refresh Token 续期。

6.2. settings — 用户偏好

typescript

```
interface SettingsState {
  theme: "system" | "light" | "dark";
  language: "zh-CN" | "en-US";
  dashboardLayout: DashboardWidget[];
  setTheme: (theme: SettingsState["theme"]) => void;
  setLanguage: (lang: SettingsState["language"]) => void;
}
```

6.3. lighting — 照明控制 (SSE 联动)

缓存照明控制状态。与 SSE 的事件推送联动，实时更新照明组状态 (`off` / `half` / `on`)。

typescript

```
interface LightingState {
  groups: Record<string, LightingGroup>;
  setGroupState: (id: string, state: LightingState) => void;
  syncFromSSE: (event: LightingSSEEvent) => void;
}
```

当 SSE 推送照明状态变更事件时，`syncFromSSE` 直接更新对应组的显示状态，无需重新请求 API。

6.4. pet — 屏幕宠物

管理 AI 屏幕宠物（Screen Pet）的状态：

typescript

```
interface PetState {
  theme: PetTheme; // 宠物外观主题
  voice: boolean; // 语音反馈开关
  position: { x: number; y: number }; // 屏幕上的像素坐标
  animation: string; // 当前动画状态 (idle / speak / walk / sleep)
  setTheme: (theme: PetTheme) => void;
  setPosition: (pos: { x: number; y: number }) => void;
}
```

宠物位置持久化到 localStorage，跨会话保持。语音反馈依赖浏览器 Web Speech API。

6.5. widgetData — 仪表盘数据缓存

typescript

```
interface WidgetDataState {
  data: Record<string, unknown>; // 各组件的数据缓存
  isLoading: Record<string, boolean>; // 加载状态
  isEditing: boolean; // 是否处于编辑模式
  fetchWidgetData: () => Promise<void>;
  setEditing: (editing: boolean) => void;
}
```

数据定时轮询（间隔 30 秒），每次轮询仅请求当前仪表盘上可见组件所需的数据端点。

7. i18n 国际化系统

Web 前端使用自研 i18n 系统（非 react-i18next 或其他第三方库），通过 Zustand store 管理语言状态。

7.1. 文件结构

```
apps/web/app/locales/  
├─ index.ts      # 语言注册与切换逻辑  
├─ en-US.ts      # 英文翻译键值对（扁平对象）  
└─ zh-CN.ts      # 中文翻译键值对
```

7.2. 实现方式

```
// locales/en-US.ts typescript  
export const en = {  
  "dashboard.title": "Dashboard",  
  "dashboard.edit": "Edit Layout",  
  "settings.title": "Settings",  
  "settings.theme": "Theme",  
  // ...  
};
```

Zustand store 中存储当前语言代码，组件通过 `useTranslation()` hook 获取翻译文本：

```
function useTranslation() { typescript  
  const lang = useSettingsStore((s) => s.language);  
  const translations = lang === "zh-CN" ? zh : en;  
  return (key: string) => translations[key] ?? key;  
}
```

语言切换即时生效，无需页面刷新。新语言通过扩展 `locales/` 目录下的文件并注册到 `index.ts` 即可添加。

8. SSE 客户端

Web 前端的长连接数据推送通过 SSE（Server-Sent Events）实现。SSE 客户端封装在自定义 hook 或服务模块中，连接管理包括：

- 自动重连：连接断开后以指数退避策略（1s / 2s / 4s / 8s / 16s，上限 30s）尝试重连
- 事件分发：按事件类型（`lighting` / `alert` / `maintenance` / `energy`）分发到对应 store 或 handler
- 生命周期：页面可见时连接，页面隐藏时断开（`visibilitychange` 事件控制）

SSE 端点路径为 `/api/sse`，客户端在认证成功后发起连接。

9. Screen Pet 组件架构

屏幕宠物（Screen Pet）是 Web 前端的特色功能，一个浮动在页面上的 AI 角色组件。

9.1. 组件结构

ScreenPet		
├─	PetRenderer	# 宠物渲染（CSS sprite / Lottie 动画）
├─	SpeechBubble	# 对话气泡（AI 回复的文本展示）
├─	DragHandle	# 拖拽控制（允许用户在屏幕任意位置拖动宠物）
├─	VoiceController	# 语音输入/输出（Web Speech API 封装）
└─	ActionMenu	# 交互菜单（主题切换、语音开关、隐藏）

9.2. 交互流程

1. 用户点击宠物 → 触发问候动画 + 随机对话气泡
2. 用户长按并通过 `DragHandle` 拖拽 → 更新 `PetState.position`
3. 用户录入语音（`VoiceController` 激活）→ 语音转文字 → 调用 AI API → 响应显示在 `SpeechBubble`
4. 宠物根据场景自动切换动画状态：
 - `idle`（空闲）→ 轻微浮动
 - `speak`（说话）→ 对话气泡出现时

text

- walk (移动) → 被拖拽时
- sleep (睡眠) → 页面闲置超过 5 分钟

9.3. 数据流

用户语音 → Web Speech API (SpeechRecognition) text
→ AI API (Claude / OpenAI 文本生成)
→ SpeechBubble 显示文本
→ 可选: Web Speech API (SpeechSynthesis) 语音播报

10. 构建与部署

- 构建工具: `vp build` (vite-plus 封装 Rolldown)
- 输出目录: `apps/web/dist/`
- Docker 镜像: SPA bundle + Caddyfile (反向代理配置)
- 服务端口: 8081 (Docker Compose 默认)
- 部署模式: 纯静态 CSR, 所有动态数据通过 API 获取

共享包体系

1. 概述

EcoCtrl Monorepo 包含两个共享 package：@ecoctrl/ui（组件库）和 @ecoctrl/shared（工具库）。两者均采用源码分发模式 — 不经过独立构建步骤，在消费者 App 的编译流水线中直接编译。这种设计避免了双重编译问题，减少了构建步骤的开销。

2. @ecoctrl/ui — 组件库

2.1. 定位

@ecoctrl/ui 是基于 shadcn/ui 的组件库，提供 Admin 和 Web 前端共享的 UI 组件。组件采用 TailwindCSS v4 样式，遵循 shadcn 的设计体系（基于 Radix UI 原语）。

2.2. 源码分发模式

消费者 App（admin、web）通过 TypeScript 路径映射直接引用 .tsx 源码，在各自的 vp build（Rolldown）过程中一并编译。 packages/ui/ 目录本身不运行任何构建命令。

分发路径：



2.3. Subpath Export 与 generate-proxies

组件通过 `package.json` 的 `exports` 字段定义子路径导出：

json

```
{
  "exports": {
    ".": "./src/index.ts",
    "./button": "./src/components/ui/button.tsx",
    "./card": "./src/components/ui/card.tsx",
    "./scroll-area": "./src/components/ui/scroll-area.tsx"
    // 每个 UI 组件自动注册一条子路径导出
  }
}
```

新增组件后需要运行 `pnpm generate-proxies`。该脚本扫描 `src/components/ui/` 下的所有 `.tsx` 文件，自动更新 `package.json` 的 `exports` 字段和 TypeScript 类型声明。

2.4. 组件清单

`@ecoctrl/ui` 包含标准 shadcn 组件（持续扩展中）：

组件	文件路径	依赖
Button	<code>src/components/ui/button.tsx</code>	Radix Slot
Card	<code>src/components/ui/card.tsx</code>	—
ScrollArea	<code>src/components/ui/scroll-area.tsx</code>	Radix ScrollArea
Input	<code>src/components/ui/input.tsx</code>	—
Select	<code>src/components/ui/select.tsx</code>	Radix Select
Table	<code>src/components/ui/table.tsx</code>	—
Dialog	<code>src/components/ui/dialog.tsx</code>	Radix Dialog
Dropdown	<code>src/components/ui/dropdown.tsx</code>	Radix Dropdown
Toast	<code>src/components/ui/toast.tsx</code>	Sonner

组件	文件路径	依赖
Sheet	src/components/ui/sheet.tsx	Radix Sheet
Tooltip	src/components/ui/tooltip.tsx	Radix Tooltip
Badge	src/components/ui/badge.tsx	—
Avatar	src/components/ui/avatar.tsx	Radix Avatar
Switch	src/components/ui/switch.tsx	Radix Switch
Tabs	src/components/ui/tabs.tsx	Radix Tabs
Skeleton	src/components/ui/skeleton.tsx	—

2.5. TailwindCSS 配置

[@ecoctrl/ui](#) 的样式依赖 TailwindCSS 预设。消费者 App 在其 `tailwind.config.ts` 中引用 [@ecoctrl/ui](#) 的样式源路径：

typescript

```
// consumer 的 tailwind.config.ts
export default {
  content: [
    './src/**/*.ts', './src/**/*.tsx',
    './packages/ui/src/**/*.ts', './packages/ui/src/**/*.tsx', // 包含 UI 包源码
  ],
  // ...
};
```

3. @ecoctrl/shared — 工具库

[@ecoctrl/shared](#) 提供三部分能力，供前后端共同使用。

3.1. 1. Zod Schema

跨前后端共享的数据结构定义，位于 [packages/shared/types/api/](#)：

- 用户相关: `UserPreferences`、`UserProfile`、`LoginRequest`、`LoginResponse`
- 3D 场景相关: `DashboardModelConfig` (场景配置)、`DashboardModelLabel` (标签)、`LabelAction` (标签行为)
- 能耗相关: `EnergyReading`、`EnergyArea`、`EnergyStats`
- 运维相关: `Alert`、`MaintenanceTask`、`FaultRecord`
- API 通用: 分页参数 `PaginationParams`、分页响应 `PaginatedResponse<T>`

这些 Zod Schema 在服务端通过 `fastify-type-provider-zod` 驱动路由校验, 前端则用于类型推导和数据验证:

```

// 服务端使用
import { LoginRequest } from "@ecoctrl/shared";

server.post(
  "/auth/login",
  {
    schema: { body: LoginRequest },
  },
  handler,
);

// 前端使用
import { LoginRequest } from "@ecoctrl/shared";
type LoginFormData = z.infer<typeof LoginRequest>;
  
```

typescript

同一份 Schema 同时驱动 OpenAPI 文档生成 (通过 `@fastify/swagger` 从 Zod Schema 自动提取), 确保文档与实现始终同步。

3.2.2. Vite 配置工具 (viteConfig)

`viteConfig` 是统一的 Vite 配置基座, 所有 App 的 `vite.config.ts` 继承此配置:

```

// apps/admin/vite.config.ts
import { viteConfig } from "@ecoctrl/shared";

export default viteConfig({
  // App 专属覆盖或扩展
  
```

typescript


```
    plugins: [myCustomPlugin],  
  });
```

基座配置包含：

- React 插件配置（@vitejs/plugin-react）
- TypeScript 路径别名映射（resolveUiAlias）
- TailwindCSS 插件配置
- 公共构建优化（压缩、分块策略）
- 开发服务器端口规范

3.3.3. resolveUiAlias — 别名解析插件

`resolveUiAlias()` 是一个 Vite 插件工厂函数，用于将 `@ecoctrl/ui/<component>` 的导入路径解析到源码的准确位置：

```
import { resolveUiAlias } from "@ecoctrl/shared";  
  
// 在 vite.config.ts 中注册  
export default defineConfig({  
  resolve: {  
    alias: [resolveUiAlias()],  
  },  
});
```

typescript

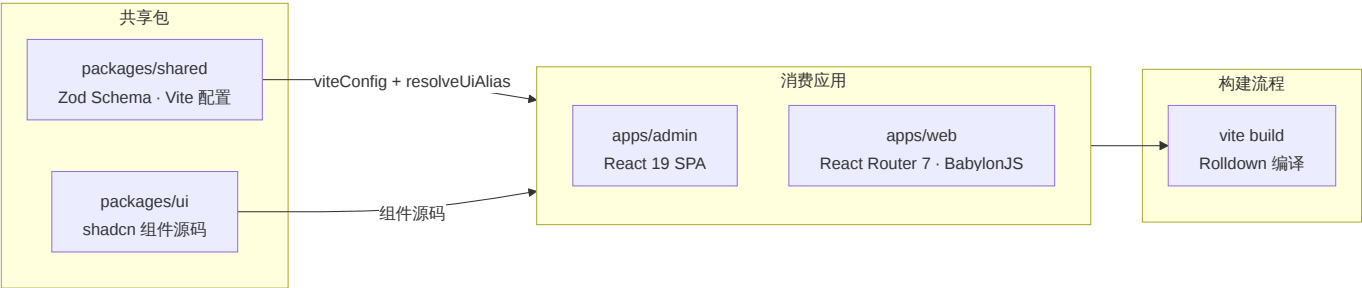
插件内部逻辑：

1. 检测导入源是否为 `@ecoctrl/ui/`
2. 是 → 将路径映射到 `<monorepo-root>/packages/ui/src/`
3. 处理 TypeScript 路径别名以匹配
4. 返回解析后的绝对路径

text

若消费者 App 未注册 `resolveUiAlias()`，Vite 将无法解析 `@ecoctrl/ui/*` 导入，编译时会抛出模块未找到错误。

4. 包依赖图



`packages/server` 不依赖 `@ecoctrl/ui` 或 `@ecoctrl/shared` — 它是独立的 Node.js Fastify 应用，拥有自己的依赖树。

5. 新增组件 workflow

向 `@ecoctrl/ui` 添加新 shadcn 组件的完整流程：

- text
- 确认当前目录为 monorepo 根目录或 `packages/ui` 目录
 - 运行： `pnpm dlx shadcn@latest add <component> -y`
 - 自动下载组件源码到 `packages/ui/src/components/ui/<component>.tsx`
 - 自动添加组件依赖（Radix UI 包等）
 - 运行： `pnpm generate-proxies`
 - 扫描 `src/components/ui/` 目录
 - 更新 `package.json` 的 `exports` 字段添加新子路径
 - 更新 TypeScript 类型声明
 - 在消费者 App 中导入使用：

```
import { Button } from "@ecoctrl/ui/button"
```

6. 构建行为总结

包	是否独立构建	原因
<code>@ecoctrl/ui</code>	否	源码分发，消费者 App 编译时直接包含 <code>.tsx</code> 源文件

包	是否独立构建	原因
@ecoctrl/shared	否	部分为纯 TypeScript 类型，部分为 Zod Schema（消费者端编译时包含）；服务端通过 rollup 打包时一并编译
consumer apps	是	<code>vp build</code> 打包所有依赖（包括 shared packages 源码）为独立 bundle

工作流引擎

工作流引擎是 EcoCtrl 的核心编排子系统，位于 `packages/server/src/engine/`，共 16 个源文件，负责执行 JSON DSL 定义的工作流 DAG。引擎按拓扑顺序依次执行节点，维护 `ExecutionContext` 上下文。Admin 后台提供基于 `@xyflow/react` 的可视化编辑器。

1. 引擎文件结构

engine/	
├─ types.ts	# 核心类型：WorkflowDSL、WorkflowNode、WorkflowEdge、E
├─ validator.ts	# DSL 结构校验（节点 ID、边连通性、必填字段）
├─ executor.ts	# 主执行器 - 顺序执行 DAG
├─ trigger.ts	# 触发器匹配逻辑（判断是否应触发）
├─ expr.ts	# 轻量级表达式求值器（条件和变量插值）
├─ template.ts	# 字符串模板引擎（HTTP 请求体、邮件主题等）
├─ scheduler.ts	# 定时调度（连接 pg-boss cron 注册/注销）
├─ sub-graph.ts	# 子图执行（loop、parallel 等控制节点的子图运行）
├─ upstream-resolver.ts	# 上游依赖解析
├─ env-utils.ts	# 环境变量工具函数
├─ plugin-api.ts	# 插件 API 定义（提供给插件的所有能力接口）
├─ plugin-types.ts	# 插件类型定义（Manifest、Definition、ExecutionContext
├─ plugin-loader.ts	# 插件加载器
├─ plugin-registry.ts	# 插件注册中心
├─ plugin-sandbox.ts	# 插件沙箱（安全隔离）
└─ plugin-executor.ts	# 插件执行器

2. DSL 结构

工作流以 JSON DSL 定义，版本 "1.0"：

```
interface WorkflowDSL {
  version: "1.0";
```

typescript

```

nodes: WorkflowNode[]; // 节点列表
edges: WorkflowEdge[]; // 边列表
envVars?: Array<{
  // 环境变量定义（可选）
  key: string;
  value: unknown;
  type: "string" | "number" | "secret" | "boolean";
  description?: string;
}>;
settings?: WorkflowSettings; // 自动保存等设置
}

```

2.1. WorkflowNode

```

interface WorkflowNode {
  id: string;
  type: string; // 节点类型标识
  name: string;
  config: Record<string, unknown>; // 节点专属配置
  onError?: ErrorHandler; // 错误处理器
  position?: { x: number; y: number }; // 画布位置
}

```

typescript

2.2. WorkflowEdge

```

interface WorkflowEdge {
  id: string;
  source: string;
  target: string;
  sourceHandle?: string; // 用于 condition/switch 条件分支
  targetHandle?: string;
  label?: string;
}

```

typescript

3. 触发器类型

类型	触发方式	配置参数
<code>manual</code>	调用 <code>POST /api/workflows/:id/trigger</code>	无
<code>schedule</code>	基于 Cron 的定时执行	<code>cron: string</code> 、 <code>timezone: string</code>
<code>webhook</code>	调用 <code>POST /api/webhook/:slug</code>	<code>secret?: string</code> 、 <code>allowedIps?: string[]</code>
<code>event</code>	系统内部命名事件	<code>event: string</code> 、 <code>condition?: string</code>

定时触发器通过 pg-boss 调度：创建/更新工作流时自动注册（或注销）对应的 Cron 任务。

4. 节点类型

4.1. 控制节点

节点	用途	说明
<code>start</code>	入口点	每个工作流必须有且只有一个
<code>end</code>	终止节点	停止工作流执行
<code>condition</code>	条件分支	评估布尔表达式，走 <code>true</code> 或 <code>false</code> 边
<code>switch</code>	多路分支	支持多个输出标签
<code>loop</code>	循环	遍历集合，为每项执行子图
<code>parallel</code>	并行	并发执行多个下游分支
<code>delay</code>	延迟	暂停指定毫秒数

4.2. 动作节点

节点	用途	配置要点
<code>http_request</code>	发起 HTTP 调用	<code>method</code> 、 <code>url</code> 、 <code>headers</code> 、 <code>body</code> （支持模板插值）
<code>database</code>	执行 SQL 查询	<code>query</code> 、 <code>params</code>
<code>email</code>	发送邮件	<code>to</code> 、 <code>subject</code> 、 <code>body</code>
<code>variable</code>	设置变量	<code>name</code> 、 <code>value</code> （支持表达式）

5. 执行模型

触发器触发

|

▼

Executor 创建 ExecutionContext:

- `triggerData` (触发载荷)
- `variables` (可变 Map)
- `nodeOutputs` (节点输出记录)
- `env` (进程环境变量)
- `secrets` (敏感配置)

|

▼

按拓扑顺序遍历 DAG:

- 解析节点输入 (模板插值、变量替换)
- 运行节点处理器
- 将输出存入 `nodeOutputs`
- 沿匹配的边进入下一个节点

|

▼

执行结束 → 持久化到 `workflow_executions` 表

6. 错误处理

每个节点可声明 `onError` 处理器:

策略	行为
<code>retry</code>	最多重试 <code>retryCount</code> 次，间隔 <code>retryDelayMs</code> 毫秒
<code>skip</code>	标记为跳过，继续执行下一个连接节点
<code>abort</code>	整个工作流执行失败
<code>goto</code>	跳转到 <code>gotoNodeId</code> 指定的节点（可用于回退路径）

7. 插件系统（5 层架构）

层次	文件	职责
API 层	<code>plugin-api.ts</code>	定义插件可用的所有能力接口（HTTP、IoT、DB、Log、SSE、Env 等）
加载层	<code>plugin-loader.ts</code>	从存储加载插件的代码 + schema
注册层	<code>plugin-registry.ts</code>	管理插件版本、依赖、生命周期
沙箱层	<code>plugin-sandbox.ts</code>	安全隔离插件运行时环境
执行层	<code>plugin-executor.ts</code>	将插件作为工作流节点执行

7.1. Plugin API

插件可访问的能力接口（`PluginApi`）包括：

typescript

```

interface PluginApi {
  variables: { get, set, delete, all };
  http: { get, post, put, patch, delete };
  iot: { readPoint, readPoints, writePoint, writePoints, ... };
  notify: { send, sendMail };
  log: { info, warn, error };
}
    
```



```
env: { get };
context: { workflowId, executionId, triggerData, nodeId, nodeName };
utils: { sleep };
expr: { evaluateBoolean, evaluateExpression };
db: { execute };
sse: { emit };
workflow: { executeSubGraph, executeById };
}
```

7.2. PluginManifest

每个插件定义 `PluginManifest` :

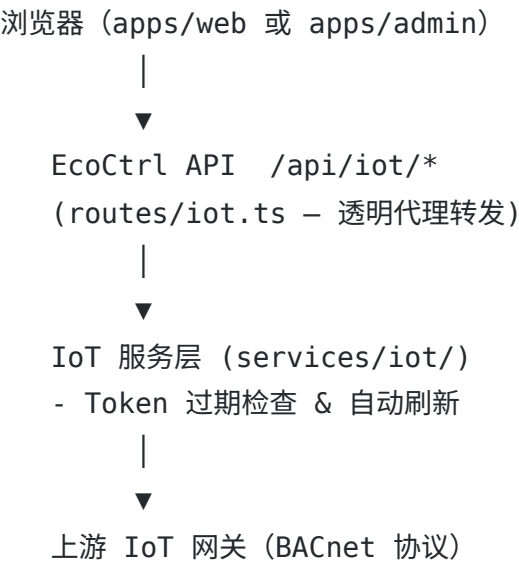
```
interface PluginManifest {
  id: string;
  name: string;
  version: string;
  category: "trigger" | "action" | "condition";
  description?: string;
  entry: string; // 代码入口
  schema: string; // 配置 Schema
  minEngineVersion?: string;
  aliases?: string[];
}
```

typescript

物联网 (IoT) 集成

EcoCtrl 充当上游第三方 IoT 网关的透明代理。前端（Admin/Web）从不直接与网关通信，所有请求经由 EcoCtrl API 流转，由 EcoCtrl 统一处理认证、Token 刷新和数据映射。

1. 架构



2. Token 管理

2.1. iotTokens Schema

`schemas/iotTokens.ts` — 单行 Token 缓存表：

列	类型	说明
<code>id</code>	serial PK	自增主键
<code>accessToken</code>	text	网关的 Bearer Token
<code>refreshToken</code>	text	长生命周期刷新令牌

列	类型	说明
<code>expiresAt</code>	bigint	绝对过期时间 (Unix 毫秒时间戳)

2.2. 自动刷新机制

每次出站请求前，服务层检查 `expiresAt`。如果 Token 在 5 分钟内过期（或已过期），自动执行：

1. 检测 Token 即将过期
2. 用 `refreshToken` 调用网关刷新端点
3. 持久化新 Token 到 `iot_tokens` 表
4. 继续执行原始请求

整个过程对调用者完全透明 — 调用者永远不会看到网关返回 401。

3. 对象模型

3.1. objects Schema

物理设备在 EcoCtrl 中表示为 `objects`：

列	类型	说明
<code>id</code>	uuid PK	主键
<code>code</code>	varchar	上游网关唯一标识符
<code>name</code>	varchar	人类可读标签
<code>type</code>	varchar	类别：传感器、执行器、仪表等
<code>description</code>	text	可选描述
<code>metadata</code>	jsonb	上游属性（单位、量程、协议等）

对象可在 Admin 后台手动创建，或从网关批量同步。

3.2. points Schema

IoT 点位数据模型：

列	类型	说明
<code>id</code>	uuid PK	主键
<code>objectCode</code>	varchar	关联对象的 <code>code</code>
<code>code</code>	varchar	点位标识符
<code>name</code>	varchar	点位名称
<code>value</code>	jsonb	当前值
<code>unit</code>	varchar	单位
<code>updatedAt</code>	timestamptz	更新时间

4. API 路由

所有 `/api/iot/*` 路由需要有效 JWT（非公开）：

方法	路径	说明
GET	<code>/api/iot/token</code>	返回缓存的 Access Token（调试用）
POST	<code>/api/iot/codes/values</code>	读取一个或多个对象码的当前点位值
POST	<code>/api/iot/codes/history</code>	查询点位在时间段内的历史值
POST	<code>/api/iot/codes/set</code>	向可写点位写入值
POST	<code>/api/iot/codes/force-set</code>	强制写入（绕过校验）
POST	<code>/api/iot/alarms</code>	网关告警历史
POST	<code>/api/iot/alarm-configs</code>	告警阈值配置

请求/响应形状与上游网关契约一致。EcoCtrl 原样转发请求体，返回网关响应（已去除凭证头）。

5. 环境变量

变量	必填	说明
<code>BASE_URL</code>	是	上游 IoT 网关基础 URL
<code>APP_ID</code>	是	Token 刷新时使用的网关应用 ID

两者从 `packages/server/.env.local` 读取，转发给 IoT 服务层。

6. Web 门户集成

`apps/web` 仪表盘组件可通过 `dataType: "iot"` 绑定到 IoT 点位值。前端定期轮询 `/api/iot/codes/values` 获取最新读数并渲染。

7. 相关文件

- `packages/server/src/routes/iot.ts` — IoT 代理路由
- `packages/server/src/schemas/iotTokens.ts` — Token 表定义
- `packages/server/src/schemas/objects.ts` — 设备对象表
- `packages/server/src/schemas/points.ts` — 点位数据表
- `packages/server/src/services/iot/` — IoT 服务层

3D 模型管线

EcoCtrl 通过 Babylon.js 支持交互式 3D 建筑可视化。管理员通过 Admin 后台上传 glTF/gLB 模型，Web 门户以可配置的相机角度、环境光和标注热点渲染。管线涉及三条数据流：模型上传、场景配置、实时交互。

1. 模型上传

1.1. 上传路由

`routes/models.ts` 处理模型上传：

- 单文件限制：100 MB
- 支持格式：`.glb`（二进制 glTF，首选）、`.gltf` + `.bin` + 纹理、`.obj`
- 存储位置：取决于 `STORAGE_PROVIDER` 配置
 - `local`（开发环境）：`uploads/models/` 目录
 - `minio`（生产环境）：MinIO 的 `ecoctrl-models` 存储桶
- 对外公开：统一通过 API 代理端点 `/api/models/:id/file` 或 `/api/dashboard-model/file` 访问
- MIME 类型：`model/gltf+json`、`model/gltf-binary`

1.2. 上传流程

1. 客户端 `multipart` 上传
2. 服务器流式写入临时文件
3. 校验（文件大小 $\leq 100\text{MB}$ 、MIME、扩展名）
4. 移动到 `uploads/models/` 最终位置
5. 插入 `models` 表记录（原始文件名、文件大小、MIME 类型、URL）
6. 清理临时文件（`try/catch/finally`）

1.3. Admin UI

- ModelFileZone.tsx — 拖拽上传区域，含格式校验
- ModelViewer.tsx — 基于 Google Model Viewer 的发布前预览

2. 数据模型

2.1. models 表

已上传 3D 模型的元数据：

列	类型	说明
<code>id</code>	uuid PK	主键
<code>name</code>	varchar	原始文件名
<code>fileUrl</code>	varchar	公共 URL: <code>/static/models/<filename></code>
<code>fileSize</code>	bigint	文件大小（字节）
<code>mimeType</code>	varchar	MIME 类型
<code>description</code>	text	可选描述
<code>createdAt</code>	timestampz	创建时间

2.2. dashboard_models 表

单行场景配置：

列	类型	说明
<code>id</code>	serial PK	自增主键
<code>modelFileUrl</code>	varchar	当前模型文件 URL
<code>cameraPreset</code>	varchar	命名相机角度
<code>ambientLightIntensity</code>	real	环境光强度（0-1，默认 0.85）

列	类型	说明
hotspots	jsonb	热点标记数组 {x, y, z, label, targetId}
labels	jsonb	标签数组 {position: {x,y,z}, text, color}
modelFiles	jsonb	多模型文件列表（含角色绑定、优先级、排序）
updatedAt	timestampz	更新时间

3. 场景编辑器

Admin 后台的 3D 场景编辑器（ `DashboardModel.tsx` ）支持 3 种编辑模式：

3.1. model 模式

管理模型文件：上传、排序、设置优先级（ `critical` / `background` ）、切换可见性。支持多模型文件分层加载。

3.2. label 模式

在 3D 场景中放置和编辑标签。编辑器通过 `modelEditorStore`（ `apps/admin/src/store/modelEditorStore.ts` ）管理状态：

```
type EditorMode = "select" | "placeLabel" | "clipPreview";
```

typescript

- 在 3D 场景中点击选取位置
- 标签包含锚点坐标、树结构、动作绑定
- 标签可嵌套（父子关系）
- 自动保存

3.3. action 模式

配置标签的交互动作。每个标签可绑定多个动作，包括：

- 显示弹窗信息

- 导航到特定系统页面
- 显示关联 IoT 数据
- 触发照明控制

4. 热点交互系统

热点是放置在 3D 坐标处的可交互标记：

- 点击显示 `label` 文本提示
- 导航到特定楼层或系统页面
- 显示关联 `targetId` 的实时 IoT 数据

标签是固定在 3D 位置的持久文本注释，适用于标记建筑分区、设备间或能耗区域。

5. Web 门户渲染

`apps/web/app/components/dashboard/building-view.tsx` 加载模型：

1. GET `/api/public/model` 获取场景配置
2. 创建 `Babylon.js Engine + Scene`
3. `SceneLoader.ImportMeshAsync` 加载模型
4. 应用相机预设和环境光强度
5. 渲染热点和标签

5.1. 相机预设

预设	说明
<code>Default_View_01</code>	正面概览
<code>Top_View</code>	鸟瞰图
<code>Side_View</code>	左侧立面

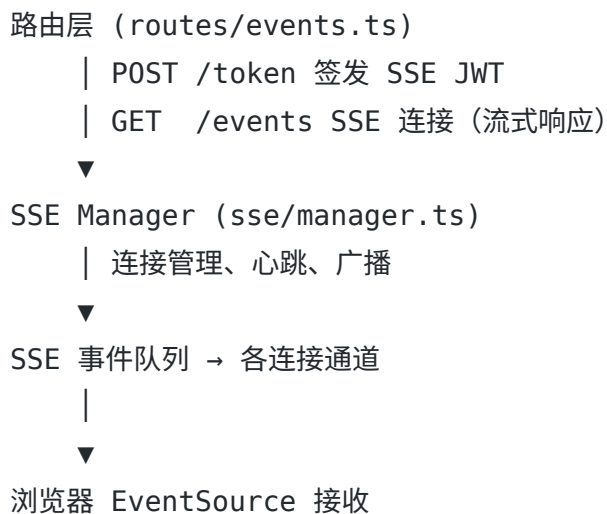
6. API

方法	路径	说明
GET	/api/public/model	公开场景配置（无需认证）
GET	/api/models	列出已上传模型（需认证）
POST	/api/models	上传新模型
GET	/api/models/:id	模型元数据
GET	/api/models/:id/file	流式获取模型文件
DELETE	/api/models/:id	删除模型和文件
GET	/api/dashboard-model	获取全部场景配置
PUT	/api/dashboard-model	更新场景配置

实时通信 (SSE)

EcoCtrl 使用 Server-Sent Events (SSE) 实现服务器到客户端的实时数据推送。

1. 架构



2. 令牌认证

SSE 连接使用独立的短期 JWT 令牌：

- 先调用 `POST /api/events/token` 获取令牌 (30 秒有效期)
- 使用令牌建立 SSE 连接 `GET /api/events?token=xxx`
- 令牌中包含 `userId` 和 `purpose: "sse"`
- 30 秒过期后需重新获取

3. 心跳机制

- 服务端每 30 秒发送心跳事件

- 心跳包含事件类型 `heartbeat` 和当前时间戳
- 客户端检测到心跳停止超过 90 秒可判定连接断开

4. 事件类型

事件	说明	payload 结构
<code>widget_update</code>	仪表盘组件数据更新	<code>{ metricKey, type, data }</code>
<code>widget_delete</code>	仪表盘组件数据清除	<code>{ metricKey }</code>
<code>lighting_update</code>	照明状态变更	<code>{ labelId, groups }</code>

5. 队列积压处理

- 事件先入队到内存队列，再逐个发送
- 队列积压时优先发送最新事件
- 连接断开时清空积压，避免重连后大量过期事件暴发

6. 数据流

Admin/Server 端发送事件 → SSE Manager

→ 序列化 → 写入响应流 (text/event-stream)

→ 浏览器 EventSource → parse → store dispatch → UI 更新

AI 与智能助手

EcoCtrl 集成了 AI 对话能力和虚拟宠物（Pet）系统。AI 子系统提供多 Provider 的大语言模型调用、SSE 流式输出、可扩展工具注册和会话管理。Pet 子系统管理精灵模型上传、贴图提取和用户偏好设置。

1. AI 提供者架构

AI 系统（`routes/ai.ts`）支持两个 LLM 提供者，通过环境变量切换：

环境变量	说明
<code>AI_PROVIDER</code>	<code>anthropic</code> 或 <code>openai</code>
<code>AI_API_KEY</code>	API 密钥
<code>AI_BASE_URL</code>	自定义 API 端点（可选，用于代理或兼容接口）
<code>AI_MODEL</code>	模型名称

1.1. Provider 适配

系统通过统一的 Provider 接口封装两个 SDK：

- Anthropic：使用 `@anthropic-ai/sdk`，支持 Claude 系列模型（如 `claude-sonnet-4-20250514`）
- OpenAI：使用 `openai` SDK，支持 GPT-4o 等模型

Provider 的选择在服务启动时通过环境变量固定，运行时不再切换。

2. 对话流程

```
客户端发送 POST /api/ai/chat
{ message, sessionId?, context: { currentPage, currentRouteData? } }
```



2.1. SSE 流式输出

`POST /api/ai/chat` 接口通过 SSE 逐块返回 LLM 生成内容：

```

事件: message_start    → { sessionId, timestamp }
事件: content_block    → { type: "text", text: "..."}
事件: content_block    → { type: "tool_use", name, input }
事件: tool_result      → { toolCallId, result }
事件: message_complete → { stopReason, usage }
  
```

3. 工具调用

AI 可以调用系统工具来获取实时数据。工具通过 tool registry (`@/ai/tools/registry`) 注册：

```

interface AiTool {
  name: string;
  description: string;
  parameters: Record<string, unknown>;
}
  
```

typescript

```
    handler: (args: unknown) => Promise<unknown>;
}
```

内置工具包括：

工具名称	功能
<code>getDeviceStatus</code>	查询设备运行状态
<code>getEnergyData</code>	查询能耗统计数据
<code>getFaultList</code>	查询当前告警列表
<code>getWeather</code>	查询天气信息

工具调用流程：LLM 返回 `tool_use` 块 → 服务器执行 handler → 结果回传给 LLM → LLM 生成最终回复。

4. 对话管理

- Session 基于 `aiConversations` 表持久化
- 每个 Session 保留完整对话历史（`messages` 数组）
- 支持创建新对话、切换对话和删除历史对话
- 超过 token 限制时自动裁剪早期消息
- 会话 24 小时无活动后自动标记为过期

5. Pet 系统

Pet 系统（ `routes/pets.ts` ）管理数字宠物的外观和行为配置：

端点	方法	说明
<code>/api/pets</code>	GET	获取可用宠物列表
<code>/api/pets/:id/spritesheet</code>	GET	获取宠物的精灵图

端点	方法	说明
/api/pets	POST	上传新宠物（ZIP 包格式，含 spritesheet + metadata）
/api/pets/:id	DELETE	删除宠物

5.1. 用户偏好 API

宠物偏好通过 AI preferences API 管理：

偏好项	说明	可选值
theme	主题色	light / dark / custom
voiceEnabled	语音反馈开关	true / false
voiceSpeed	语音速度	slow / normal / fast
petPositionX/Y	屏幕位置坐标	0 - 100
wakeWordEnabled	唤醒词开关	true / false

偏好存储在数据库 [pets_preferences](#) 表中，每次更新后可通过 SSE 推送到前端。

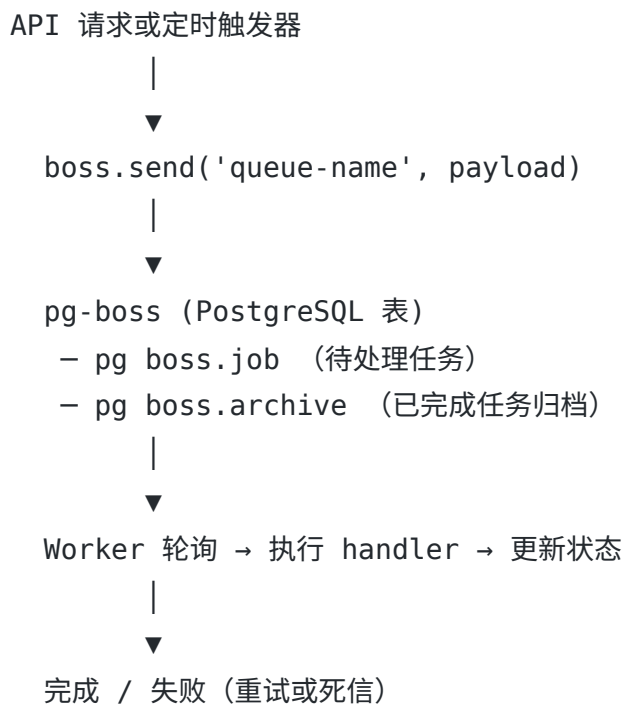
6. 相关文件

- [packages/server/src/routes/ai.ts](#) — AI 聊天路由
- [packages/server/src/routes/pets.ts](#) — Pet 路由
- [packages/server/src/services/ai/](#) — AI 服务层
- [packages/server/src/services/ai/tools.ts](#) — 工具注册表
- [packages/server/src/schemas/pets.ts](#) — Pet 数据模型

任务队列系统

EcoCtrl 使用 [pg-boss](#) 实现可靠的后台任务处理。所有任务存储在 PostgreSQL 中，由运行在 API 服务同一进程中的 Worker 消费。

1. 架构



`pg-boss` 直接使用 Fastify 所在的 PostgreSQL 数据库连接，不需要独立的队列基础设施。任务元数据和状态都存储在数据库的 `pgboss` schema 下，简化了运维复杂度。

2. 初始化

队列系统通过 `queue/pgboss.ts` 初始化：

```
import PgBoss from "pg-boss";

const boss = new PgBoss({
```

typescript

```
    schema: "pgboss", // 独立 schema 避免表名冲突
    connectionString: DATABASE_URL,
  });

  await boss.start();
```

关键初始化参数：

参数	说明
schema	独立 schema 名称，默认 pgboss
connectionString	PostgreSQL 连接字符串
pollIntervalSeconds	轮询新任务间隔（默认 2 秒）
retentionDays	完成任务保留天数（默认 30 天）

3. 任务类型

任务	队列名	说明
报表生成	report-generation	异步生成 PDF/CSV 能耗报表
数据库备份	backup	定时备份数据库到存储
工作流执行	workflow-execution	执行工作流 DSL 定义的任务

3.1. 发送任务

```
await boss.send(
  "report-generation",
  {
    reportId: "rpt_123",
    format: "pdf",
    userId: "user_456",
  },
  {
```

typescript

```
    retryLimit: 3,  
    retryDelay: 60, // 秒  
  },  
);
```

4. Worker

Worker 在 `queue/worker.ts` 中注册:

```
import { getWorker } from "../queue/pgboss";  
  
const worker = await getWorker();  
  
// 注册报表生成处理器  
await worker.work("report-generation", async (job) => {  
  const { reportId, format } = job.data;  
  // 执行报表生成...  
});
```

typescript

4.1. 运行模式

- 同一 Node.js 进程中运行（不独立部署）
- 开发环境自动启动（伴随 API 服务）
- 生产环境同进程运行
- 支持优雅关闭（监听 SIGTERM 信号完成当前任务后退出）

5. 重试与死信

5.1. 指数退避

任务失败后按指数增长间隔重试:

- 第 1 次重试: 60 秒后

- 第 2 次重试：120 秒后
- 第 3 次重试：240 秒后
- ...

退避公式： `retryDelay * 2^(attempt - 1)` ，默认初始延迟 60 秒。

5.2. 死信策略

条件	行为
重试次数未达上限	按指数退避重新排队
达到最大重试次数	任务标记为 <code>failed</code> 进入死信状态
死信任务	保留在 <code>pgboss.job</code> 表中，状态为 <code>failed</code> ，可供人工审查
手动重新排队	通过 pg-boss API 或 SQL 可将死信重新加入队列

5.3. 死信重入

sql

```
-- 查看死信任务
SELECT * FROM pgboss.job WHERE state = 'failed';

-- 手动重新排队（设置新状态为 created）
UPDATE pgboss.job SET state = 'created' WHERE id = 123;
```

6. 监控

通过 pg-boss 内置表监控队列状态：

表	内容
<code>pgboss.job</code>	所有任务记录（含待处理、运行中、失败）
<code>pgboss.archive</code>	已完成任务归档
<code>pgboss.schedule</code>	定时任务配置

6.1. 常用监控查询

sql

```
-- 各队列待处理任务数
SELECT name, COUNT(*) FROM pgboss.job
WHERE state = 'created' GROUP BY name;

-- 最近失败的报表生成任务
SELECT * FROM pgboss.job
WHERE name = 'report-generation' AND state = 'failed'
ORDER BY created_on DESC LIMIT 10;
```

7. 相关文件

- [packages/server/src/queue/pgboss.ts](#) — pg-boss 初始化与配置
- [packages/server/src/queue/worker.ts](#) — Worker 注册与处理器
- [packages/server/src/queue/types.ts](#) — 任务相关类型定义

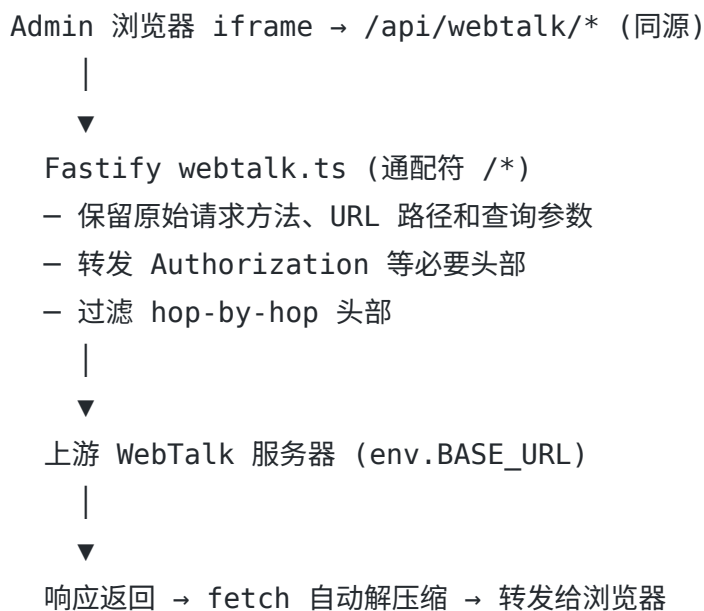
WebTalk 反向代理

1. 解决的问题

WebTalk 是一个第三方 IoT 设备管理平台。Admin 后台通过 iframe 嵌入其管理界面时，浏览器同源策略会阻止跨域 Cookie 的传递，导致会话无法维持。

WebTalk 反向代理（[routes/webtalk.ts](#)）通过通配符路由 `/*` 将所有 `/api/webtalk/*` 请求透明地转发到上游的 WebTalk 服务，使 iframe 保持与 Admin 后台同源，从而解决跨域 Cookie 问题。

2. 工作原理



3. 实现细节

3.1. 请求转发

代理转发保留原始请求的完整性：

- 保留原始 HTTP 方法（GET、POST、PUT、DELETE 等）
- 保留原始 URL 路径和查询参数
- 转发 `Authorization` 和 `Content-Type` 等必要头
- 支持请求体透传（JSON、表单、二进制）

3.2. 头部过滤

转发响应时会过滤 hop-by-hop 头部。这些头部是 HTTP 逐跳元数据，不应转发给客户端：

- `connection`
- `keep-alive`
- `transfer-encoding`
- `te`
- `trailer`
- `upgrade`
- `proxy-authorization`
- `proxy-authenticate`

3.3. 响应处理

- 使用 `fetch` API 向上游发起请求，利用其原生解压缩能力
- 自动处理 `gzip`、`deflate`、`br`（Brotli）等压缩编码
- 响应头部同样过滤 `content-encoding` 和 `content-length`，避免与分块传输冲突
- 支持大响应分块传输（streaming）
- 设置 `cache-control` 为 `no-store` 防止响应被缓存

3.4. 代理逻辑伪代码

```
async function proxy(req, reply) {  
  const upstreamUrl = `${env.BASE_URL}${req.url}`;  
  const headers = filterHopByHop(req.headers);  
  
  const upstreamRes = await fetch(upstreamUrl, {
```

typescript

```
    method: req.method,
    headers,
    body: req.method !== "GET" ? req.body : undefined,
  });

// 过滤响应头中的 hop-by-hop 头部
const responseHeaders = filterHopByHop(upstreamRes.headers);

reply.code(upstreamRes.status);
reply.headers(responseHeaders);
reply.send(upstreamRes.body); // 流式发送
}
```

4. 安全

- WebTalk 代理路径需要有效的 JWT 认证（同 Admin API）
- 上游 URL 从 `env.BASE_URL` 读取，不暴露给客户端
- 不会将上游认证凭据透传给浏览器
- 对上游响应做严格的头部白名单过滤

5. 配置

环境变量	说明
<code>WEBTALK_BASE_URL</code>	上游 WebTalk 服务器基础 URL

6. 相关文件

- `packages/server/src/routes/webtalk.ts` — WebTalk 反向代理路由
- `packages/server/src/utils/http.ts` — HTTP 工具函数（hop-by-hop 过滤等）

碳排放因子系统

碳排放因子系统将能耗数据转换为碳排放量。因子值定义了每单位能耗对应的碳排放量（如 [kgCO2/kWh](#) ）。系统采用层次化的分类树结构存储因子，支持与国家温室气体排放因子数据库同步，并与 Energy 模块联动实现碳核算。

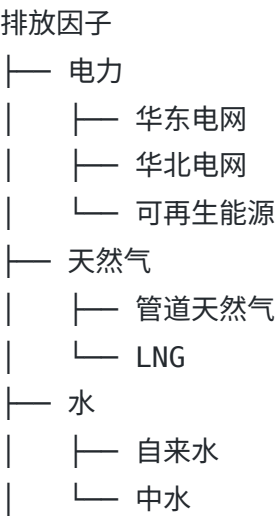
1. 数据模型

系统包含两张核心表：[carbon_factor_nodes](#) 和 [carbon_factors](#) ，分别存储分类树结构和具体的因子值。

1.1. carbon_factor_nodes（分类树）

字段	类型	说明
id	serial PK	主键
name	varchar	分类名称（如「电力」「天然气」「水」「热力」）
parentId	int FK	父节点 ID，支持多级嵌套
isExpandable	boolean	是否有子节点（用于前端展开/折叠）

分类树示例：



- └─ 热力
 - └─ 集中供热

1.2. carbon_factors（因子值）

字段	类型	说明
id	serial PK	主键
nodeId	int FK	所属分类节点 ID
name	varchar	因子名称（如「华东电网电力排放因子」）
value	numeric	因子数值（如 0.7035）
unit	varchar	单位（如 kgC02/kWh 、 tC02/万m3 ）
source	text	数据来源说明（如「国家发改委 2024 年数据」）
effectiveYear	int	生效年份
createdAt	timestamptz	创建时间
updatedAt	timestamptz	更新时间

2. 与国家数据库同步

系统可连接到国家温室气体排放因子数据库进行增量同步：

- 1. 定时任务（每日）触发同步
- 2. 请求国家数据库 API，获取增量更新
- 3. 新增因子 → 自动插入分类树和因子表
- 4. 已有因子 → 覆盖更新（保留版本记录）
- 5. 同步完成后触发事件 → 重新计算引用该因子的历史碳排放

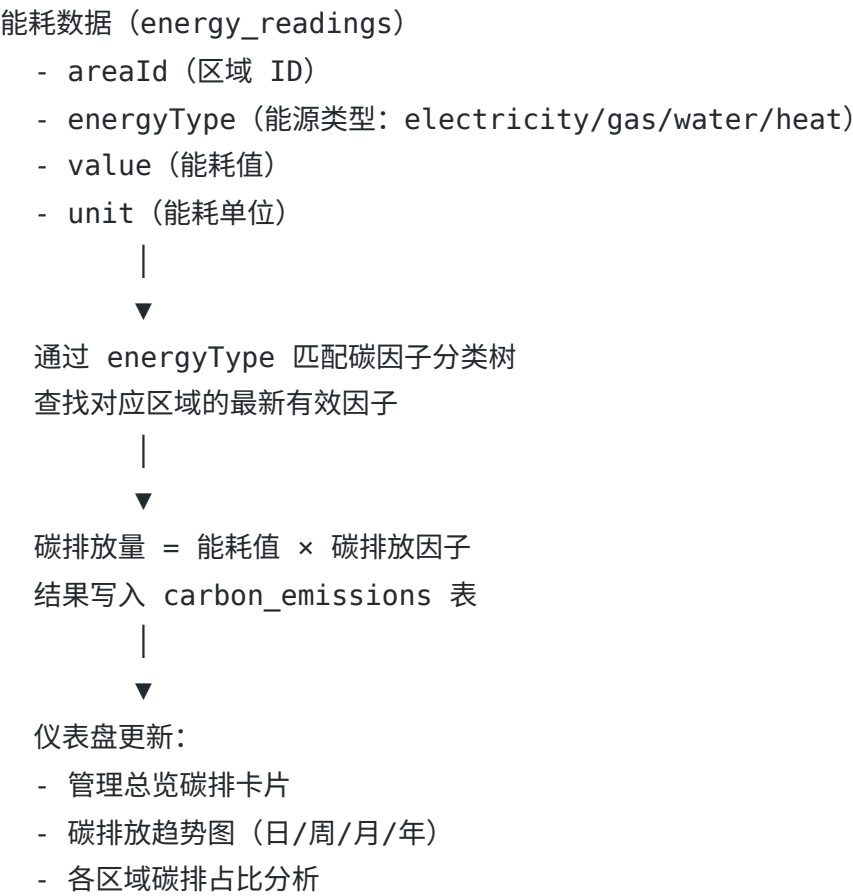
2.1. 同步策略

变更类型	处理方式
新增因子	自动插入对应的分类节点和因子记录
值变更	覆盖更新，保留 <code>source</code> 变更记录
废弃因子	标记为 <code>deprecated</code> ，不影响已有历史数据

同步完成后，所有引用已变更因子的历史碳排放数据自动进入重新计算队列（通过 pg-boss 异步处理）。

3. 与 Energy 模块联动

碳排放因子系统与 Energy（能耗）模块紧密集成，实现从能耗到碳排放的自动转换：



3.1. 碳排计算示例

typescript

```
// 简化的碳排计算
function calculateCarbonEmission(energyValue: number, energyType: string, region: string) {
  const factor = findFactor(energyType, region);
  return energyValue * factor.value;
}
```

4. API 端点

方法	路径	说明
GET	/api/carbon-factors	获取因子分类树（含因子值）
PUT	/api/carbon-factors	更新因子值（需管理员权限）
GET	/api/carbon-factors/sync	触发与数据库的同步
GET	/api/carbon-emissions	查询碳排放数据（支持按区域、时间筛选）

5. 相关文件

- [packages/server/src/schemas/carbonFactors.ts](#) — 因子值表定义
- [packages/server/src/schemas/carbonFactorNodes.ts](#) — 分类树表定义
- [packages/server/src/routes/carbon.ts](#) — 碳因子与排放 API 路由
- [packages/server/src/services/carbon/](#) — 碳计算服务

照明控制系统

照明控制系统提供对照明组状态的查询与控制能力。用户通过 3D 场景点击区域标签，打开 LightingSheet 侧边面板进行开关操作。后端优先通过 IoT 服务读写物理点位，在 IoT 离线时以内存 Mock 模式降级运行。

1. 整体架构



2. 照明状态模型

每个照明分组有三种状态：

状态	含义
on	组内所有灯具全部开启
off	组内所有灯具全部关闭
half	部分开启（组内有开有关混合状态）

3. API 端点

方法	路径	说明
GET	/api/control/lighting/:labelId/status	查询某区域的照明组状态列表
POST	/api/control/lighting/:labelId/toggle	切换某照明组的状态
POST	/api/control/lighting/:labelId/batch	批量全部打开或全部关闭

3.1. 查询状态

[GET /api/control/lighting/:labelId/status](#) 返回：

```
{
  "labelId": "area_1",
  "groups": [
    { "id": "g1", "name": "主照明", "status": "on" },
    { "id": "g2", "name": "氛围灯", "status": "half" },
    { "id": "g3", "name": "应急照明", "status": "off" }
  ]
}
```

json

3.2. 切换与批量控制

- `toggle`：单个照明组在 `on` ↔ `off` 之间切换（如果当前是 `half`，先切换到 `on`）
- `batch`：支持全部打开（`action: "on"`）或全部关闭（`action: "off"`）

后端更新状态后，通过 SSE 广播 `lighting_update` 事件通知所有连接的浏览器同步。

4. 前端的集成

4.1. LightingSheet 组件

[apps/web/app/components/dashboard/lighting-sheet.tsx](#)：

- 320px 宽侧边面板
- 区域选择器下拉框（label 列表）
- 每组显示 light-bulb 图标（ on / half / off 三种视觉样式）
- 点击灯泡图标切换状态
- 支持批量操作按钮
- 乐观更新模式：先更新 UI，服务端确认失败后回滚

4.2. Zustand Store

照明状态使用 Zustand 管理（ [store/lighting.ts](#) ）：

typescript

```
interface LightingStore {
  groupsByLabel: Record<string, LightingGroup[]>;

  initGroups: (labelId: string, groups: LightingGroup[]) => void;
  updateGroup: (labelId: string, groupId: string, status: LightStatus) => void;
  mergeGroups: (labelId: string, groups: LightingGroup[]) => void;
}
```

方法	说明
<code>initGroups</code>	首次加载区域照明组数据
<code>updateGroup</code>	更新单个组状态（操作后调用）
<code>mergeGroups</code>	合并 SSE 推送的批量状态更新

5. Mock 降级策略

当 IoT 服务不可用时（网络故障、网关离线），照明路由内置的 mock 降级机制提供内存级状态模拟：

- 从 `dashboard_models` 的 label 配置中推断照明分组
- 状态保存在内存 Map 中（keyed by point ID）
- 支持完整的查询、切换、批量操作

5.1. Mock 状态聚合规则

`getMockStatus(labelId)` 聚合每个分组下的点位状态：

点位状态集合	聚合结果
全部 <code>on</code>	<code>on</code>
全部 <code>off</code>	<code>off</code>
混合 <code>on</code> 和 <code>off</code>	<code>half</code>

Mock 模式对所有 API 调用方透明 — 前端不需要感知 IoT 是否在线。

6. SSE 更新

照明状态变更通过 SSE Manager 广播 `lighting_update` 事件：

```
{
  "type": "lighting_update",
  "data": {
    "labelId": "area_1",
    "groups": [{ "id": "g1", "name": "主照明", "status": "off" }]
  }
}
```

json

前端通过 `mergeGroups` 方法自动合并新状态到 Zustand store，触发 UI 重新渲染。

7. 相关文件

- `packages/server/src/routes/lighting.ts` — 照明控制路由（含 Mock 降级逻辑）

认证与用户 API

认证与用户模块覆盖用户注册登录、Token 生命周期管理、用户 CRUD、个人设置以及 OAuth 第三方登录。

所有端点（除标记为公共外）均需携带 `Authorization: Bearer <accessToken>` 。Access Token 默认 15 分钟过期，可通过 `/api/auth/refresh` 轮换。

完整的 API 参考可在运行时通过 Swagger UI 查阅（本地开发：
<http://localhost:3000/documentation> ，生产环境： </documentation> ）

1. 登录与注册

1.1. POST /api/auth/login

认证: 公共

使用用户名或邮箱 + 密码登录，返回 Access Token 与 Refresh Token。

字段	类型	说明
<code>usernameOrEmail</code> (body)	string	用户名或注册邮箱
<code>password</code> (body)	string	用户密码
响应 <code>accessToken</code>	string	JWT Access Token（默认 15 分钟）
响应 <code>refreshToken</code>	string	JWT Refresh Token（用于轮换）
响应 <code>user</code>	object	当前用户基本信息

1.2. POST /api/auth/register

认证: 公共

创建新用户。调用前需先通过 `/api/auth/register/send-code` 发送验证码。

字段	类型	说明
<code>username</code> (body)	string	用户名
<code>email</code> (body)	string	电子邮箱
<code>password</code> (body)	string	密码
<code>code</code> (body)	string	6 位验证码（5 分钟有效）

1.3. POST /api/auth/refresh

认证: 公共

用 Refresh Token 换取新的 Access Token + Refresh Token 对。调用后原 Refresh Token 立即失效。

字段	类型	说明
<code>refreshToken</code> (body)	string	当前有效的 Refresh Token
响应 <code>accessToken</code>	string	新 Access Token
响应 <code>refreshToken</code>	string	新 Refresh Token

2. 密码管理

2.1. POST /api/auth/forgot-password/send-code

认证: 公共

向注册邮箱发送 6 位密码重置验证码。

字段	类型	说明
<code>email</code> (body)	string	注册邮箱

2.2. POST /api/auth/forgot-password/reset

认证: 公共

凭验证码重置密码。

字段	类型	说明
<code>email</code> (body)	string	注册邮箱
<code>code</code> (body)	string	验证码
<code>newPassword</code> (body)	string	新密码

3. 当前用户

3.1. GET /api/auth/me

返回已认证用户的详细信息。

响应字段	类型	说明
<code>id</code>	number	用户 ID
<code>username</code>	string	用户名
<code>email</code>	string	邮箱
<code>avatar</code>	string	头像 URL
<code>role</code>	string	用户角色
<code>createdAt</code>	string	创建时间

4. 用户管理 (CRUD)

基础路径: `/api/users/:id`

4.1. GET /api/users/:id

获取指定用户的详细信息。需具有对应用户或管理员权限。

4.2. PUT /api/users/:id

更新指定用户的字段（如邮箱、昵称、头像）。

字段	类型	说明
<code>email</code> (body, 可选)	string	新邮箱
<code>username</code> (body, 可选)	string	新用户名
<code>avatar</code> (body, 可选)	string	头像文件 ID

5. 用户设置

5.1. GET /api/settings

返回当前用户的个性化设置。

响应示例	
<code>theme</code>	<code>"light"</code> / <code>"dark"</code> / <code>"system"</code>
<code>language</code>	<code>"zh-CN"</code> / <code>"en"</code>
<code>timezone</code>	<code>"Asia/Shanghai"</code>
<code>notification</code>	object (通知偏好)

5.2. PUT /api/settings

更新当前用户的设置。仅需提供要修改的字段，未提供的字段保持不变。

6. OAuth 第三方登录

6.1. POST /api/auth/oauth/:provider/authorize

认证: 公共

参数	说明
<code>:provider</code> (path)	支持 <code>wechat</code> / <code>feishu</code> 等
响应 <code>redirectUrl</code>	跳转到第三方授权页面的 URL

6.2. GET /api/auth/oauth/:provider/callback

认证: 公共

第三方授权回调入口。根据回调参数自动完成登录或跳转。

6.3. POST /api/auth/oauth/bind

将 OAuth 身份绑定到当前已登录账号。

字段	类型	说明
<code>provider</code> (body)	string	OAuth 提供商
<code>code</code> (body)	string	授权码

6.4. POST /api/auth/oauth/register-and-bind

使用第三方 OAuth 返回的用户资料自动创建账号并绑定。

字段	类型	说明
<code>provider</code> (body)	string	OAuth 提供商
<code>code</code> (body)	string	授权码
响应		自动创建用户 + 绑定 + 签发 Token

完整的 API 参考可在运行时通过 Swagger UI 查阅（本地开发：
<http://localhost:3000/documentation>，生产环境：</documentation>）

数据模型与对象 API

数据模型与对象模块管理物联网场景中的三层抽象：模型（3D 模型文件）、对象（物理设备/逻辑实体）、点位（数据采集点）。所有端点均需认证。

完整的 API 参考可在运行时通过 Swagger UI 查阅（本地开发：
<http://localhost:3000/documentation>，生产环境：[/documentation](#)）

1. 模型

模型对应上传的 3D 模型文件（如建筑结构、设备模型）。文件存储位置取决于 `STORAGE_PROVIDER` 配置：local 模式落盘到 `uploads/models/`，minio 模式存储于 MinIO `ecoctrl-models` 存储桶。统一通过 API 代理端点 `/api/models/:id/file` 访问。

1.1. GET /api/models

列出所有已上传模型。支持分页参数。

查询参数	类型	说明
<code>page</code>	number	页码，默认 1
<code>limit</code>	number	每页数量，默认 20
<code>search</code>	string	按名称搜索
响应	array	模型列表

1.2. POST /api/models

上传新模型。

字段	类型	说明
<code>name</code> (body)	string	模型名称

字段	类型	说明
<code>description</code> (body, 可选)	string	描述
<code>file</code> (multipart)	file	3D 模型文件
响应 <code>id</code>	number	新模型 ID

1.3. GET /api/models/:id/file

流式获取模型文件二进制内容。注意：不存在 `GET /api/models/:id`；获取模型文件应使用此端点。

1.4. PUT /api/models/:id

更新模型元数据。

字段	类型	说明
<code>name</code> (body, 可选)	string	新名称
<code>description</code> (body, 可选)	string	新描述
<code>version</code> (body, 可选)	string	版本号

1.5. DELETE /api/models/:id

删除模型及其关联文件。

1.6. POST /api/models/:id/import-points

为模型关联的业务对象批量导入点位。

字段	类型	说明
<code>points</code> (body)	array	点位定义列表

2. 业务对象

对象代表上游 IoT 网关中的物理设备或逻辑数据单元。

2.1. GET /api/objects

列出所有业务对象。

查询参数	类型	说明
<code>modelId</code>	number	按模型筛选
<code>page</code>	number	页码
<code>limit</code>	number	每页数量

2.2. POST /api/objects

创建业务对象。

字段	类型	说明
<code>name</code> (body)	string	对象名称
<code>modelId</code> (body)	number	关联模型 ID
<code>code</code> (body)	string	对象编码
<code>description</code> (body, 可选)	string	描述

2.3. GET /api/objects/:id

获取业务对象详情，包含关联的点位列表。

2.4. PUT /api/objects/:id

更新业务对象信息。

2.5. DELETE /api/objects/:id

删除业务对象及关联的点位数据。

3. 点位

点位是数据采集的最小单元，对应 IoT 网关中的一个数据通道。

3.1. GET /api/points

列出点位。

查询参数	类型	说明
<code>objectId</code>	number	按业务对象筛选
<code>page</code>	number	页码
<code>limit</code>	number	每页数量

3.2. POST /api/points

创建点位。

字段	类型	说明
<code>name</code> (body)	string	点位名称
<code>code</code> (body)	string	点位编码
<code>objectId</code> (body)	number	所属对象 ID
<code>unit</code> (body, 可选)	string	单位
<code>dataType</code> (body, 可选)	string	数据类型

3.3. GET /api/points/:id

获取点位详情。

3.4. PUT /api/points/:id

更新点位配置。

3.5. DELETE /api/points/:id

删除点位。

3.6. POST /api/points/import

批量导入点位数据。

字段	类型	说明
<code>points</code> (body)	array	点位定义数组
<code>objectId</code> (body)	number	默认关联对象 ID
响应 <code>imported</code>	number	成功导入数量
响应 <code>failed</code>	number	失败数量

完整的 API 参考可在运行时通过 Swagger UI 查阅（本地开发：
<http://localhost:3000/documentation>，生产环境：</documentation>）

能源与碳排放 API

能源与碳排放模块提供区域能耗概览、详细数据查询、统计计算以及碳排放因子管理。所有端点均需认证。

完整的 API 参考可在运行时通过 Swagger UI 查阅（本地开发：
<http://localhost:3000/documentation>，生产环境：[/documentation](#)）

1. 区域能耗总览

1.1. GET /api/energy/areas

返回各区域当前能耗快照，以卡片形式呈现。

响应字段	类型	说明
<code>areas</code>	array	区域列表
<code>areas[].name</code>	string	区域名称
<code>areas[].currentPower</code>	number	当前功率 (kW)
<code>areas[].energyToday</code>	number	今日累计能耗 (kWh)
<code>areas[].energyMonth</code>	number	本月累计能耗 (kWh)
<code>areas[].status</code>	string	能耗状态： <code>normal</code> / <code>warning</code> / <code>overLimit</code>

2. 详细能耗数据

2.1. GET /api/energy/details

按时间范围查询精细粒度能耗数据。

查询参数	类型	说明
areaId	number (可选)	区域 ID，不传则返回全部区域
from	string (ISO 8601)	起始时间
to	string (ISO 8601)	结束时间
interval	string	聚合粒度： hour / day / week / month
响应	array	时序数据点列表

每个数据点包含 `timestamp`、`value`、`unit`、`areaId`。

3. 能耗统计

3.1. GET /api/energy/stats

返回选定时间范围的能耗统计分析。

查询参数	类型	说明
areaId	number (可选)	区域筛选
period	string	统计周期： today / thisWeek / thisMonth / custom
from / to	string	custom 周期时必填

响应字段	类型	说明
total	number	总能耗 (kWh)
avg	number	平均能耗
peak	number	峰值能耗
peakTime	string	峰值发生时间
comparedToLastPeriod	object	环比变化百分比

4. 碳排放因子

4.1. GET /api/energy/carbon-factors

返回碳排放因子树形结构。植被按类别层级组织（如电力、天然气、水等）。

响应字段	类型	说明
factors	array	因子树，每个节点含 id 、 name 、 value 、 unit 、 children

4.2. POST /api/energy/carbon-factors/refresh

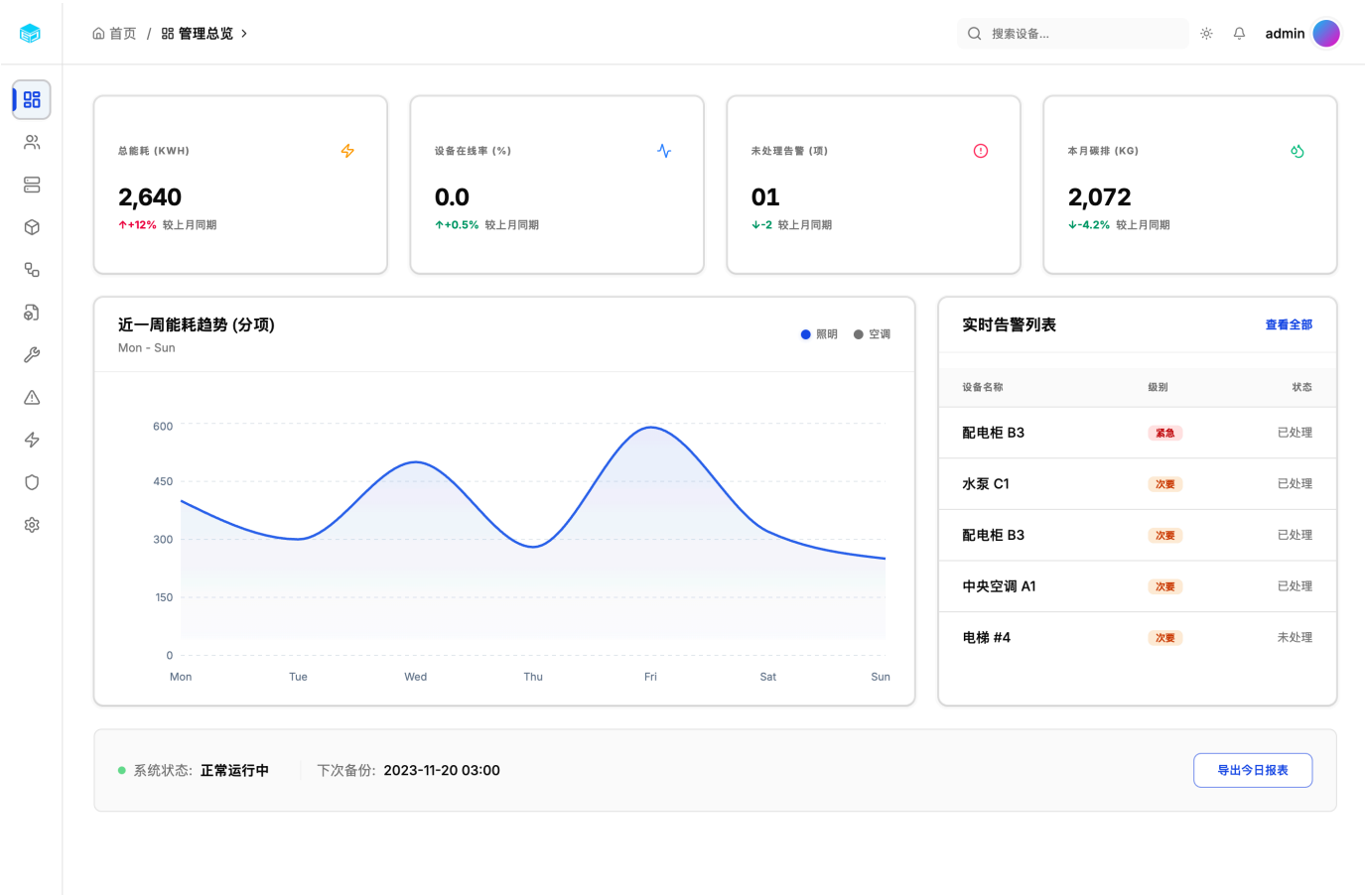
触发碳排放因子数据同步/刷新。从上游数据源重新拉取最新的碳排放因子。

响应	
success	boolean
message	刷新结果说明

完整的 API 参考可在运行时通过 Swagger UI 查阅（本地开发：
<http://localhost:3000/documentation> ， 生产环境： [/documentation](#) ）

workflow与执行 API

workflow模块提供自动化流程的定义、触发、执行监控及插件节点扩展能力。所有端点均需认证。



1. workflow CRUD

1.1. GET /api/workflows

列出所有 workflow 定义。

查询参数	类型	说明
page	number	页码
limit	number	每页数量
enabled	boolean (可选)	按启停状态筛选

查询参数	类型	说明
响应	array	工作流列表

1.2. POST /api/workflows

创建工作流定义。

字段	类型	说明
<code>name</code> (body)	string	工作流名称
<code>description</code> (body, 可选)	string	描述
<code>nodes</code> (body)	array	节点定义列表
<code>edges</code> (body)	array	节点间连线定义

1.3. GET /api/workflows/:id

获取单个工作流定义的完整信息，包含节点与连线的结构化数据。

1.4. PUT /api/workflows/:id

更新工作流定义（名称、描述、节点编排等）。

1.5. DELETE /api/workflows/:id

删除工作流定义及关联的所有执行记录。

2. 工作流控制

2.1. POST /api/workflows/:id/trigger

手动触发工作流执行。适用于测试或按需执行。

字段	类型	说明
<code>input</code> (body, 可选)	object	传入工作流的初始上下文数据
响应 <code>executionId</code>	string	本次执行的唯一 ID

2.2. PUT /api/workflows/:id/toggle

启用/停用工作流。停用后不响应任何触发（包括 Webhook 和定时触发）。

字段	类型	说明
<code>enabled</code> (body)	boolean	<code>true</code> 启用, <code>false</code> 停用

3. 执行记录

3.1. GET /api/workflows/:id/executions

列出指定工作流的执行历史。

查询参数	类型	说明
<code>page</code>	number	页码
<code>limit</code>	number	每页数量
<code>status</code>	string (可选)	按状态筛选: <code>running</code> / <code>success</code> / <code>failed</code>

响应字段	类型	说明
<code>id</code>	string	执行 ID
<code>status</code>	string	执行状态
<code>startedAt</code>	string	开始时间
<code>finishedAt</code>	string	结束时间

响应字段	类型	说明
<code>trigger</code>	string	触发方式: <code>manual</code> / <code>webhook</code> / <code>schedule</code>

3.2. GET /api/workflows/:id/executions/:executionId

获取单次执行的详细记录，包含每个节点的输入/输出、耗时及日志。

响应字段	类型	说明
<code>nodeResults</code>	array	各节点执行结果
<code>nodeResults[].nodeId</code>	string	节点 ID
<code>nodeResults[].status</code>	string	节点状态
<code>nodeResults[].input</code>	object	节点输入
<code>nodeResults[].output</code>	object	节点输出
<code>nodeResults[].duration</code>	number	执行耗时 (ms)
<code>nodeResults[].logs</code>	array	节点日志

3.3. POST /api/workflows/executions/batch-delete

批量删除执行记录。

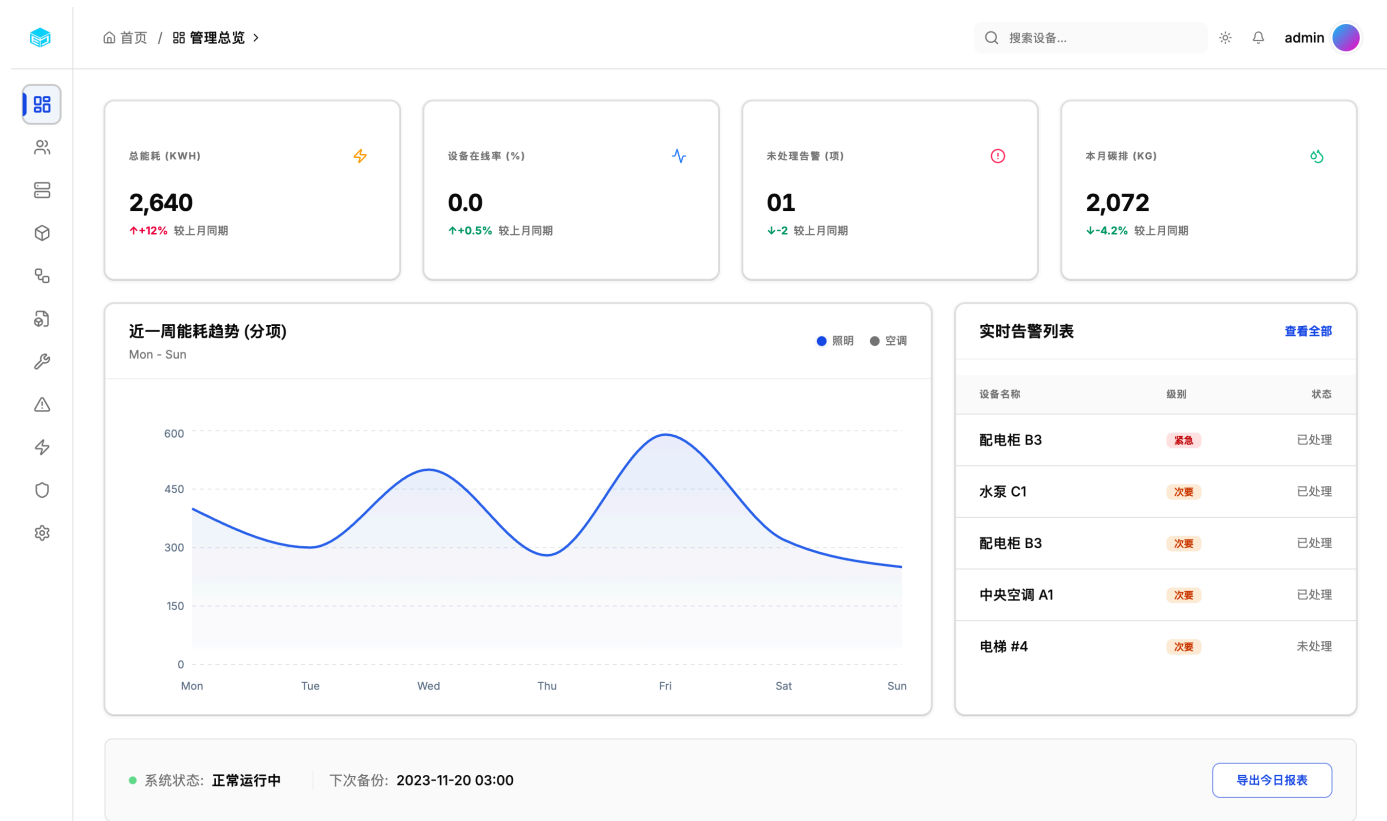
字段	类型	说明
<code>ids</code> (body)	array	要删除的执行记录 ID 列表
响应 <code>deletedCount</code>	number	实际删除数量

4. 插件节点

4.1. GET /api/nodes

177 / 257

响应字段	类型	说明
<code>nodes</code>	array	节点类型列表
<code>nodes[].type</code>	string	节点类型标识
<code>nodes[].name</code>	string	展示名称
<code>nodes[].category</code>	string	分类（如 <code>trigger</code> / <code>action</code> / <code>condition</code> ）
<code>nodes[].inputs</code>	array	输入接口定义
<code>nodes[].outputs</code>	array	输出接口定义



IoT 与实时事件 API

IoT 模块代理上游 IoT 网关的请求/响应，EcoCtrl 透明处理 Token 刷新（依赖 `iot_tokens` 表）。实时事件模块提供基于 SSE（Server-Sent Events）的推送订阅。

所有端点均需认证。

完整的 API 参考可在运行时通过 Swagger UI 查阅（本地开发：
<http://localhost:3000/documentation>，生产环境：[/documentation](#)）

1. IoT Token

1.1. GET /api/iot/token

返回当前缓存的 IoT 网关 Access Token。EcoCtrl 在后台自动维护 Token 的刷新与缓存。

响应字段	类型	说明
<code>token</code>	string	缓存的 IoT Access Token
<code>expiresAt</code>	string	过期时间

1.2. POST /api/iot/token

主动签发或刷新 IoT 网关 Token。

字段	类型	说明
<code>forceRefresh</code> (body, 可选)	boolean	是否强制刷新缓存
响应 <code>token</code>	string	新 Token

2. 点位数据

请求/响应保持上游 IoT 网关契约，EcoCtrl 透明转发。

2.1. POST /api/iot/codes/values

读取当前点位值。

字段	类型	说明
<code>codes</code> (body)	array	点位编码列表
响应	array	每个点位的当前值、时间戳、质量戳

2.2. POST /api/iot/codes/history

查询点位历史值。

字段	类型	说明
<code>codes</code> (body)	array	点位编码列表
<code>from</code> (body)	string (ISO 8601)	起始时间
<code>to</code> (body)	string (ISO 8601)	结束时间
<code>interval</code> (body, 可选)	string	聚合间隔
响应	array	时序数据

2.3. POST /api/iot/codes/set

向点位写入控制值。

字段	类型	说明
<code>code</code> (body)	string	点位编码
<code>value</code> (body)	any	要写入的值
响应 <code>success</code>	boolean	写入结果

2.4. POST /api/iot/codes/force-set

强制写入点位值（绕过安全检查或权限校验）。

字段	类型	说明
<code>code</code> (body)	string	点位编码
<code>value</code> (body)	any	要写入的值
<code>reason</code> (body, 可选)	string	强制写入原因
响应 <code>success</code>	boolean	写入结果

3. 报警

3.1. POST /api/iot/alarms

查询 IoT 报警历史。

字段	类型	说明
<code>from</code> (body)	string (ISO 8601)	起始时间
<code>to</code> (body)	string (ISO 8601)	结束时间
<code>level</code> (body, 可选)	string	报警级别筛选
<code>page</code> (body)	number	页码
<code>limit</code> (body)	number	每页数量

3.2. POST /api/iot/alarm-configs

查询或配置报警规则。

字段	类型	说明
<code>action</code> (body)	string	<code>query</code> / <code>create</code> / <code>update</code> / <code>delete</code>

字段	类型	说明
<code>config</code> (body, 可选)	object	报警规则配置

4. 实时事件 (SSE)

4.1. POST /api/events/token

获取 SSE 订阅所需的认证 Token。

响应字段	类型	说明
<code>token</code>	string	SSE 订阅 Token
<code>expiresAt</code>	string	过期时间

4.2. GET /api/events

建立 SSE (Server-Sent Events) 长连接，用于实时推送事件。

查询参数	类型	说明
<code>token</code>	query	从 /api/events/token 获取的订阅 Token
<code>types</code>	query (可选)	逗号分隔的事件类型筛选

事件类型包括：

事件	说明
<code>iot:value:change</code>	点位值实时变化
<code>iot:alarm:trigger</code>	报警触发
<code>iot:alarm:clear</code>	报警恢复
<code>workflow:execution:status</code>	workflow 执行状态变更

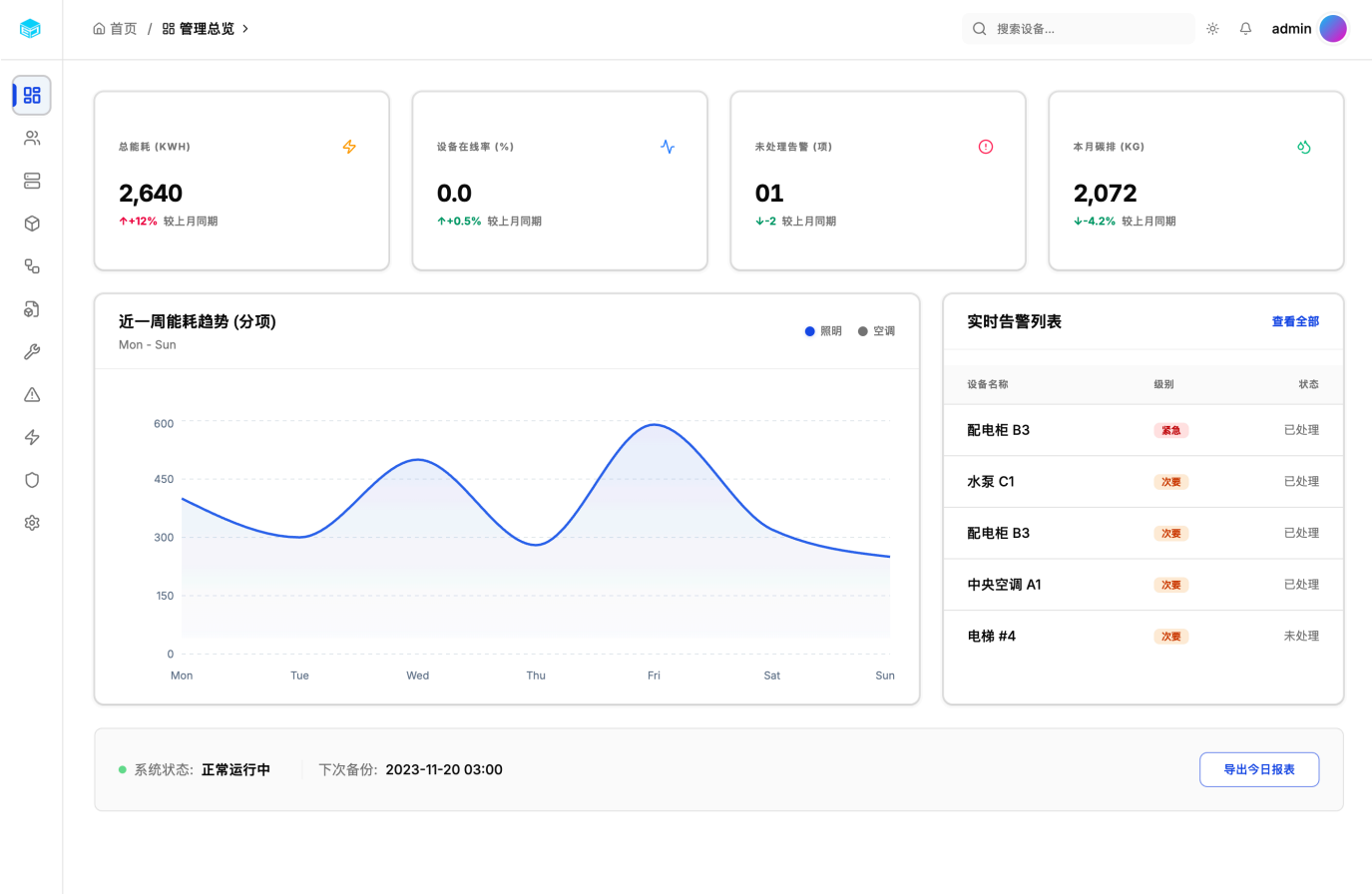
完整的 API 参考可在运行时通过 Swagger UI 查阅（本地开发：

<http://localhost:3000/documentation> ， 生产环境： [/documentation](#) ）

报表与导出 API

报表模块管理定时报表计划与模板配置，并提供基于队列的异步数据导出能力。所有端点均需认证。

注意：不提供直接的 `/instances` 端点；已生成的报表通过队列 Worker 输出获取。



查询参数	类型	说明
<code>limit</code>	number	每页数量
<code>enabled</code>	boolean (可选)	按启停状态筛选
响应	array	报表计划列表

1.2. POST /api/reports

创建定时报表计划。

字段	类型	说明
<code>name</code> (body)	string	计划名称
<code>templateId</code> (body)	number	关联的报表模板 ID
<code>cron</code> (body)	string	Cron 表达式
<code>enabled</code> (body)	boolean	是否立即启用
<code>recipients</code> (body, 可选)	array	邮件通知接收人列表
<code>format</code> (body)	string	输出格式: <code>pdf</code> / <code>csv</code> / <code>excel</code>

1.3. PUT /api/reports/:id

更新报表计划（名称、Cron 表达式、模板等）。

1.4. DELETE /api/reports/:id

删除报表计划。

2. 报表模板

2.1. GET /api/reports/templates

列出所有可用的报表模板。模板定义了报表的布局、指标和数据源。

响应字段	类型	说明
<code>templates</code>	array	模板列表
<code>templates[].id</code>	number	模板 ID
<code>templates[].name</code>	string	模板名称
<code>templates[].description</code>	string	模板说明
<code>templates[].category</code>	string	模板分类

2.2. POST /api/reports/templates

创建报表模板。

字段	类型	说明
<code>name</code> (body)	string	模板名称
<code>description</code> (body, 可选)	string	描述
<code>config</code> (body)	object	模板布局与指标配置

3. 数据导出

3.1. POST /api/reports/export

触发异步 CSV 导出任务。导出结果通过队列 Worker 处理，完成后可在文件列表中获取。

字段	类型	说明
<code>type</code> (body)	string	导出数据类型: <code>energy</code> / <code>faults</code> / <code>alarms</code>
<code>from</code> (body)	string (ISO 8601)	数据起始时间
<code>to</code> (body)	string (ISO 8601)	数据结束时间

字段	类型	说明
<code>filters</code> (body, 可选)	object	额外筛选条件
响应 <code>jobId</code>	string	队列任务 ID，可用于追踪导出进度

🏠 首页 / 设备管理总览 >

🔍 搜索设备...

⚙️ 🔔 admin

🔧

👤

📁

🏠

🔧

🔧

🔧

⚠️

⚡

🛡️

⚙️

总能耗 (KWH)

⚡

2,640

↑+12% 较上月同期

设备在线率 (%)

📶

0.0

↑+0.5% 较上月同期

未处理告警 (项)

🔴

01

↓-2 较上月同期

本月减排 (KG)

🌱

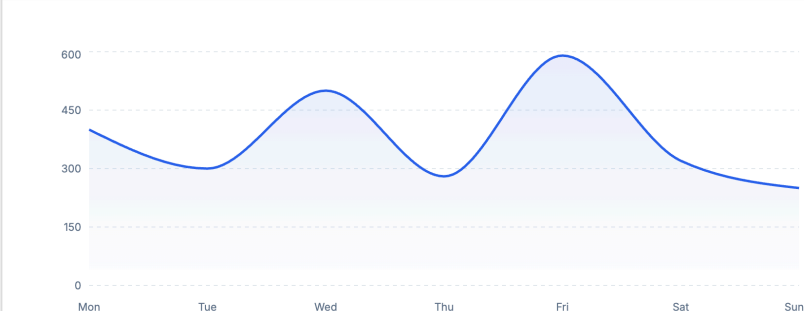
2,072

↓-4.2% 较上月同期

近一周能耗趋势 (分项)

Mon - Sun

● 照明 ● 空调



实时告警列表

查看全部

设备名称	级别	状态
配电柜 B3	紧急	已处理
水泵 C1	次要	已处理
配电柜 B3	次要	已处理
中央空调 A1	次要	已处理
电梯 #4	次要	未处理

● 系统状态: 正常运行中

下次备份: 2023-11-20 03:00

导出今日报表

https://docs.godot.qzz.io/reference/api/reports-export

186 / 257

照明控制 API

照明控制模块提供对建筑/区域照明设备的远程管理：查询状态、开关切换和批量控制。对接底层 IoT 网关协议，支持 Mock 模式用于开发调试。

所有端点均需认证。

完整的 API 参考可在运行时通过 Swagger UI 查阅（本地开发：
<http://localhost:3000/documentation>，生产环境：[/documentation](#)）

1. 查询照明状态

1.1. GET /api/control/lighting/:labelId/status

查询指定照明标签组的状态。

参数	类型	说明
<code>:labelId</code> (path)	string	照明标签标识（如区域编码）

响应字段	类型	说明
<code>labelId</code>	string	标签 ID
<code>on</code>	boolean	是否开启（false=关，true=开）
<code>brightness</code>	number (可选)	亮度百分比（0-100）
<code>power</code>	number (可选)	当前功率 (W)
<code>updatedAt</code>	string	状态更新时间

2. 开关切换

2.1. POST /api/control/lighting/:labelId/toggle

切换指定照明标签组的开关状态。

参数	类型	说明
:labelId (path)	string	照明标签标识
on (body)	boolean	true 开灯， false 关灯； half 半开（仅 toggle 端支持）

响应字段	类型	说明
success	boolean	操作是否成功
labelId	string	标签 ID
on	boolean	切换后状态

3. 批量控制

3.1. POST /api/control/lighting/:labelId/batch

向指定标签组下发批量控制指令。适用于场景联动（如"一键关灯"）。

参数	类型	说明
:labelId (path)	string	照明标签标识
action (body)	string	控制动作： on / off / dim / scene
value (body, 可选)	any	动作参数（如 dim 时的亮度值， scene 时的场景 ID）
devices (body, 可选)	array	设备子集，不传则控制组内全部设备

响应字段	类型	说明
success	boolean	整体操作结果
results	array	每个设备的执行结果

响应字段	类型	说明
<code>results[].deviceId</code>	string	设备 ID
<code>results[].success</code>	boolean	单设备执行结果

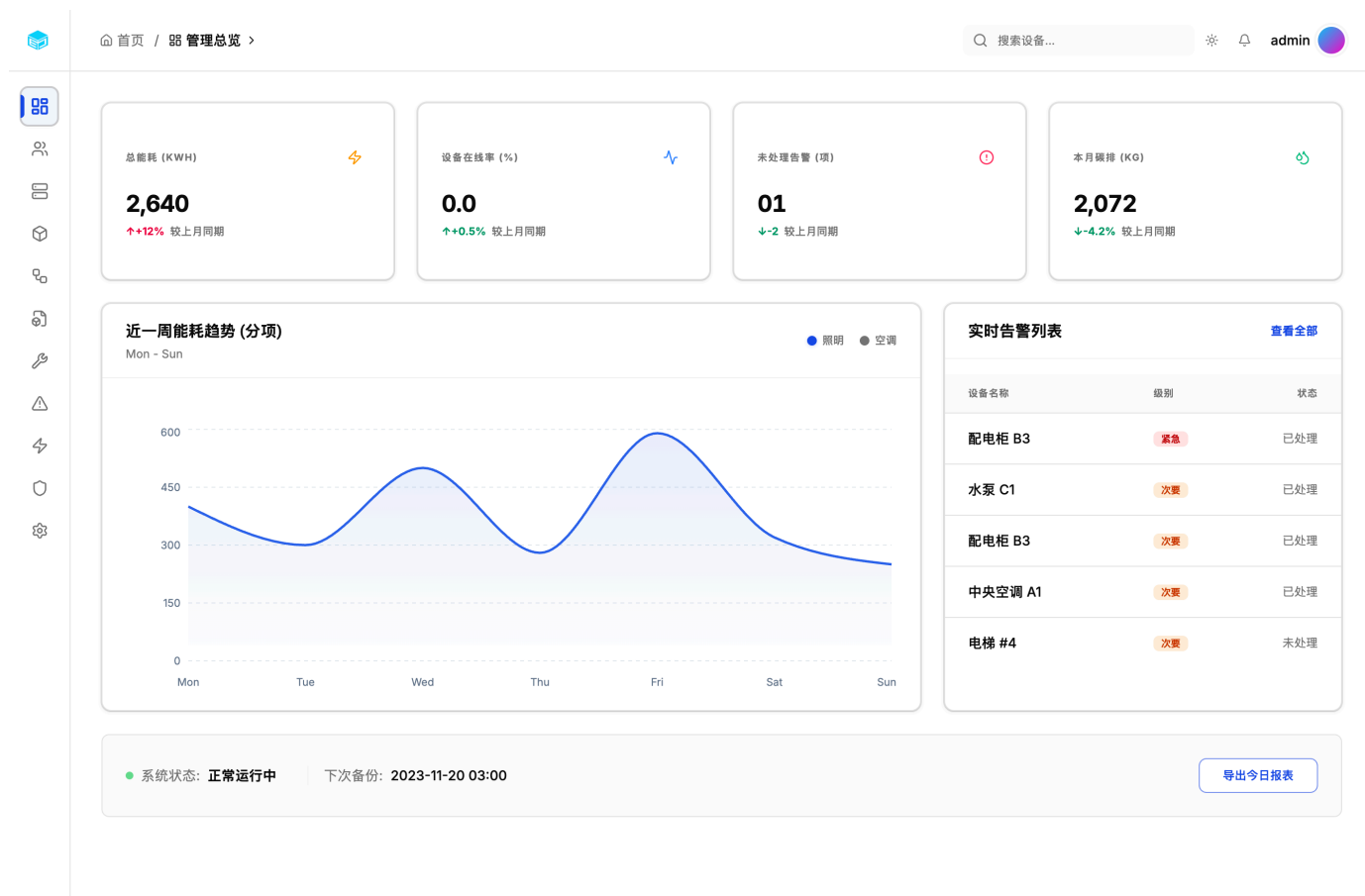
完整的 API 参考可在运行时通过 Swagger UI 查阅（本地开发：
<http://localhost:3000/documentation>，生产环境：</documentation>）

WebTalk 代理 API

WebTalk 是 EcoCtrl 内置的 WebSocket 反向代理模块，用于将第三方 Web 应用嵌入到 Admin 管理后台。通过统一认证中转和 CORS 策略白名单，实现安全可控的第三方系统集成。

所有端点均需认证，除非另有说明。

相关子系统文档：[WebTalk 反向代理设计](#)



1. 反向代理 (通配符路由)

WebTalk 使用一条通配符路由 `/*` 捕获并转发所有匹配的请求到上游目标服务。

1.1. `/*` (通配符代理)

属性	说明
方法	全部 HTTP 方法（GET / POST / PUT / DELETE / PATCH / ...）
路径	通配符 <code>/*</code> ，匹配所有子路径
认证	需要通过 EcoCtrl 认证（Bearer Token）
上游目标	从 WebTalk 配置中读取的目标第三方服务 URL

行为说明:

1. 请求到达 EcoCtrl 后端
2. 验证请求中的 `Authorization: Bearer <token>`
3. 从 WebTalk 配置中查找路由映射，确定上游目标 URL
4. 请求头转发：自动传递认证信息，可将 EcoCtrl 用户身份注入到上游请求中
5. 响应透传：上游响应直接返回给客户端（状态码、Headers、Body 均保持原样）

2. 认证转发机制

WebTalk 在反向代理请求时，将 EcoCtrl 的认证身份转发给上游第三方系统：

转发方式	说明
Header 注入	在转发请求头中注入 <code>X-EcoCtrl-User-Id</code> 、 <code>X-EcoCtrl-User-Name</code> 等标识
Token 透传	可选将原始 Bearer Token 透传给上游
免登录嵌入	用户无需在第三方系统中再次登录

配置要求：上游第三方系统需信任 EcoCtrl 的 Header 注入，通常需要在白名单网络中部署。

3. iframe CORS 策略

为支持在 Admin 后台中以 iframe 嵌入第三方 Web 应用，WebTalk 配置了以下策略：

策略	说明
Content-Security-Policy: frame-ancestors	仅允许 EcoCtrl Admin 域名作为父页面
X-Frame-Options: ALLOW-FROM	兼容旧版浏览器的帧来源限制
CORS 白名单	预配置允许跨域访问的来源域名列表

配置字段：

字段	类型	说明
upstreamUrl	string	上游第三方服务的 URL
allowedOrigins	array	CORS 白名单
injectHeaders	object	转发时注入的头部
wsSupport	boolean	是否启用 WebSocket 代理支持

🏠 首页 / 管理总览 >

🔍 搜索设备...

⚙️ 🔔 admin

🔧

👤

📅

🏠

🔗

🔍

🔧

⚠️

⚡

🛡️

⚙️

总能耗 (KWH)

2,640

↑ +12% 较上月同期

设备在线率 (%)

0.0

↑ +0.5% 较上月同期

未处理告警 (项)

01

↓ -2 较上月同期

本月碳排 (KG)

2,072

↓ -4.2% 较上月同期

近一周能耗趋势 (分项)

Mon - Sun

● 照明

● 空调

实时告警列表

查看全部

设备名称	级别	状态
配电柜 B3	紧急	已处理
水泵 C1	次要	已处理
配电柜 B3	次要	已处理
中央空调 A1	次要	已处理
电梯 #4	次要	未处理

● 系统状态: 正常运行中

下次备份: 2023-11-20 03:00

导出今日报表

https://docs.godot.qzz.io/reference/api/webtalk

192 / 257

核心实体参考

EcoCtrl 在 `packages/server/src/schemas/` 下定义 28 个 Drizzle ORM 表。下面对核心表的用途和关键字段进行说明。

1. 用户与权限

1.1. users

字段	类型	说明
id	uuid PK	用户唯一标识
username	varchar(50) unique	登录用户名
email	varchar(255)	邮箱地址
password	varchar(255)	bcrypt 哈希密码
role	varchar(20)	admin / manager / operator / viewer
avatarUrl	text	头像 URL

1.2. refresh_tokens

字段	类型	说明
id	serial PK	
userId	uuid FK → users.id	关联用户
tokenHash	varchar(64) unique	Refresh Token 的 SHA-256 哈希
expiresAt	timestamp	过期时间（7天）

1.3. oauth_accounts

字段	类型	说明
id	serial PK	
userId	uuid FK → users.id	关联用户
provider	varchar(20)	wechat / feishu
providerUserId	varchar(255)	第三方平台用户 ID

2. 模型与对象

2.1. models

字段	类型	说明
id	uuid PK	模型唯一标识
name	varchar(100)	模型名称
version	varchar(20)	版本号
code	varchar(50)	模型编码
filePath	text	3D 模型文件路径

2.2. objects

字段	类型	说明
id	uuid PK	
modelId	uuid FK → models.id	关联模型
code	varchar(50)	对象编码
name	varchar(100)	对象名称

2.3. points

字段	类型	说明
id	uuid PK	
objectId	uuid FK → objects.id	关联对象
type	varchar(20)	sensor / actuator / meter
code	varchar(50)	点位编码
name	varchar(100)	点位名称
zone	varchar(50)	分区
system	varchar(50)	所属系统

3. 工作流

3.1. workflows

字段	类型	说明
id	uuid PK	
name	varchar(200)	工作流名称
dsl	jsonb	工作流 DSL 定义
triggerType	varchar(20)	manual / schedule / webhook / event
enabled	boolean	启用状态
version	int	版本号

3.2. workflow_executions

字段	类型	说明
id	uuid PK	

字段	类型	说明
workflowId	uuid FK → workflows.id	
status	varchar(20)	running / completed / failed
output	jsonb	执行输出
startedAt	timestamp	开始时间
duration	int	耗时 (ms)

4. 能耗与碳排放

4.1. energy_readings

字段	类型	说明
id	serial PK	
areaId	int FK → energy_areas.id	区域
value	numeric	能耗值 (kWh)
powerFactor	numeric	功率因素
loadRate	numeric	负载率
recordedAt	timestamp	记录时间

4.2. energy_areas

字段	类型	说明
id	serial PK	
name	varchar(100)	区域名称
targetValue	numeric	目标能耗值

字段	类型	说明
basePowerFactor	numeric	基准功率因素

4.3. carbon_factors

字段	类型	说明
id	serial PK	
nodeId	int FK → carbon_factor_nodes.id	分类节点
name	varchar(200)	因子名称
value	numeric	因子值
unit	varchar(50)	单位（如 kgCO ₂ /kWh）

5. 故障与维护

5.1. faults

字段	类型	说明
id	serial PK	
deviceName	varchar(100)	设备名称
level	varchar(10)	critical / warning / info
status	varchar(20)	pending / processing / resolved
triggeredAt	timestamp	触发时间

5.2. fault_stats

字段	类型	说明
id	serial PK	
totalFaults	int	故障总数
mttr	int	平均修复时间（秒）
period	varchar(10)	统计周期

5.3. maintenance

字段	类型	说明
id	serial PK	
title	varchar(200)	维护标题
scheduledAt	date	计划日期
status	varchar(20)	pending / completed

6. 报表

6.1. report_plans

字段	类型	说明
id	serial PK	
name	varchar(200)	计划名称
recipients	text	收件人邮箱（CSV）
frequency	varchar(20)	daily / weekly / monthly
enabled	boolean	启用状态

7. IoT

7.1. `iot_tokens`

字段	类型	说明
id	serial PK	
accessToken	text	IoT 网关访问令牌
refreshToken	text	IoT 网关刷新令牌
expiresAt	bigint	绝对过期时间 (Unix ms)

8. 系统配置

8.1. `platform_configs`

键值对存储平台配置 (SMTP、备份设置等)

8.2. `dashboard_widgets`

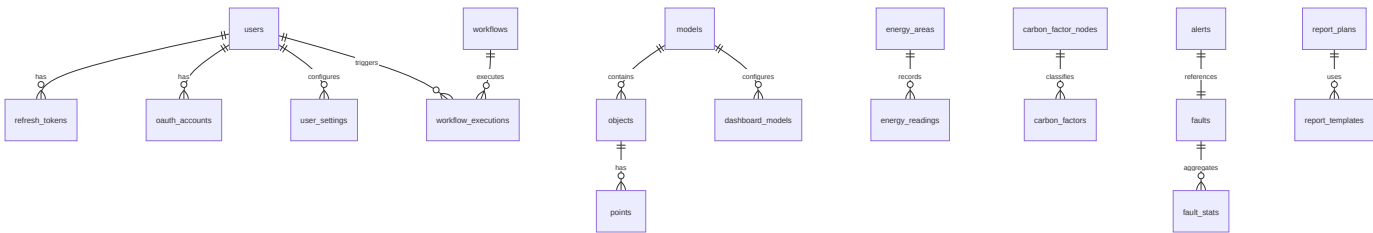
字段	类型	说明
id	serial PK	
type	varchar(20)	widget 类型
dataType	varchar(20)	数据源类型
dataJson	jsonb	配置数据
layoutX/Y/W/H	int	Bento Grid 布局坐标

8.3. `dashboard_models`

字段	类型	说明
id	serial PK	
modelId	uuid FK → models.id	关联模型
config	jsonb	场景配置（相机、灯光、热点）

实体关系图 (ERD)

1. 核心表关系



2. 键的外键关系

源表	外键	目标表	级联删除
refresh_tokens	userId	users.id	CASCADE
oauth_accounts	userId	users.id	CASCADE
objects	modelId	models.id	CASCADE
points	objectId	objects.id	CASCADE
energy_readings	areaId	energy_areas.id	RESTRICT
carbon_factors	nodeId	carbon_factor_nodes.id	CASCADE
workflow_executions	workflowId	workflows.id	CASCADE
dashboard_models	modelId	models.id	CASCADE

3. 索引策略

表	索引字段	目的
refresh_tokens	tokenHash (unique)	快速查找 Refresh Token
refresh_tokens	userId	按用户查询活跃会话
users	username (unique), email	登录加速
energy_readings	(arealId, recordedAt)	区域能耗时序查询
faults	status, level	告警列表快速筛选
workflow_executions	workflowId, startedAt	执行记录查询排序

系统要求与前提条件

本文介绍部署 EcoCtrl 所需的软件版本、端口分配和最低系统资源建议。

1. 软件版本要求

组件	最低版本	说明
Docker	24.0+	生产部署推荐 Docker Compose v2
Node.js	24.x	源码构建部署时需要
pnpm	10.33.1+	源码构建时需要（corepack 可自动管理版本）
PostgreSQL	16.x	生产环境推荐使用托管数据库

1.1. 验证安装

```
docker --version      # Docker 24.0+
node --version        # Node.js 24.x
pnpm --version        # pnpm 10.33.1+
psql --version        # PostgreSQL 16.x
```

bash

2. 端口分配

以下端口在部署主机上必须可用：

端口	服务	说明
3000	REST API	Fastify 服务端（Swagger UI 同端口）
4173	Admin 后台	管理面板 SPA（Caddy 代理）
8081	Web 门户	公共 3D 驾驶舱 SPA（Caddy 代理）

端口	服务	说明
5432	PostgreSQL	数据库（仅容器内部暴露）
9000	MinIO API	对象存储 API 端口
9001	MinIO 控制台	对象存储管理控制台

使用 `docker compose -f compose.yaml up` 启动后，如需调整端口映射，请编辑 `docker/.env.local` 中的 `SERVER_PORT`、`WEB_PORT`、`ADMIN_PORT` 等变量。

3. 网络访问

生产环境需要以下出站连接：

目标	用途
ghcr.io	拉取预构建 Docker 镜像
hub.docker.com	拉取 PostgreSQL、MinIO 镜像
api.openweathermap.org	天气数据（选用时）
api.openai.com / 自定义 AI	AI 助手（选用时）
SMTP 服务器	邮件发送（选用时）
OAuth Provider	微信/飞书登录（选用时）

离线环境：使用[离线部署包](#)可跳过 GHCR 和 Docker Hub 的直接访问。

4. 系统资源建议

规模	CPU	内存	磁盘空间	适用场景
小型	2 核	4 GB	20 GB SSD	测试评估、小型建筑监控

规模	CPU	内存	磁盘空间	适用场景
中型	4 核	8 GB	50 GB SSD	日常生产（少于 500 点位）
大型	8 核	16 GB	100 GB+ SSD	大型设施（1000+ 点位、3D 模型存储）

磁盘空间主要消耗于：

- PostgreSQL 数据目录：历史能耗数据随运行时间增长
- MinIO 对象存储：上传的 3D 模型、插件和附件文件
- 服务端日志：pino JSON 日志文件（按 LOG_MAX_DAYS 配置自动轮转）

5. 操作系统兼容性

EcoCtrl 的 Docker 镜像基于 `node:24-alpine` 和 `postgres:16-alpine`，可在任何支持 Docker 的操作系统上运行，包括：

- Linux (Ubuntu 22.04+, Debian 12, CentOS Stream 9, 等)
- macOS (Docker Desktop)
- Windows (Docker Desktop / WSL2, 仅限测试环境)

生产环境推荐使用 Linux (Ubuntu 24.04 LTS)。

Docker 部署指南

EcoCtrl 以 Docker 镜像形式发布到 GitHub Container Registry (GHCR)。推荐使用 Docker Compose 部署全部服务。

1. 部署方案

提供三份 Compose 文件，分别对应不同场景：

文件	用途
<code>compose.yaml</code>	从 GHCR 拉取预构建镜像（默认，推荐生产使用）
<code>compose.build.yaml</code>	从本地 Dockerfile 构建所有镜像（自定义代码时）
<code>compose.dev.yaml</code>	开发叠加层，挂载本地源码并启用热重载

三份文件均位于 `docker/` 目录，由 `scripts/generate-compose.py` 从 `compose.base.yaml` 模板生成，请勿直接编辑生成的 yaml 文件。

2. 前置准备

```
git clone https://github.com/hyooeewee/ecoctrl.git
cd ecoctrl/docker
cp .env.example .env.local
$EDITOR .env.local      # 至少设置 JWT_SECRET（必填）
```

bash

所有 Compose 场景都使用 `docker/.env.local` 作为共享环境变量文件。

3. 场景一：预构建镜像部署（推荐）

一条命令启动全部服务：

```
docker compose up -d
```

bash

服务列表：

服务	端口	地址
Web 门户	8081	<a href="http://<host>:8081">http://<host>:8081
Admin 后台	4173	<a href="http://<host>:4173">http://<host>:4173
REST API	3000	<a href="http://<host>:3000">http://<host>:3000
Swagger UI	3000	<a href="http://<host>:3000/documentation">http://<host>:3000/documentation
PostgreSQL	5432	容器内部
MinIO	9000	对象存储（容器内部）

3.1. 升级

```
docker compose pull
docker compose up -d
docker compose exec server npx drizzle-kit migrate # 运行数据库迁移
```

bash

4. 场景二：源码构建部署

从本地源码构建镜像，适合需要自定义代码或审查构建过程：

```
cd docker
cp .env.example .env.local
# 编辑 .env.local 填入必要配置

# 使用串行构建脚本（推荐，避免并发构建耗尽资源）
./build.sh
```

bash

```
# 或手动构建
# docker compose -f compose.build.yaml build
# docker compose -f compose.build.yaml up -d
```

注意：并发构建多个 Node 镜像（server + web + admin）可能因内存或网络资源不足而失败。串行构建脚本 `build.sh` 每次构建一个镜像，是最可靠的方式。

5. 场景三：开发调试

挂载本地源码并启用热更新，适合日常开发：

```
cd docker
docker compose -f compose.build.yaml -f compose.dev.yaml up --build
```

bash

开发模式下：

- API 服务运行在端口 3001（避免与本地开发冲突）
- Admin 运行在端口 5173（Vite dev server）
- Web 运行在端口 8080（Vite dev server）
- 源码修改后自动重载

6. Caddyfile 配置

前端 SPA 容器（admin/web）使用 Caddy 作为轻量级 Web 服务器和反向代理。每个容器的 `Caddyfile` 内容相同：

```
:80 {
    handle_path /api/* {
        rewrite * {$API_PREFIX:/api}{uri}
        reverse_proxy {$API_BASE_URL:http://localhost:3000}
    }

    handle_path /static/* {
```

caddyfile

```
rewrite * {{$STATIC_PREFIX:/static}{uri}
reverse_proxy {{$API_BASE_URL:http://localhost:3000}}
}

handle {
  root * /usr/share/caddy
  try_files {path} /index.html
  file_server
}

encode gzip zstd

@static {
  path *.js *.css *.png *.jpg *.jpeg *.gif *.ico *.svg *.woff *.woff2 *.t
}

header @static Cache-Control "public, max-age=31536000, immutable"

header X-Frame-Options "SAMEORIGIN"
header X-Content-Type-Options "nosniff"
header Referrer-Policy "strict-origin-when-cross-origin"
}
```

关键规则：

规则	说明
<code>/api/*</code> 代理	将所有 <code>/api</code> 请求转发到后端服务器端口
<code>/static/*</code> 代理	将静态资源请求（3D 模型等）转发到后端
SPA fallback	<code>try_files {path} /index.html</code> 确保客户端路由正常工作
静态资源缓存	前端 bundle 的 JS/CSS/字体文件设置一年强缓存
安全响应头	<code>X-Frame-Options</code> 、 <code>X-Content-Type-Options</code> 、 <code>Referrer-Policy</code>

7. 环境文件配置

部署时通过 `docker/.env.local` 统一配置所有环境变量。

bash

```
cp .env.example .env.local
```

关键变量（完整参考见[环境变量参考](#)）：

变量	说明	必填
JWT_SECRET	JWT 签名密钥	是
DATABASE_URL	外部数据库连接（可选）	否
INIT_ADMIN_PASSWORD	初始管理员密码	否
CORS_ORIGIN	允许的跨域来源	否
SMTP_HOST / SMTP_PASS	邮件服务器	否

`DATABASE_URL` 留空时使用 Compose 内置的 PostgreSQL。

8. 默认管理员账号

首次启动时，服务端会自动创建初始管理员账号：

字段	默认值
用户名	<code>admin</code>
密码	由 <code>INIT_ADMIN_PASSWORD</code> 指定，未设置则随机生成并输出到服务端日志
邮箱	<code>admin@example.com</code>

生产环境务必设置 `INIT_ADMIN_PASSWORD` 为一个强密码，或启动后立即在 Admin 后台修改。
如果不设置，请查看 `docker compose logs server` 获取随机生成的管理员密码。

9. 停止服务

bash

```
docker compose down          # 保留数据卷
docker compose down -v       # 同时清空 PostgreSQL 和 MinIO 数据卷（危险！将丢失所有数据）
```

10. TLS 与 HTTPS

Compose 配置的是明文 HTTP。生产环境请在容器之外终结 TLS，可选方案：

- 反向代理（推荐）：使用 Caddy、Nginx、Traefik 等在前置层提供 HTTPS
- 云平台负载均衡：AWS ALB、Cloudflare、阿里云 SLB 等在入口层终结 TLS
- 修改 Caddyfile：直接在每个 SPA 容器中配置 Caddy 自动 HTTPS（需要修改 Dockerfile）

10.1. 前置反向代理示例（Nginx）

nginx

```
server {
    listen 443 ssl;
    server_name admin.yourdomain.com;

    ssl_certificate /etc/letsencrypt/live/yourdomain.com/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/yourdomain.com/privkey.pem;

    location / {
        proxy_pass http://127.0.0.1:4173;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

11. 离线部署

如果目标机器无法访问外部仓库：

1. 在有网络的机器上下载离线部署包

2. 传输到目标机器

```
unzip ecoctrl.zip
cd ecoctrl-docker
sh migrate-images.sh compose.yaml --load ecoctrl-images.tar
docker compose up -d
```

bash

离线包内已包含：

- 预拉取的所有 Docker 镜像（含 PostgreSQL、MinIO、Server、Web、Admin）
- `compose.yaml` 与加载脚本
- 无需连接外部仓库

12. 健康检查

每个容器都配置了健康检查：

容器	检查方式	间隔
postgres	<code>pg_isready -U ecoctrl -d ecoctrl</code>	5s
minio	<code>curl http://localhost:9000/minio/health/live</code>	5s
server	<code>wget -q0- http://127.0.0.1:3000/health</code>	10s

`depends_on` 配合 `condition: service_healthy` 确保服务按正确顺序启动。

环境变量参考

环境变量通过 `docker/.env.local` 配置。所有变量均由服务端的 Zod schema 校验（`packages/server/src/lib/env.ts`），仅有明确定义的变量才会被读取并生效。

1. 核心服务

变量	默认值	必填	说明
<code>NODE_ENV</code>	<code>development</code>	否	运行模式 <code>production</code> / <code>development</code>
<code>PORT</code>	<code>3000</code>	否	服务端 HTTP 监听端口
<code>HOST</code>	<code>0.0.0.0</code>	否	服务端监听地址
<code>JWT_SECRET</code>	—	是	JWT 签名密钥，生产环境需设置为强随机字符串
<code>CORS_ORIGIN</code>	—	否	允许跨域来源，多个用逗号分隔。生产环境必须限制为真实域名
<code>DATABASE_URL</code>	—	是	PostgreSQL 连接字符串，格式 <code>postgresql://user:password@host:5432/db</code>

2. 认证与初始管理员

变量	默认值	必填	说明
<code>INIT_ADMIN_PASSWORD</code>	—	否	首次启动时创建的超级管理员密码。未设置则随机生成并输出到日志

3. 日志

变量	默认值	必填	说明
LOG_LEVEL	info	否	日志级别 <code>debug</code> / <code>info</code> / <code>warn</code> / <code>error</code>
LOG_DESTINATION	stdout	否	输出目标 <code>stdout</code> / <code>file</code> / <code>both</code>
LOG_DIR	./logs	否	日志文件目录（ <code>file</code> 或 <code>both</code> 时有效）
LOG_PRETTY	true	否	是否使用美化输出（仅开发模式有效）
LOG_ROTATE_INTERVAL	1d	否	日志轮转间隔（ <code>1d</code> / <code>7d</code> / <code>1h</code> ）
LOG_MAX_DAYS	30	否	日志保留天数

4. AI 服务

变量	默认值	必填	说明
AI_PROVIDER	openai	否	AI 提供商， <code>anthropic</code> / <code>openai</code>
AI_API_KEY	—	是（使用 AI 功能时）	API 密钥
AI_BASE_URL	https://openrouter.ai/api/v1	否	自定义 API 端点，可切换为兼容 OpenAI 的第三方服务
AI_MODEL	nvidia/nemotron-3-super-120b-a12b:free	否	模型标识符

AI 功能为可选。未配置时，AI 助手功能在前端自动隐藏。

5. SMTP 邮件服务

变量	默认值	必填	说明
SMTP_HOST	smtp.163.com	否	SMTP 服务器地址
SMTP_PORT	465	否	SMTP 端口 (SSL: 465, TLS: 587)
SMTP_USER	—	否	SMTP 用户名
SMTP_PASS	—	否	SMTP 密码或授权码
SMTP_SECURE	true	否	是否使用 SSL 连接 (true / false)

详情见 [SMTP 邮件服务配置](#)。

6. OAuth 登录

变量	默认值	必填	说明
WECHAT_APP_ID	—	否	微信开放平台 AppID
WECHAT_APP_SECRET	—	否	微信开放平台 AppSecret
FEISHU_APP_ID	—	否	飞书开放平台 AppID
FEISHU_APP_SECRET	—	否	飞书开放平台 AppSecret

OAuth 功能可选。未配置时，登录界面对应的 OAuth 按钮自动隐藏。

7. 天气服务

变量	默认值	必填	说明
OPENWEATHER_API_KEY	—	否	OpenWeather API Key
WEATHER_LAT	39.9042	否	默认纬度 (北京)
WEATHER_LNG	116.4074	否	默认经度 (北京)

变量	默认值	必填	说明
<code>WEATHER_LOCATION</code>	<code>Beijing</code>	否	显示的地点名称

天气功能可选。未配置时，仪表盘天气组件自动隐藏。

8. IoT 集成

变量	默认值	必填	说明
<code>BASE_URL</code>	—	否	IoT 网关基础 URL（BACnet 网关/第三方网关地址）
<code>APP_ID</code>	—	否	IoT 应用 ID（用于自动刷新 IoT 令牌）

IoT 集成功能可选。未配置时，照明控制等功能会降级到 Mock 模式运行。

9. 对象存储 (S3)

变量	默认值	必填	说明
<code>STORAGE_PROVIDER</code>	<code>local</code>	否	存储后端 <code>minio</code> / <code>local</code> （local 仅用于开发）
<code>S3_ENDPOINT</code>	—	否	S3 兼容端点地址
<code>S3_REGION</code>	<code>us-east-1</code>	否	S3 区域
<code>S3_ACCESS_KEY</code>	—	否	S3 访问密钥
<code>S3_SECRET_KEY</code>	—	否	S3 秘密密钥
<code>S3_BUCKET_FILES</code>	<code>ecoctrl-files</code>	否	文件附件存储桶
<code>S3_BUCKET_MODELS</code>	<code>ecoctrl-models</code>	否	3D 模型存储桶

变量	默认值	必填	说明
S3_BUCKET_NODES	ecoctrl-nodes	否	workflow节点插件存储桶
S3_BUCKET_PETS	ecoctrl-pets	否	AI 宠物插件存储桶
S3_FORCE_PATH_STYLE	false	否	是否强制使用路径风格（MinIO 需设为 true ）
S3_PUBLIC_ENDPOINT	—	否	公开访问的 S3 端点（若需要客户端直连）

STORAGE_PROVIDER=local 将文件存储在本地磁盘的 uploads/ 目录。 minio 模式需要运行 MinIO 容器并配置 S3 环境变量。

10. Docker Compose 配置

以下变量仅用于 Docker Compose 部署配置，不会被服务端代码直接读取：

变量	默认值	说明
POSTGRES_USER	ecoctrl	PostgreSQL 用户名
POSTGRES_PASSWORD	—	PostgreSQL 密码
POSTGRES_DB	ecoctrl	PostgreSQL 数据库名
POSTGRES_PORT	5432	PostgreSQL 端口
MINIO_ACCESS_KEY	ecoctrl	MinIO 访问密钥
MINIO_SECRET_KEY	—	MinIO 秘密密钥
MINIO_API_PORT	9000	MinIO API 端口
MINIO_CONSOLE_PORT	9001	MinIO 管理控制台端口
SERVER_PORT	3000	服务端端口映射
SERVER_HOST	0.0.0.0	服务端监听地址

变量	默认值	说明
WEB_PORT	8081	Web 门户端口映射
ADMIN_PORT	4173	Admin 后台端口映射
API_PREFIX	/api	API 路径前缀
STATIC_PREFIX	/static	静态资源路径前缀

11. Docker Compose 环境变量传递架构



注意：服务端会通过 `dotenv` 加载 `packages/server/.env.local` 文件，但主配置来源是 `compose.yaml` 中的 `env_file: .env.local`。重复定义时，容器的环境变量优先级更高。

12. 配置验证

在 Admin 后台的 系统配置 页面可以查看已加载的配置状态（部分变量不会暴露到前端）。服务端启动时如果缺少必填变量（如 `JWT_SECRET`），将立即抛出错误并退出。

SMTP 邮件服务配置

SMTP 配置使 EcoCtrl 能够发送注册验证码、密码重置邮件和系统通知。如不使用 SMTP，注册和密码重置功能将无法正常工作。

1. 配置方式

SMTP 配置支持两种途径（优先级从高到低）：

- 1. 环境变量（推荐）：在 `docker/.env.local` 中设置 `SMTP_*` 变量，服务端启动时自动同步到 `platform_configs` 表
- 2. Admin 后台：登录超级管理员账号，进入 系统配置 → SMTP 设置 在线修改

两种方式效果相同，环境变量方式更适合持续部署。

2. 参数说明

参数	环境变量	说明
SMTP 服务器	<code>SMTP_HOST</code>	SMTP 服务器地址，如 <code>smtp.163.com</code> 、 <code>smtp.gmail.com</code>
SMTP 端口	<code>SMTP_PORT</code>	连接端口（SSL: <code>465</code> ， TLS/STARTTLS: <code>587</code> ， 无加密: <code>25</code> ）
用户名	<code>SMTP_USER</code>	SMTP 认证用户名，通常为邮箱地址
密码/授权码	<code>SMTP_PASS</code>	SMTP 密码或第三方授权码（如 163 邮箱、QQ 邮箱需使用授权码）
SSL 模式	<code>SMTP_SECURE</code>	<code>true</code> : 使用 SSL（端口 465）； <code>false</code> : 使用 STARTTLS（端口 587）

3. SSL 与 TLS 选择

模式	端口	SMTP_SECURE	说明
SSL	465	true	从连接开始就使用加密通道，推荐、兼容性最好
STARTTLS	587	false	明文连接后升级为加密，部分企业邮件服务器使用
无加密	25	false	明文传输，仅限内网或测试环境

大多数公有邮件服务（163、QQ、Gmail）推荐使用 SSL 端口 465。

4. 常用邮件服务提供商配置

4.1. 网易 163 邮箱

SMTP_HOST=smtp.163.com

SMTP_PORT=465

SMTP_USER=yourname@163.com

SMTP_PASS=<授权码> # 非登录密码，需在 163 设置中开启 SMTP 并获取授权码

SMTP_SECURE=true

env

4.2. QQ 邮箱

SMTP_HOST=smtp.qq.com

SMTP_PORT=465

SMTP_USER=yourname@qq.com

SMTP_PASS=<授权码> # QQ 邮箱设置 → 账户 → 生成授权码

SMTP_SECURE=true

env

4.3. Gmail

SMTP_HOST=smtp.gmail.com

SMTP_PORT=587

SMTP_USER=yourname@gmail.com

env

```
SMTP_PASS=<应用专用密码>    # 需开启两步验证后生成 App Password
SMTP_SECURE=false
```

4.4. 企业自建 SMTP

```
SMTP_HOST=mail.yourcompany.com
SMTP_PORT=25
SMTP_USER=notifications@yourcompany.com
SMTP_PASS=<密码>
SMTP_SECURE=false
```

env

内网 SMTP 服务器通常使用端口 25 无加密。如果服务运行在 Docker 中，请确保 `SMTP_HOST` 可从 Docker 容器内部访问。

5. 同步机制

服务端启动时，`syncSmtpFromEnv()` 函数自动将环境变量中的 SMTP 参数同步到数据库 `platform_configs` 表：

```
.env.local (SMTP_*)  →  env.ts (Zod 校验)  →  syncSmtpFromEnv()  →  platftext
```

这个同步仅在启动时执行。运行时修改 `platform_configs` 表中的 SMTP 配置同样即时生效，无需重启服务。

6. 验证 SMTP 配置

6.1. 方法一：Admin 后台测试邮件

1. 使用超级管理员账号登录 Admin 后台
2. 进入 系统配置 → SMTP 设置
3. 填写/确认 SMTP 参数

- 4. 点击 发送测试邮件，输入收件邮箱地址
- 5. 成功收到邮件则配置正确

6.2. 方法二：API 验证端点

超级管理员可以通过 API 直接验证 SMTP 连接：

```
# 仅验证连接（不发邮件）
POST /api/configs/verify-smtp
Authorization: Bearer <admin-jwt-token>

# 发送测试邮件
POST /api/configs/test-email
Authorization: Bearer <admin-jwt-token>
Content-Type: application/json

{"to": "admin@example.com"}
```

bash

6.3. 方法三：查看服务端日志

```
docker compose logs server | grep -i smtp
```

bash

正确配置后的日志输出示例：

```
info: [mailer] SMTP configuration validated successfully
```

错误的配置会显示具体错误（如连接超时、认证失败）。

7. 常见问题

问题	原因	解决方法
连接超时	SMTP_HOST / SMTP_PORT 不可达	检查 Docker 容器能否解析域名并访问 SMTP 服务器

问题	原因	解决方法
认证失败	<code>SMTP_USER</code> / <code>SMTP_PASS</code> 错误	检查凭据，部分服务需使用授权码而非登录密码
SSL/TLS 握手失败	<code>SMTP_SECURE</code> 与端口不匹配	确认端口 465 对应 <code>true</code> ，端口 587 对应 <code>false</code>
发送成功但收不到邮件	被收件方 SMTP 服务器拦截为垃圾邮件	检查收件箱的垃圾邮件文件夹
发送方地址与认证用户不匹配	某些邮件服务器要求发件人地址必须与 <code>SMTP_USER</code> 一致	确认 <code>SMTP_USER</code> 与 From 地址相同

数据库备份与恢复

EcoCtrl 所有重要数据（用户、设备、能耗记录、配置、工作流定义等）存储在 PostgreSQL 数据库中。备份方案基于 PostgreSQL 原生工具 `pg_dump`，可选择自动化定时备份。

1. 备份策略概览

备份类型：逻辑备份（`pg_dump`）
存储位置：容器挂载卷或外部 S3
调度方式：手动 / 定时（`pg-boss cron`）
保留策略：按天数配置保留周期

text

2. 手动备份

2.1. 使用 docker compose exec

```
# 备份到宿主机当前目录
docker compose exec -T postgres pg_dump -U ecoctrl ecoctrl > ecoctrl-backup-$(d

# 压缩备份
docker compose exec -T postgres pg_dump -U ecoctrl ecoctrl | gzip > ecoctrl-bac
```

bash

2.2. 使用 pg_dump（远程连接）

如果使用托管 PostgreSQL：

```
pg_dump -h <host> -U <user> -d ecoctrl -F c -f ecoctrl-backup.dump
```

bash

`-F c` 使用自定义格式，支持 `pg_restore` 的并行恢复和选择性恢复。

3. 定时自动备份

3.1. 配置方式

在 Admin 后台的 系统配置 页面中：

1. 启用 自动备份 (`autoBackup = true`)
2. 设置 备份保留天数 (`backupRetentionDays` ，默认 30 天)

`autoBackup` 启用后，服务端通过 pg-boss 调度一条定时任务执行备份。

3.2. 备份调度实现

备份调度信息存储在 `backup_schedules` 表中：

```
interface BackupSchedule {  
  id: string;  
  nextBackup: string; // ISO 8601 时间戳  
  createdAt: string;  
  updatedAt: string;  
}
```

typescript

平台中的 `backup_schedules` 仅记录下一次执行时间。实际备份执行需要依赖托管 PostgreSQL 的计划任务或外部 cron 脚本。

当前版本中的自动备份是调度框架（记录下一次时间），备份执行逻辑仍需配合以下方案之一：

- 宿主机 cron 调用 `docker compose exec postgres pg_dump`
- 托管 PostgreSQL 的内置备份功能（RDS、Cloud SQL、Supabase 等）
- 外部备份工具（pgBackRest、pg_probackup 等）

3.3. 基于宿主机 cron 的备份示例

```
# /etc/cron.d/ecoctrl-backup  
0 3 * * * root docker compose -f /opt/ecoctrl/docker/compose.yaml exec -T postg
```

bash

此 cron 实现：

- 每天凌晨 3:00 执行完整备份
- 保留最近 30 天的备份文件（自动清理 30 天前的文件）

4. 备份保留策略

维度	建议	说明
本地保留	30 天	使用 <code>find -mtime +30 -delete</code> 清理
异地备份	每周一次送至对象存储	防止本地故障导致数据完全丢失
备份前验证	<code>pg_restore --list</code>	确认备份文件可读
恢复演练	每季度一次	在测试环境验证恢复流程

5. 恢复步骤

5.1. 恢复完整备份

```
bash
# 停止 API 服务（避免恢复期间写入数据）
docker compose stop server

# 删除旧数据库（谨慎操作！）
docker compose exec postgres psql -U ecoctrl -c "DROP DATABASE IF EXISTS ecoctrl;"
docker compose exec postgres psql -U ecoctrl -c "CREATE DATABASE ecoctrl;"

# 恢复数据
cat ecoctrl-backup-20250101_030000.sql | docker compose exec -T postgres psql -

# 启动 API 服务
docker compose start server
```

5.2. 恢复压缩备份


```
gunzip -c ecoctrl-backup-20250101_030000.sql.gz | docker compose exec -T postgres bash
```

5.3. 使用 pg_restore (自定义格式)

```
docker compose exec -T postgres pg_restore -U ecoctrl -d ecoctrl -v < ecoctrl-b bash
```

6. 数据完整性验证

恢复后建议执行以下检查：

```
# 检查用户表
docker compose exec postgres psql -U ecoctrl -c "SELECT count(*) FROM users;"

# 检查平台配置
docker compose exec postgres psql -U ecoctrl -c "SELECT count(*) FROM platform_"

# 检查备份调度记录
docker compose exec postgres psql -U ecoctrl -c "SELECT * FROM backup_schedules" bash
```

7. 最佳实践

1. 定期验证备份：不要等到需要恢复时才发现备份文件损坏
2. 异地存储：至少保留一份备份在 MinIO、AWS S3 或 NAS 上
3. 备份前检查磁盘空间：大数据库备份可能消耗大量磁盘空间
4. 事务性一致性：使用 `pg_dump` 默认的事务快照模式，保证备份数据一致性
5. 文档化恢复流程：确保团队成员知道如何执行恢复

安全提醒：备份文件包含所有数据库内容（包括密码哈希和用户邮箱）。将备份文件保存到安全位置，限制访问权限。

性能调优

本文介绍如何针对不同规模的部署场景，优化 EcoCtrl 各服务的性能表现。

1. PostgreSQL 连接池

EcoCtrl 服务端使用 `pg-pool` 管理数据库连接。合理配置连接池可以避免数据库连接耗尽或过度空闲。

1.1. 配置参数

```
# docker/.env.local
DB_POOL_MIN=2
DB_POOL_MAX=10
```

bash

参数	默认值	说明	推荐值
DB_POOL_MIN	2	连接池最小保持的连接数	CPU 核数 * 2
DB_POOL_MAX	10	连接池最大连接数	CPU 核数 * 4

1.2. 规模参考

部署规模	CPU	DB_POOL_MIN	DB_POOL_MAX	PostgreSQL max_connections
小型	2	2	10	50
中型	4	4	20	100
大型	8	8	40	200

连接池的总连接数不应超过 PostgreSQL `max_connections` 的 1/3，为管理系统预留连接余量。

1.3. 监控连接数

bash

```
# 查看当前活跃连接数
docker compose exec postgres psql -U ecoctrl -c "
SELECT count(*) as active_connections FROM pg_stat_activity WHERE datname = 'ec
"

# 查看连接等待状态
docker compose exec postgres psql -U ecoctrl -c "
SELECT wait_event_type, wait_event, count(*)
FROM pg_stat_activity
WHERE state = 'active'
GROUP BY wait_event_type, wait_event;
"
```

1.4. 慢查询分析

bash

```
docker compose exec postgres psql -U ecoctrl -c "
SELECT query, calls, mean_time
FROM pg_stat_statements
ORDER BY mean_time DESC
LIMIT 10;
"
```

如果 `pg_stat_statements` 无数据，需要先启用扩展：`CREATE EXTENSION IF NOT EXISTS pg_stat_statements;`

2. Docker 资源限制

Docker Compose 部署时，可以通过 `deploy.resources` 设置各容器的资源上限，防止某个服务耗尽主机资源。

2.1. 推荐配置

yaml

```
# 在 compose.yaml 中配置 (以 server 为例)
services:
  server:
    deploy:
      resources:
        limits:
          cpus: "2"
          memory: 1G
        reservations:
          cpus: "0.5"
          memory: 256M
```

2.2. 各服务参考值

服务	CPU 上限	内存上限	CPU 预留	内存预留	说明
server	2	1 GB	0.5	256 MB	API 服务，主要消耗 CPU
web	1	256 MB	0.25	128 MB	Caddy 静态文件服务
admin	1	256 MB	0.25	128 MB	Caddy 静态文件服务
postgres	2	1 GB	0.5	512 MB	数据库，内存影响查询性能
minio	1	512 MB	0.25	256 MB	对象存储

2.3. PostgreSQL 内存参数

为 PostgreSQL 容器设置 `shared_buffers` 参数，建议值为可用内存的 25%：

yaml

```
services:
  postgres:
    command: >
      -c shared_buffers=256MB
      -c work_mem=16MB
      -c effective_cache_size=768MB
      -c random_page_cost=1.1
```

参数	建议值	说明
<code>shared_buffers</code>	总内存的 25%	共享缓冲区大小
<code>work_mem</code>	16-64 MB	单次查询排序/哈希操作内存上限
<code>effective_cache_size</code>	总内存的 75%	操作系统缓存估算值
<code>random_page_cost</code>	1.1 (SSD)	随机 I/O 成本，SSD 设为 1.1

3. SSE 队列背压处理

SSE (Server-Sent Events) 用于向前端推送实时数据。

3.1. 背压机制

EcoCtrl 在 SSE 连接层面实现了以下背压策略：

- 1. 写缓冲区：每个 SSE 连接维护有限长度的写缓冲区
- 2. 丢弃策略：缓冲区满时丢弃最旧的消息，保证最新数据优先到达
- 3. 连接健康检查：30 秒心跳间隔，检测到客户端断连时及时回收资源

3.2. 监控背压

查看 SSE 连接丢弃事件

`docker compose logs server | grep "SSE buffer full"`

检查活跃 SSE 连接数

`docker compose logs server | grep "SSE connected" | tail -5`

bash

3.3. 优化建议

场景	建议
大量客户端并发连接 (>100)	增加服务器实例数，或使用消息队列作为 SSE 中继

场景	建议
消息推送频率过高 (< 100ms 间隔)	在前端合并短时间内的事件 (debounce)
部分客户端消费速度慢 (网络不佳)	缩短缓冲区超时时间，加速清理慢客户端
推送目标数过多 (> 1000 客户端)	切换到 WebSocket + Redis Pub/Sub 架构

4. 其他优化建议

4.1. 3D 模型优化

建议使用项目自带的模型优化脚本压缩 glTF 文件：

```
pnpm tsx scripts/optimize-model.ts --input model.glb --output model-optimized.glb
```

bash

该脚本会执行 Draco 压缩、纹理合并和顶点优化。

4.2. 能耗查询优化

`energy_readings` 表会随时间快速增长。确保以下索引存在：

```
CREATE INDEX IF NOT EXISTS idx_energy_readings_area_recorded
ON energy_readings (area_id, recorded_at DESC);

CREATE INDEX IF NOT EXISTS idx_faults_status_level_triggered
ON faults (status, level, triggered_at DESC);
```

sql

4.3. Node.js 内存限制

如果 API 服务出现内存压力，可通过 `NODE_OPTIONS` 限制堆内存：

```
# 在 Dockerfile 或 compose.yaml 中设置
```

bash


```
NODE_OPTIONS="--max-old-space-size=512"
```

安全加固

EcoCtrl 遵循纵深防御原则。本文介绍各安全层级的配置建议和操作步骤。

1. JWT Secret 轮换

JWT (JSON Web Token) 用于 API 请求的身份认证。 `JWT_SECRET` 是签名的核心密钥。

1.1. 密钥强度要求

```
# 生成一个安全的 JWT_SECRET (至少 32 字符, 建议 64 字符)
openssl rand -base64 64
```

bash

1.2. 完整轮换流程

更换 `JWT_SECRET` 会使所有已签发的 Access Token 立即失效, 但 Refresh Token 仍然有效:

```
#### 更新环境变量
# docker/.env.local
JWT_SECRET=<new-secret>

#### 重启 API 服务
docker compose up -d server

#### 【可选】强制所有用户重新登录 (使所有 Refresh Token 失效)
docker compose exec postgres psql -U ecoctrl -d ecoctrl -c "TRUNCATE refresh_to
```

bash

步骤	执行前	执行后
仅更新 JWT_SECRET	用户在线, Access Token 仍有效	重启后所有 Access Token 立即失效

步骤	执行前	执行后
重启服务	用户收到 401，前端自动刷新 Token	Refresh Token 换取新 Access Token
TRUNCATE refresh_tokens	用户可无感刷新	所有用户被迫重新登录

1.3. 生命周期建议

- 生产环境：每 90 天轮换一次
- 安全事件后：立即轮换（如发现密钥泄露或员工离职）

1.4. 密钥存储

环境	存储方式
开发	<code>.env.local</code> 文件
测试	CI/CD Secrets + <code>.env.local</code>
生产	密钥管理服务（AWS Secrets Manager、HashiCorp Vault、1Password CLI）

2. CORS 配置

CORS (Cross-Origin Resource Sharing) 限制跨域请求的来源。

2.1. 配置方式

```
# docker/.env.local
CORS_ORIGIN=https://admin.yourdomain.com,https://web.yourdomain.com
```

bash

2.2. 环境差异

环境	CORS_ORIGIN 值	说明
开发	* （默认允许所有来源）	方便本地调试
测试	http://localhost:4173,http://localhost:8081	仅允许本地前端访问
生产	https://admin.yourdomain.com,https://web.yourdomain.com	仅允许受信域名

2.3. 安全规则

- 禁止使用通配符：生产环境永远不要设置 `CORS_ORIGIN=*`
- 只列出受信域名：只包含实际需要访问 API 的前端域名
- 逗号分隔：多个域名之间用 英文逗号 分隔

2.4. 验证 CORS 配置

```
bash
curl -H "Origin: https://evil-site.com" -H "Access-Control-Request-Method: GET" \
  -X OPTIONS http://localhost:3000/api/health -v 2>&1 | grep -i "access-control"
```

正确响应应不返回 `Access-Control-Allow-Origin: https://evil-site.com` 。

3. Content-Security-Policy

CSP 通过 HTTP 响应头限制页面可以加载的资源来源，防止 XSS 攻击。

3.1. 建议的生产环境 CSP

```
caddyfile
# 在 Caddyfile 中添加
header Content-Security-Policy "
  default-src 'self';
  script-src 'self' 'unsafe-inline';
  style-src 'self' 'unsafe-inline';
  img-src 'self' data: blob: https://*.tile.openstreetmap.org;
  connect-src 'self' https://api.openweathermap.org;
  font-src 'self';
```

```
frame-src 'self';
object-src 'none';
base-uri 'self';
"
```

指令	说明
default-src	所有资源类型的默认来源
script-src	允许自身脚本和内联脚本（ <i>审查内联脚本安全性</i> ）
style-src	允许自身和内联样式
img-src	允许自身、data URI、blob 和地图瓦片
connect-src	允许的 API 连接目标
frame-src	允许内嵌的 frame（如 WebTalk）
object-src	禁止 <object> 标签（防止插件类攻击）

添加 CSP 后请全面测试所有功能，特别是 3D 场景（WebGL）渲染是否正常。

4. OAuth 凭据管理

微信和飞书登录需要配置 APP_ID 和 APP_SECRET。

4.1. 安全建议

- AppSecret 视同密码：不要在代码仓库、日志或 URL 参数中泄露
- 定期轮换：在 OAuth 提供商管理后台定期重新生成 AppSecret
- 最小权限：只申请登录必需的权限范围（scope）
- 回调 URL 白名单：在 OAuth 提供商后台严格限制回调 URL 为生产域名

4.2. 验证配置

bash

```
# 检查 OAuth 提供商列表
curl http://localhost:3000/api/auth/oauth/providers
```

预期响应示例：

json

```
{
  "providers": ["wechat", "feishu"]
}
```

4.3. WebTalk 凭据

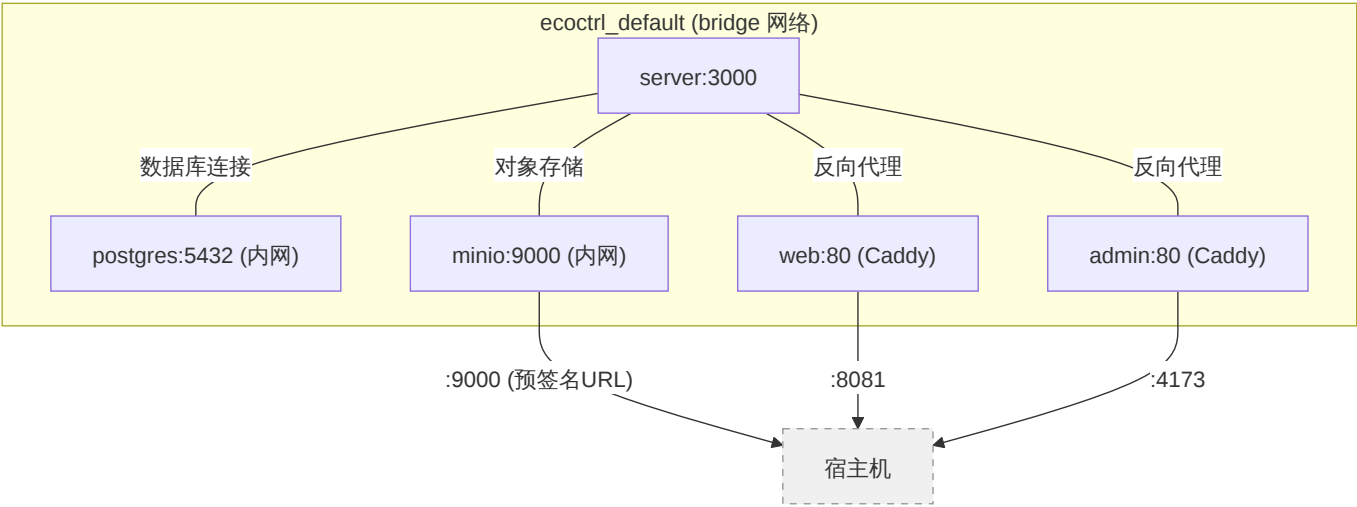
WebTalk（内部通讯工具）的用户名和密码通过 Admin UI 提交，存储在服务端内存/会话中：

- 凭据不会持久化到数据库
- 退出登录时自动清除缓存的 WebTalk 认证凭据
- 每次 WebTalk 会话独立管理凭据生命周期

5. Docker 网络隔离

5.1. 网络架构

Compose 部署使用内部 bridge 网络，服务间通过服务名称互相访问：



5.2. 安全基线

措施	说明
最小暴露原则	仅向外暴露 admin(4173)、web(8081)、minio(9000)
数据库不对外暴露	PostgreSQL 仅在内部网络中可访问
MinIO 文件访问	通过 Server 预签名 URL 授权访问
使用非 root 用户运行容器	所有应用容器使用 <code>node</code> 用户（非 root）
Caddy 处理 TLS 和路由	Caddy 作为反向代理处理 HTTPS 和路径重写

5.3. 验证端口暴露

```
# 检查哪些端口暴露到宿主机
docker compose ps

# 验证数据库不可从宿主机直接访问（除非特意映射了端口）
psql -h localhost -U ecoctrl -d ecoctrl # 应返回连接失败
```

bash

版本升级指南

EcoCtrl 的版本管理遵循 [SemVer](#) 规范。Docker 镜像标签使用语义化版本（如 `v1.2.3` ），`latest` 标签指向最新的稳定版本。

1. 标准升级流程

1.1. 第一步：查看更新日志

在升级前，查阅 [GitHub Releases](#) 了解新版本的变更内容、破坏性变更和迁移说明。

1.2. 第二步：备份数据库

```
# 执行完整数据库备份
cd docker
docker compose exec -T postgres pg_dump -U ecoctrl ecoctrl > pre_upgrade_$(date)
```

bash

升级前务必备份数据库。如果升级涉及数据库迁移（migration），回滚时需要此备份。

1.3. 第三步：拉取新镜像

```
# 拉取所有服务的最新镜像
docker compose pull

# 或指定版本标签（推荐生产环境使用）
docker compose pull --image-policy=always
```

bash

1.4. 第四步：重启服务

```
docker compose up -d
```

bash

新镜像将自动替换旧容器。如果 `compose.yaml` 使用了 `:latest` 标签，Docker 会使用刚拉取的最新镜像。

1.5. 第五步：运行数据库迁移

API 服务启动时默认自动运行 Drizzle 迁移（migration）。如果未自动执行，可手动运行：

```
docker compose exec server npx drizzle-kit migrate
```

bash

迁移过程可能耗时数秒到数分钟（取决于数据量）。迁移期间 API 服务保持可用，但部分新功能可能暂不可用，直到迁移完成。

1.6. 第六步：验证升级

```
# 检查服务健康状态
curl http://localhost:3000/health

# 检查版本信息
curl http://localhost:3000/api/version

# 查看所有容器运行状态
docker compose ps

# 检查 workflow 执行是否正常
docker compose logs server | grep "workflow"
```

bash

2. 破坏性变更检查清单

类别	检查项	说明
数据库	检查是否有新增数据库迁移	缺少迁移会导致 500 错误
环境变量	检查是否有新增/废弃/必填的环境变量	缺少必填变量会导致服务启动失败
API	检查 API 路由或响应格式是否有变化	前端可能因接口变更而功能异常

类别	检查项	说明
前端	检查是否有浏览器兼容性变化	可能需要更新浏览器版本
OAuth	检查回调 URL 是否有变化	需同步更新 OAuth 提供商的白名单
Docker	检查 <code>compose.yaml</code> 是否有服务或端口变更	数据卷、端口映射或网络配置可能受影响
3D 模型	检查 3D 模型格式是否有变更	旧模型文件可能需要重新导出
配置	检查是否有新增的配置文件或配置项	可能需要更新 <code>docker/.env.local</code>

2.1. 如何检查变更

```
# 对比 compose.yaml 变化
git diff v1.2.3..v2.0.0 -- docker/compose.yaml

# 查看环境变量变更
git diff v1.2.3..v2.0.0 -- docker/.env.example

# 查看数据库迁移文件
ls packages/server/src/db/migrations/

# 查看新增 API 路由
git diff v1.2.3..v2.0.0 -- packages/server/src/routes/
```

3. 版本回退

如果升级后发现问题，按以下步骤回滚：

3.1. 回滚数据库

```
### 停止服务
docker compose down

### 恢复数据库（使用升级前的备份）
```

```
cat pre_upgrade_20260701.sql | docker compose exec -T postgres psql -U ecoctrl

### 回滚镜像版本
# 编辑 compose.yaml, 将 image 标签改回旧版本号

### 启动服务
docker compose up -d
```

3.2. 使用指定版本标签

yaml

```
# compose.yaml 中指定具体版本（生产环境推荐）
services:
  server:
    image: ghcr.io/hyooeewee/ecoctrl-server:v1.2.3
  web:
    image: ghcr.io/hyooeewee/ecoctrl-web:v1.2.3
  admin:
    image: ghcr.io/hyooeewee/ecoctrl-admin:v1.2.3
```

推荐做法：生产环境始终使用具体版本标签（如 `v1.2.3`），而非 `latest`。这样回滚时只需修改标签值并重启。

4. 多环境升级策略

环境	升级方式	说明
开发	直接拉取 <code>:latest</code> 或 <code>:dev</code> 标签	快速迭代，允许短暂不可用
测试	使用预发布标签，运行自动化测试套件	验证无破坏性变更后升级生产
生产	滚动升级（先升级 server，再升级前端）	分批升级，监控每批服务状态

4.1. 生产环境滚动升级

bash

先升级 API 服务

```
docker compose up -d server
```

```
# 检查: curl http://localhost:3000/health
```

再升级前端

```
docker compose up -d web admin
```

最后手动执行数据库迁移（如未自动运行）

```
docker compose exec server npx drizzle-kit migrate
```

全面验证

```
docker compose ps
```

常见问题排查

本文按类别列出 EcoCtrl 部署和使用中的常见问题及解决方法。

1. 安装与配置

1.1. Node.js 版本不匹配

现象： `pnpm install` 时出现引擎版本错误，或 `node --version` 显示非 24.x。

解决：

```
# 使用 nvm 切换到正确的 Node.js 版本
nvm install 24
nvm use 24

# 或使用 corepack 启用自动版本管理
corepack enable
pnpm --version # 应显示 10.33.1+
```

bash

1.2. pnpm 版本不匹配

现象： `pnpm install` 报错 `ERR_PNPM_UNEXPECTED_STORE` 或 `Unsupported pnpm version`。

解决：

```
# 启用 corepack 自动管理版本
corepack enable
corepack prepare pnpm@10.33.1 --activate

# 或直接安装指定版本
npm install -g pnpm@10.33.1
```

bash

1.3. 数据库 "ecoctrl" 不存在

现象：API 启动时报 `database "ecoctrl" does not exist`。

解决：

```
# 手动创建数据库
docker compose exec postgres createdb -U ecoctrl ecoctrl

# 执行迁移和种子数据
pnpm db:push
pnpm db:seed
```

bash

Docker 部署的用户请使用 `docker compose exec server pnpm db:push` 和 `docker compose exec server pnpm db:seed` 替代，宿主机上的 pnpm 无法直接连接容器内的数据库。

1.4. 端口冲突

现象：`docker compose up` 时报 `port is already allocated`。

解决：

```
# 检查占用端口的进程
sudo lsof -i :3000    # API 端口
sudo lsof -i :4173    # Admin 端口
sudo lsof -i :8081    # Web 端口

# 修改 docker/.env.local 中的端口变量
SERVER_PORT=3001
ADMIN_PORT=4174
WEB_PORT=8082
```

bash

2. 认证与授权

2.1. 15 分钟被强制登出

现象：登录后每 15 分钟被强制退出，频繁弹出登录页面。

原因与解决：

原因	检查方法	解决
Access Token 正常过期	浏览器 Console 是否有 <code>/auth/refresh</code> 报错	前端应自动刷新，检查网络连接
<code>JWT_SECRET</code> 前后不一致	检查 <code>.env.local</code> 是否被修改或覆盖	统一所有环境的 <code>JWT_SECRET</code> 值
服务端时间不同步	<code>docker compose exec server date</code>	同步服务器时间（NTP）
Token 存储异常	浏览器 Application > Local Storage	清除 Storage 后重新登录

2.2. OAuth 按钮点击无反应

现象：点击微信/飞书登录按钮后页面不跳转或报错。

解决：

```
# 检查 OAuth 配置是否加载
curl http://localhost:3000/api/auth/oauth/providers

# 确认浏览器允许弹窗
# 检查回调 URL 是否与 OAuth 提供商后台配置一致
# 回调 URL 格式: {BASE_URL}/api/auth/oauth/{provider}/callback

# 查看服务端日志中的 OAuth 错误
docker compose logs server | grep -i oauth
```

3. 3D 可视化

3.1. 建筑视图黑屏

现象：打开 Web 门户页面显示黑屏，浏览器控制台无报错。

可能原因：

原因	检查方法	解决
未上传建筑模型	Admin 后台 > 3D 配置 > 模型管理	上传 .glb/.gltf 格式的 3D 模型
WebGL 不兼容	访问 chrome://gpu 检查 WebGL 状态	更新显卡驱动或更换浏览器
默认模型文件缺失	检查 apps/web/public/building.glb 文件是否存在	上传模型或恢复默认模型文件
BASE_URL 配置错误	检查 3D 模型资源加载的 URL 是否正确	调整 BASE_URL 或 STATIC_PREFIX
模型文件损坏或格式不兼容	浏览器网络面板查看模型加载 HTTP 状态	重新导出模型为 glTF 2.0 格式

Babylon.js 加载模型失败时默认静默处理，不会在前端抛出明显的错误提示。建议打开浏览器开发者工具的 Network 面板，筛选 glb/gltf 请求查看是否返回 404。

3.2. 3D 模型加载慢

解决：

- 压缩模型文件：使用 [gltf-transform](#) 或 Draco 压缩
- 启用 MinIO 缓存：确保模型文件存储在 MinIO 而非数据库
- 网络优化：将模型文件部署到 CDN 或使用更近的服务器

3.3. 自动旋转相机不工作

现象：3D 场景静止不动，页面没有进入自动旋转模式。

检查：

1. Admin 后台 > 3D 配置 > 相机设置，检查「自动旋转」是否开启
2. 确认用户交互后超时时间配置是否合理（默认 30 秒无操作后恢复旋转）

4. 构建部署

4.1. resolveUiAlias 错误

现象： `pnpm build` 时出现 `Module not found: Error: Can't resolve '#ui/...'` 。

原因：TypeScript 路径别名 `#ui` 未正确解析。

解决：

```
# 确保已构建 UI 组件库
pnpm --filter @ecoctrl/ui build

# 确认 vite.config.ts 已引入 resolveUiAlias() 插件
# pnpm install 完成后清除缓存重试
pnpm clean
pnpm install
pnpm build
```

bash

4.2. 找不到工作空间包

现象： `pnpm build` 时报 `ERR_PNPM_NO_PACKAGE_MANIFEST` 或 `Cannot find module '@ecoctrl/shared'` 。

解决：

```
# 确认所有依赖包已安装
pnpm install

# 确认工作空间配置正确
cat pnpm-workspace.yaml
# 应包含：
# packages:
#   - 'apps/*'
#   - 'packages/*'

# 按依赖顺序构建
pnpm --filter @ecoctrl/shared build
```

bash

```
pnpm --filter @ecoctrl/server build
pnpm --filter @ecoctrl/web build
pnpm --filter @ecoctrl/admin build
```

4.3. Docker 构建失败 (OOM)

现象： `docker compose -f compose.build.yaml build` 失败，进程退出码 137 (SIGKILL)。

解决：

使用串行构建脚本（推荐）

```
cd docker
./build.sh
```

bash

或手动设置 Node 内存限制后逐个构建

```
docker compose -f compose.build.yaml build --memory=2g server
docker compose -f compose.build.yaml build --memory=1g web
docker compose -f compose.build.yaml build --memory=1g admin
```

4.4. API_BASE_URL 修改后前端无变化

现象：修改 `API_BASE_URL` 后刷新页面，前端仍然请求旧地址。

解决：前端使用字面量 `/api` 前缀，运行时的反向代理（Caddy）负责将请求改写转发到后端。
修改 Caddyfile 或 Nginx 配置即可，无需重新构建前端。

5. 运行时故障

5.1. 数据库连接失败

现象：服务端启动时日志报 `Error: connect ECONNREFUSED 127.0.0.1:5432`。

检查清单：

1. PostgreSQL 容器是否正在运行： `docker compose ps | grep postgres`

2. DATABASE_URL 格式是否正确: `postgresql://user:password@host:5432/dbname`
3. 启动顺序: PostgreSQL 先准备就绪后再启动 server (健康检查应确保顺序)
4. 网络连通性: `docker compose exec server ping postgres`

5.2. API 响应缓慢

现象: 前端页面加载缓慢, API 请求耗时超过 2 秒。

排查步骤:

bash

检查数据库连接数

```
docker compose exec postgres psql -U ecoctrl -c "SELECT count(*) FROM pg_stat_a
```

检查慢查询

```
docker compose exec postgres psql -U ecoctrl -c "  
SELECT query, calls, mean_time  
FROM pg_stat_statements  
ORDER BY mean_time DESC  
LIMIT 10;  
"
```

检查服务器资源使用

```
docker compose stats
```

5.3. 邮件发送失败

现象: 密码重置邮件、通知邮件无法发送。

排查:

bash

检查 SMTP 配置

```
docker compose logs server | grep -i smtp
```

测试 SMTP 连接

在 Admin 后台系统配置 > SMTP 设置中点击「发送测试邮件」

检查 pg-boss 队列中待发送邮件

```
docker compose exec postgres psql -U ecoctrl -c "
```

```
SELECT count(*) FROM pgboss.job WHERE name = 'send-email' AND state = 'created'
"
```

5.4. 数据不同步

现象：Web 门户或 Admin 后台显示的数据不是最新值。

可能原因：

- SSE 连接断开：检查浏览器网络面板的 SSE EventStream 连接状态
- 缓存未刷新：硬刷新页面 (Ctrl+Shift+R) 清除浏览器缓存
- IoT 数据上报延迟：检查 `BASE_URL` 连通性及 IoT 设备状态
- Mock 模式激活：查看日志中是否有 `[lighting]` Mock 告警标记

5.5. 磁盘空间不足

现象：服务异常、Docker 操作失败，报 `no space left on device`。

清理步骤：

```
### 清理 Docker 无用的镜像和卷
docker system prune -af
```

```
### 检查数据卷使用情况
docker system df
```

```
### 清理 MinIO 中的过期模型文件
# Admin 后台 > 存储管理 > 清理未引用模型
```

```
### 检查 PostgreSQL 数据大小
docker compose exec postgres psql -U ecoctrl -c "
SELECT pg_database_size('ecoctrl')/1024/1024 as size_mb;
"
```

bash

5.6. 运行 `pnpm db:push` 后 Admin 无数据

现象：数据库迁移执行成功，但 Admin 后台显示空白无数据。

解决：

bash

运行种子数据填充脚本

`pnpm db:seed`

如需完全重置数据库（破坏性操作，清除所有数据）

`pnpm db:refresh`