MODEL

新增了ystc_strategy_detail、ystc_control_command、ystc_game_strategy表的model，因此可以通过SQLAlchemy将博弈结果写入数据库中。

同样的也增加了ystc_user_auth_token、ystc_device表的model，完善了登录之策，以及这个device对于SN序列号的绑定。

CORE

Core由`cycle_manager.py`（周期管理）和`jar_executor.py`（博弈调用）组成。为了实现按固定周期自动化执行博弈策略：

- 周期管理器负责「调度周期、控制启停、防重复执行」
- JAR 执行器负责「预处理设备数据、调用博弈模型、解析结果、落地数据库」

cycle_manager.py

1. 核心设计目标

确保周期服务 “只启动一次、每个周期只执行一次、异常自动恢复”，同时按固定时间间隔驱动博弈流程，是整个周期调用的 “总指挥”。

2. 关键控制变量

变量名	核心作用
`IS_LOOP_RUNNING`	全局标记周期服务是否已启动，杜绝`service_loop`被重复调用（比如应用热重载、异步任务重复提交导致的多实例运行）；
`EXECUTED_CYCLES`	集合类型，记录已执行的周期 ID（如`2025-12-12T14:35:16.386067+11:00`），防止同一个周期被重复执行；
`get_current_cycle.start_time`	周期基准时间（时间戳），是所有周期计算的 “锚点”，确保全系统用同一基准生成周期 ID，避免时间混乱；

3. 核心函数逻辑

（1）start_cycle_service()：周期服务的安全启动入口

- 是外部启动周期服务的唯一入口，先检查`IS_LOOP_RUNNING`：若已为`True`，直接打印警告并返回，避免重复启动；若为`False`，通过`asyncio.create_task`异步启动`service_loop`（主循环），不阻塞主线程；
- 设计目的：避免服务重复启动导致的 “一个周期执行多次” 问题。

（2）service_loop()：周期主循环

这是周期调度的核心，逻辑链路如下：

1. **初始化准备**：标记 `IS_LOOP_RUNNING = True`，打印启动日志，然后等待至少一个设备完成注册（无设备则每 5 秒检查一次，直到有设备注册）；
2. **基准时间初始化**：生成当前时间戳作为周期基准时间，同步到 `app.utils.cycle` 模块的全局变量（确保其他模块计算周期时用同一基准）；
3. 无限循环调度周期：
   - 计算当前周期 ID：调用 `get_current_cycle()`（基于基准时间 + 周期间隔生成标准化的周期 ID，如 `2025-12-12T14:35:16.386067+11:00`）；
   - 等待上传窗口关闭：周期启动后，先等待 `config.UPLOAD_WINDOW` 秒（设备上传数据的窗口期），确保所有设备数据已上传；
   - 执行单个周期逻辑：调用 `run_cycle(当前周期ID)`；
   - 清理过期数据：调用 `clean_expired_data()` 清理历史周期的无效数据；
   - 等待下一个周期：计算下一个周期的开始时间，休眠对应时长后进入下一轮循环；
4. **异常容错**：若循环中抛出异常，重置 `IS_LOOP_RUNNING = False`，休眠 10 秒后重启 `service_loop`，保证服务不宕机。

（3）run_cycle(cycle_time)：单个周期的执行逻辑

每个周期的具体任务都在这里触发，核心逻辑：

1. **防重复执行校验**：先检查 `cycle_time` 是否在 `EXECUTED_CYCLES` 中，若已存在则打印警告并返回，避免同一周期重复执行；若不存在则加入集合标记；
2. **设备数据检查**：通过 `STORAGE_LOCK` 加锁读取 `DEVICE_DATA`（全局内存中存储的设备上传数据），判断当前周期是否有设备数据；
3. 分支处理：
   - 无数据：跳过博弈，打印日志，清理当前周期数据，从 `EXECUTED_CYCLES` 中移除该周期标记；
   - 有数据：调用 `jar_executor.py` 中的 `call_jar_model(cycle_time)` 执行博弈模型，执行完成后清理当前周期数据，移除周期标记；
4. **异常处理**：无论博弈执行成功 / 失败，最终都会清理周期数据，避免内存泄漏，同时记录异常信息。

jar_executor.py

核心函数：call_jar_model (cycle_time)

`call_jar_model(cycle_time)` 是周期内博弈计算的唯一入口，输入周期时间标识，输出 JAR 模型计算结果（或 None），全程保障数据一致性和异常可控性，完整逻辑链路如下：

1. 前置校验

校验项	校验逻辑	异常处理
JAR 文件存在性	检查项目根目录下 `game-model-1.0.jar` 是否存在	日志报错，通过 `STORAGE_LOCK` 标记 `CYCLE_STATUS[cycle_time] = "failed"`，返回 None
周期设备数据有效性	从 `DEVICE_DATA[cycle_time]` 提取设备数据（结构：`{serial_number: device_data}`）	无数据则日志警告，标记 `CYCLE_STATUS[cycle_time] = "completed"`，返回 None

2. 设备数据预处理

JAR 模型强制要求「设备 ID 为从 0 开始的连续整数」，且核心字段有固定长度要求（如 `chargeSpeed` 为 10 个元素、`currentStorage` 为 3 个元素），但业务侧存在「设备原始 ID（前端 / 业务传入）」「数据库主键 ID（ystc_device.id）」两种 ID，因此需通过三层映射完成格式适配，同时强制修复模型必填字段的长度：

2.1 核心映射表构建

映射表名称	结构	用途
`new_id_original_id_map`	`{新ID(int): 原设备ID}`	将业务侧的原始 ID 转换为 JAR 要求的连续新 ID（0/1/2...）
`original_id_serial_map`	`{原设备ID: (序列号, 数据库Device主键ID)}`	存储原始 ID 与「设备序列号 + 数据库主键 ID」的关联关系，用于后续落库
`serial_user_map`	`{序列号: 用户ID(int)}`	关联设备序列号与所属用户 ID，实现“按用户拆分博弈结果”

2.2 字段规则

JAR 模型对核心字段有**硬性长度要求**，其中仅 `produce` 字段做兼容修复（避免因该字段异常导致整体失败），其余字段要求业务传入时必须符合长度规则（不可缺少、不可格式错误）：

字段类别	涉及字段	长度要求	处理规则（代码现状）
3 长度时间片字段	`produce`	TimeSlots	1. 为 `None` → 补全为 `[0.0] * TimeSlots`； 2. 非数组 → 强制转为 `[0.0] * TimeSlots`； 3. 长度不足 → 末尾补 0； 4. 长度过长 → 截断至 TimeSlots 个元素

字段类别	涉及字段	长度要求	处理规则（代码现状）
3 长度时间片字段	`currentStorage`、`demands`	TimeSlots	必填项，要求业务传入时必须是长度为 TimeSlots 的数组，代码不做修复，缺失 / 格式错误会导致 JAR 执行失败
10 长度参数字段	`chargeSpeed`、`chargeCost`、`dischargeSpeed`、`dischargeCost`	10	必填项，要求业务传入时必须是长度为 10 的数组，代码不做修复，缺失 / 格式错误会导致 JAR 执行失败
单值字段	`overallCapacity`	-	必填项，无值则默认补 0.0，业务传入时建议显式传值

2.3 预处理后设备数据结构

```
# 单设备预处理后结构示例
{
    "id": 0,  # JAR要求的连续新ID（从0开始）
    "produce": [100.0, 90.0, 80.0],  # 修复后长度为3的数组
    "currentStorage": [50.0, 50.0, 50.0],
    "demands": [20.0, 25.0, 18.0],
    "chargeSpeed": [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0],
    "chargeCost": [0.5, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1, 0.0, 0.1, 0.2],
    "dischargeSpeed": [8.0, 8.0, 8.0, 7.0, 7.0, 6.0, 6.0, 5.0, 5.0, 4.0],
    "dischargeCost": [0.3, 0.4, 0.3, 0.2, 0.1, 0.0, 0.1, 0.2, 0.3, 0.4],
    "overallCapacity": 100.0,  # 总容量（单值，无默认补0.0）
}
```

3. JAR 博弈模型调用

3.1 调用配置与调整

配置项	取值 / 逻辑
执行方式	`asyncio.to_thread` 异步调用 `subprocess.run`，避免阻塞周期调度主线程
超时控制	30 秒（防止 JAR 模型卡死导致周期挂起）
输出捕获	同时捕获 `stdout` / `stderr`，便于故障排查
编码	UTF-8，兼容中文日志 / 结果
输入格式调整	预处理后的设备数组需包裹在 `{"devices": [...]}` 外层键中（JAR 硬性要求）

3.2 JAR 模型输入格式

JAR 要求输入为**外层带 `devices` 键的 JSON 对象**，而非直接的设备数组，完整格式示例：

```
{
  "devices": [
```

```
    {
      "id": 0,
      "produce": [100.0, 90.0, 80.0],
      "currentStorage": [50.0, 50.0, 50.0],
      "demands": [20.0, 25.0, 18.0],
      "chargeSpeed": [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0],
      "chargeCost": [0.5, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1, 0.0, 0.1, 0.2],
      "dischargeSpeed": [8.0, 8.0, 8.0, 7.0, 7.0, 6.0, 6.0, 5.0, 5.0, 4.0],
      "dischargeCost": [0.3, 0.4, 0.3, 0.2, 0.1, 0.0, 0.1, 0.2, 0.3, 0.4],
      "overallCapacity": 100.0
    },
    {
      "id": 1,
      "produce": [80.0, 70.0, 60.0],
      "currentStorage": [40.0, 40.0, 30.0],
      "demands": [15.0, 20.0, 18.0],
      "chargeSpeed": [0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1],
      "chargeCost": [0.6, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1, 0.2, 0.3],
      "dischargeSpeed": [7.0, 7.0, 7.0, 6.0, 6.0, 5.0, 5.0, 4.0, 4.0, 3.0],
      "dischargeCost": [0.4, 0.5, 0.4, 0.3, 0.2, 0.1, 0.2, 0.3, 0.4, 0.5],
      "overallCapacity": 90.0
    }
  ]
}
```

3.3 JAR 模型输出格式

JAR 输出可能包含日志信息，需通过 `re.search(r'\{[\s\S]*\}', result.stdout)` 提取核心 JSON，格式如下：

```
{
    "full_result": {
        "iteration": 100,          // 模型迭代次数
        "timeConsumption": 0.5,    // 计算耗时（秒）
        "benefit": 200.0,          // 总收益
        "cost": 50.0,              // 总成本
        "revenue": 150.0,          // 总利润（收益-成本）
        "decisions": [             // 每个设备的博弈决策
            {
                "deviceId": 0,     // JAR返回的新ID（需映射回原ID）
                "dc": [1, 0, -1],  // 决策类型：1=充电，0=闲置，-1=放电（按时间片）
                "speed": [10.0, 0.0, 8.0], // 功率设定值（按时间片）
                "cost": [0.5, 0.0, 0.3],   // 成本（按时间片）
                "benefit": 80.0            // 该设备总收益
            },
            {
                "deviceId": 1,
                "dc": [0, -1, 1],
                "speed": [0.0, 8.0, 10.0],
                "cost": [0.0, 0.3, 0.5],
                "benefit": 70.0
            }
        ]
    }
}
```

4. 结果解析与多表联动落库

4.1 结果预处理

1. 将 JAR 返回的 `deviceId`（新 ID）通过 `new_id_original_id_map` 映射回「设备原始 ID」；
2. 通过 `original_id_serial_map` 从「原始 ID」获取「设备序列号 + 数据库主键 ID」；
3. 通过 `serial_user_map` 从「序列号」获取「用户 ID」，按用户拆分博弈结果（`user_decision_map`）；
4. 为每个决策新增 `_db_device_id` 临时字段（存储数据库 Device 主键 ID），避免落库时重复查询数据库。

4.2 落库核心函数：write_strategy_to_db ()

该函数接收「数据库会话、周期时间、用户级博弈结果、用户 ID、ID 映射表」，完成 `ystc_game_strategy` / `ystc_strategy_detail` / `ystc_control_command` 三张表的**事务级联动写入**，失败则全量回滚，保证数据一致性。

4.2.1 表 1：ystc_game_strategy

字段名	数据类型	取值逻辑	示例值
id	BIGINT	自增主键	1
user_id	BIGINT	当前落库的用户 ID	1001
strategy_name	VARCHAR	周期时间 + 用户名 + 博弈策略	周期 2025-12-12T14:35:16*用户 test*博弈策略
strategy_type	VARCHAR	固定值：博弈优化策略	博弈优化策略
algorithm_version	VARCHAR	固定值：1.0	1.0
start_time	DATETIME	周期起始时间（ISO 转时区后）	2025-12-12 14:35:16
end_time	DATETIME	周期起始时间 + CYCLE_INTERVAL（120 秒）	2025-12-12 14:37:16
time_slices	INT	时间片数量（config.TIME_SLOTS）	3
time_slice_interval	FLOAT	单时间片间隔（CYCLE_INTERVAL / TIME_SLOTS）	40.0
strategy_params	JSON	模型核心参数（迭代次数、耗时、收益、成本、利润）	{"iteration":100,"timeConsumption":0.5,...}
strategy_json	JSON	完整博弈结果（清理临时字段 `_db_device_id`，deviceId 替换为数据库主键 ID）	{"full_result":{"decisions":[{"deviceId":101,...}]}}
external_conditions	VARCHAR	固定值：默认市场电价 + 基准电网负荷	默认市场电价 + 基准电网负荷
is_active	TINYINT	固定值：1（激活）	1
status	VARCHAR	固定值：已生成	已生成
create_by	VARCHAR	当前用户名	test
create_time	DATETIME	当前时间（AUS_TZ 时区）	2025-12-12 14:36:00
update_by	VARCHAR	当前用户名	test
update_time	DATETIME	当前时间（AUS_TZ 时区）	2025-12-12 14:36:00

4.2.2 表 2：ystc_strategy_detail

字段名	数据类型	取值逻辑	示例值
id	BIGINT	自增主键	1
strategy_id	BIGINT	关联 ystc_game_strategy.id	1
time_slice_index	INT	时间片索引（0/1/2...）	0
time_point	DATETIME	周期起始时间 + 索引 × 时间片间隔	2025-12-12 14:35:16
action_type	VARCHAR	dc 值映射：1=charge，0=idle，-1=discharge，其他 = unknown	charge
power_setpoint	FLOAT	对应时间片的 speed 值	10.0
expected_price	FLOAT	对应时间片的 cost 值	0.5
expected_benefit	FLOAT	该设备总收益	80.0
create_by	VARCHAR	当前用户名	test
create_time	DATETIME	当前时间（AUS_TZ 时区）	2025-12-12 14:36:00
update_by	VARCHAR	当前用户名	test
update_time	DATETIME	当前时间（AUS_TZ 时区）	2025-12-12 14:36:00

4.2.3 表 3：ystc_control_command

字段名	数据类型	取值逻辑	示例值
id	BIGINT	自增主键	1
device_id	BIGINT	关联 ystc_device.id（数据库主键 ID，核心！）	101
strategy_id	BIGINT	关联 ystc_game_strategy.id	1
command_type	VARCHAR	dc 值映射：1=charge_exec，0=idle_exec，-1=discharge_exec，其他 = unknown_exec	charge_exec
command_params	JSON	命令参数（dc/speed/cost/benefit + 数据库 device_id）	{"dc": [1,0,-1],"speed": [10.0,0.0,8.0],...}
priority	INT	固定值：1	1
issued_at	DATETIME	当前时间（AUS_TZ 时区）	2025-12-12 14:36:00
scheduled_at	DATETIME	第一个时间片的起始时间	2025-12-12 14:35:16

字段名	数据类型	取值逻辑	示例值
expire_at	DATETIME	scheduled_at + 单时间片间隔	2025-12-12 14:35:56
executed_at	DATETIME	初始值：NULL（设备执行后更新）	NULL
status	VARCHAR	初始值：pending（待执行）	pending
result	JSON	初始值：NULL（执行后更新结果）	NULL
error_message	VARCHAR	初始值：NULL（执行失败时更新）	NULL
retry_count	INT	初始值：0	0
max_retries	INT	固定值：3	3
create_by	VARCHAR	当前用户名	test
create_time	DATETIME	当前时间（AUS_TZ 时区）	2025-12-12 14:36:00
update_by	VARCHAR	当前用户名	test
update_time	DATETIME	当前时间（AUS_TZ 时区）	2025-12-12 14:36:00

4.3 落库关键保障

- 事务控制：所有表写入包裹在数据库事务中，单表写入失败则全量回滚；
- 数据清理：落库前删除 `strategy_json` 中的 `_db_device_id` 临时字段，避免冗余；
- 权限校验：落库时验证 `device_id` 归属当前用户，防止跨用户数据写入；
- 日志溯源：输出策略 ID、用户 ID、设备 ID 等关键信息，便于问题定位。

4.4 内存状态更新

落库成功后，通过 `STORAGE_LOCK` 加锁更新内存状态：

```
# 博弈结果存入内存（供API查询）
DEVICE_STRATEGIES[cycle_time] = decisions
# 标记周期为完成
CYCLE_STATUS[cycle_time] = "completed"
```

周期调用的完整实现链路

从服务启动到单个周期执行，再到下一个周期的循环，完整调用链路如下：

1. **服务启动**：外部调用 `cycle_manager.py` 的 `start_cycle_service()`，安全启动 `service_loop` 主循环；
2. **周期初始化**：`service_loop` 初始化周期基准时间，等待设备注册完成；
3. **周期生成**：`service_loop` 按基准时间 + 周期间隔生成当前周期 ID，等待上传窗口关闭；
4. **周期执行触发**：调用 `run_cycle(cycle_time)`，检查设备数据后调用 `call_jar_model(cycle_time)`；
5. **博弈模型执行**：`call_jar_model` 完成数据预处理、JAR 调用、结果落库；

6. **周期收尾**：清理当前周期数据，`service_loop` 清理过期数据，计算下一个周期的开始时间，休眠对应时长后重复步骤 3-5；
7. **异常兜底**：任何环节抛出异常，都会重置标记 / 清理数据 / 重启服务，保证周期调度不中断。

API

> 本地测试请求地址（BASE_URL）：`http://localhost:8080`，所有接口均基于此地址拼接路径访问。

API 层基于 Flask Blueprint 实现模块化拆分，分为**认证模块（auth）、设备模块（device）、系统模块（system）** 三大核心模块，承接前端 / 设备侧请求，联动 Core 层的周期调度逻辑和 Model 层的数据库操作，实现 “用户 / 设备注册登录、设备数据上传、博弈策略查询、系统数据重置” 全流程能力。

核心设计原则

1. **权限管控**：核心业务接口（数据上传、策略查询、退出登录）均通过 `login_required` / `page_login_required` 装饰器校验登录状态，关联当前用户 / 设备（Flask g 对象存储）；
2. **数据一致性**：数据库操作均通过事务（commit/rollback）保障，异常时回滚；
3. **安全防护**：登录 Token 采用 HttpOnly Cookie 存储，防 XSS 攻击；死锁重试装饰器保障数据库操作稳定性；
4. **链路联动**：设备上传数据写入 Core 层的 `DEVICE_DATA` 内存字典，供周期调度逻辑读取；策略查询直接关联 Model 层的博弈结果表。

认证模块（auth_bp）

1. 模块概述

路由前缀：无（根路由），核心承接 “用户 + 设备” 的注册、登录、Token 校验、退出登录，关联 `YstcUser`、`ystc_device`、`UserAuthToken` 表，是所有业务接口的权限基础。

2. 核心接口

接口路径	请求方法	功能说明	关键逻辑
`/`	GET	根路由跳转登录页	重定向到 `/login` 页面
`/login`	GET	渲染登录页面	返回 login.html 模板
`/register`	GET	渲染注册页面	返回 register.html 模板
`/dashboard`	GET	渲染仪表盘页面	需 `page_login_required` 校验，返回当前用户 / 设备信息（g.user/g.device）

接口路径	请求方法	功能说明	关键逻辑
`/api/device/register`	POST	设备 + 用户注册接口	1. 校验账号 / 设备序列号唯一性； 2. 事务创建 YstcUser+Device 关联记录； 3. 返回用户 / 设备 ID
`/api/device/login`	POST	登录接口	1. 校验账号密码； 2. 生成 JWT Token（双存储：返回前端 + 写入 HttpOnly Cookie）； 3. 失效用户旧 Token，事务更新用户最后登录时间
`/api/device/verify_token`	POST	Token 校验接口	1. 解码 JWT Token 获取用户 ID； 2. 校验 Token 在 UserAuthToken 表中的有效性； 3. 返回用户 / 设备基础信息
`/api/device/logout`	POST	退出登录接口	1. 数据库标记当前 Token 无效； 2. 更新设备为离线状态； 3. 清除客户端 Cookie，返回登录页重定向指令

3. 关键设计

- **Token 双存储**：登录生成的 JWT Token 既通过 JSON 返回前端（可存 localStorage），又写入 HttpOnly Cookie（浏览器自动携带，防 XSS）；
- **死锁重试**：登录接口添加 `retry_on_deadlock` 装饰器，捕获数据库死锁异常（错误码 1213），最多重试 3 次；
- **事务保障**：注册 / 登录 / 退出操作均包裹在数据库事务中，异常时回滚，避免数据不一致。

设备模块（device_bp）

1. 模块概述

路由前缀：`/api/device`，核心承接 “设备数据上传、博弈策略查询、当前周期获取”，是 Core 层周期逻辑的 “数据输入” 和 “结果输出” 入口，关联 `ystc_game_strategy`、`ystc_strategy_detail`、`ystc_control_command` 表。

2. 核心接口

接口路径	请求方法	功能说明	关键逻辑
`/api/device/upload`	POST	设备数据上传接口	1. 需 `login_required` 校验； 2. 校验上传窗口是否开启（is_upload_window_open）； 3. 加锁写入 Core 层 `DEVICE_DATA`（按周期 + 设备序列号存储）

接口路径	请求方法	功能说明	关键逻辑
`/api/device/get_strategy`	GET	博弈策略查询接口	1. 按周期时间 + 当前用户 ID 查询 ystc_game_strategy 主表； 2. 关联查询 ystc_strategy_detail（时间片详情）、ystc_control_command（设备控制命令）； 3. 结构化返回策略数据
`/api/device/current_cycle`	GET	获取当前周期 ID 接口	调用 Core 层 `get_current_cycle()` 方法，返回标准化周期 ID（如 `2025-12-12T14:35:16.386067+11:00`）

3. 关键设计

- **数据隔离**：设备上传数据通过 `STORAGE_LOCK` 加锁写入 `DEVICE_DATA`，避免多设备并发上传导致的数据冲突；
- **周期联动**：上传接口校验 “上传窗口” 状态，仅在 Core 层允许的窗口期内接收数据，保证周期数据完整性；
- **权限过滤**：策略查询仅返回当前用户所属的博弈策略，通过 `g.user.id` 过滤，避免跨用户数据泄露。

系统模块（system_bp）

1. 模块概述

路由前缀：无，核心提供 “全量数据重置” 功能，用于测试 / 调试场景，级联删除所有业务表数据。**当前 `reset.html` 模板暂未实现，该功能是否纳入云端平台仍在评估中。**

2. 核心接口

接口路径	请求方法	功能说明	关键逻辑
`/reset`	GET	渲染重置页面	返回 reset.html 模板
`/reset`	POST	重置所有数据接口	1. 按外键依赖顺序级联删除数据 2. 事务保障，异常时回滚

补充说明

1. 所有 POST 接口需以 `application/json` 格式提交请求体；
2. 需登录的接口（标注 `login_required` / `page_login_required`）会自动校验 Cookie 中的 `access_token`，无需前端手动在请求头携带；
3. 本地开发环境下，Flask 服务通过 `0.0.0.0:8080` 暴露，可通过 `localhost:8080` 或服务器内网 IP（如 `192.168.1.100:8080`）访问；
4. 禁用了 Flask 的 `debug` 和 `use_reloader` 模式，避免周期后台服务重复启动。

TEMPLATES

模板概述

模板基于纯 HTML + JavaScript 实现（无前端框架依赖），对应认证模块的页面路由（`/login`/`/register`/`/dashboard`），核心实现用户交互、Token 本地管理、页面权限控制。

模板文件清单

模板文件	访问路径	功能说明	核心设计
`login.html`	`http://localhost:8080/login`	设备登录页面	1. 页面加载时自动校验 localStorage 中的 Token，有效则直接跳转仪表盘； 2. 登录表单提交后存储 Token 到 localStorage； 3. 异常处理：网络错误 / Token 无效时友好提示并显示登录表单
`register.html`	`http://localhost:8080/register`	设备注册页面	1. 完整的注册表单，包含必填项（用户名 / 密码 / 设备名称 / SN 码 / 设备类型）和选填项（手机号 / 邮箱 / 地址）； 2. 表单提交前做基础校验，提交后根据接口返回提示结果； 3. 注册成功后自动跳转登录页；
`dashboard.html`	`http://localhost:8080/dashboard`	设备仪表盘页面	1. 页面加载时校验 Token 有效性，无效则跳转登录页； 2. 展示当前登录用户 / 设备序列号、设备状态、当前周期； 3. 异步获取当前周期数据并渲染； 4. 退出登录按钮：调用退出接口，清除 localStorage Token 并跳转登录页；

模板文件	访问路径	功能说明	核心设计
`reset.html`	`http://localhost:8080/reset`	数据重置页面	暂未实现，该页面计划用于测试 / 调试场景，提供可视化的全量数据重置确认入口，功能是否落地需结合云端平台规划评估

核心交互逻辑

1. Token 管理

- **存储**：登录成功后，将接口返回的 `access_token` 存入 `localStorage`（`localStorage.setItem('access_token', token)`）；
- **校验**：所有需要登录的页面（`dashboard.html` / `login.html`）加载时，先读取 `localStorage` 中的 Token，调用 `/api/device/verify_token` 接口校验有效性；
- **失效处理**：Token 无效 / 不存在时，清除 `localStorage` 中的 Token，自动跳转登录页；
- **退出**：点击退出登录按钮时，调用 `/api/device/logout` 接口，清除 `localStorage` 中的 Token，跳转登录页。

2. 表单交互（login.html/register.html）

- 禁用表单默认提交行为，采用异步 `fetch` 提交数据；
- 请求头固定为 `Content-Type: application/json`，符合后端接口要求；
- 异常提示：网络错误 / 接口返回错误时，在页面内显示友好提示文本；
- 注册表单做空值处理：选填项为空时赋值 `undefined`，避免传递空字符串到后端。

3. 仪表盘数据渲染（dashboard.html）

- 页面加载完成后异步调用 `/api/device/current_cycle` 接口获取当前周期；
- 周期数据加载失败时显示 “获取失败”，保证页面体验；
- 设备状态默认显示 “在线”，后续可扩展对接设备状态接口；
- 用户名 / 设备序列号通过 Flask 模板变量（`{{ username }}` / `{{ serial_number }}`）从后端传入。

CONFIG

核心设计原则

1. **多环境适配**：优先读取系统环境变量（生产环境），其次读取 `.env` 文件（本地开发），最后使用代码默认值，避免硬编码；
2. **路径统一管理**：自动拼接项目根目录下的 `data` / `logs` 等目录路径，避免路径硬编码；
3. **配置集中化**：数据库、JWT、周期调度等核心配置全部收敛到 `Config` 类，便于维护和修改；

4. **无侵入扩展**：新增配置项只需在 `Config` 类中添加，其他模块通过导入 `config` 实例即可使用，无需修改业务代码。

配置加载优先级

```
系统环境变量（生产部署） > .env 文件（本地开发） > 代码默认值
```

示例：若系统环境变量中配置了 `DB_PASSWORD=prod123`，同时 `.env` 文件中配置了 `DB_PASSWORD=dev123`，最终生效的是 `prod123`。

核心配置项说明

1. 数据库配置

配置项	含义	默认值	配置来源
`DB_HOST`	数据库主机地址	`127.0.0.1`	环境变量 /.env/ 默认值
`DB_PORT`	数据库端口	`3306`	环境变量 /.env/ 默认值
`DB_USER`	数据库用户名	`root`	环境变量 /.env/ 默认值
`DB_PASSWORD`	数据库密码	无（必须配置）	环境变量 /.env（本地开发）
`DB_NAME`	数据库名	`ystc`	环境变量 /.env/ 默认值
`SQLALCHEMY_DATABASE_URI`	数据库连接 URL	自动拼接	基于上述参数动态生成（无需手动配）

2. JWT 认证配置

配置项	含义	默认值	配置来源
`JWT_SECRET_KEY`	JWT 加密密钥	`device-auth-secret-永久有效`	环境变量 /.env/ 默认值
（备注）	Token 有效期	无限制（代码层面移除有效期）	-

3. 周期调度配置

配置项	含义	默认值	配置来源
`CYCLE_INTERVAL`	周期调度总间隔	120 秒（2 分钟）	环境变量 /.env/ 默认值
`UPLOAD_WINDOW`	数据上传窗口期	20 秒	环境变量 /.env/ 默认值
`TIME_SLOTS`	单次周期内时间片数量	3	环境变量 /.env/ 默认值

4. 路径配置

配置项	含义	生成规则
`BASE_DIR`	项目根目录	自动计算（基于 `config.py` 所在路径）
`DATA_DIR`	数据存储目录	`BASE_DIR + /data`
`LOG_DIR`	日志存储目录	`BASE_DIR + /logs`

5. 时区配置

配置项	含义	默认值	配置来源
`TZ`	项目时区	`Australia/Melbourne`	环境变量 /.env/ 默认值

UTILS

核心设计原则

1. **复用性**：所有工具函数 / 装饰器均为通用能力，不耦合具体业务逻辑；
2. **线程安全**：周期数据操作通过 `STORAGE_LOCK` 线程锁保障多设备并发安全；
3. **配置联动**：所有工具均依赖 `config.py` 全局配置，支持环境动态适配；
4. **异常可控**：关键操作（鉴权、数据库、周期计算）均包含完整的异常捕获和日志输出；
5. **易用性**：通过 `__init__.py` 统一导出核心工具，其他模块一键导入即可使用。

核心子模块详解

1. 鉴权工具（auth.py）

封装 `login_required`（API 接口鉴权）和 `page_login_required`（页面访问鉴权）两个装饰器，核心实现 Token 校验、用户 / 设备关联、权限管控，是所有需登录接口 / 页面的基础依赖。

1.1 核心装饰器说明

装饰器名称	适用场景	核心逻辑
`login_required`	API 接口（如 `/api/device/upload`）	1. 从请求头 `Authorization` 获取 Bearer Token； 2. 解码 Token（关闭过期校验），验证 Token 哈希在 `UserAuthToken` 表中的有效性； 3. 校验用户 / 设备是否存在，更新设备在线状态； 4. 将 `user` / `device` 存入 Flask `g` 对象； 5. 异常时返回 JSON 格式的 401/500 错误

装饰器名称	适用场景	核心逻辑
`page_login_required`	页面路由（如 `/dashboard`）	1. 优先从请求头、其次从 Cookie 获取 Token； 2. 同 Token 有效性校验逻辑； 3. 无效 / 异常时重定向到登录页（而非返回 JSON）； 4. 有效则将 `user` / `device` 存入 `g` 对象供模板渲染

1.2 关键特性

- Token 校验规则：基于 `UserAuthToken` 表的 `token_hash` + `is_valid` 字段，确保 Token 未被注销；
- 时区适配：设备最后在线时间使用 `config.TZ` 配置的时区（默认 `Australia/Melbourne`）；
- 自动失效：若用户 / 设备不存在，自动标记 Token 为无效并提交数据库；
- 异常处理：捕获 `InvalidTokenError`（Token 解析失败）和通用异常，分别返回标准化错误信息。

2. 周期调度工具（cycle.py）

封装周期调度相关的全局变量和工具函数，支撑 Core 层周期逻辑和 Device 模块数据上传 / 策略查询，核心解决「周期标识生成、上传窗口校验、非持久化数据清理」问题。

2.1 全局变量说明

变量名	类型	用途	线程安全
`STATE`	dict	周期服务状态存储（是否启动、最后周期起止时间、最后错误信息）	否（仅用于状态记录）
`DEVICE_DATA`	dict	非持久化设备数据存储（结构：`{cycle_time: {serial_number: device_data}}`）	是（需通过 `STORAGE_LOCK` 操作）
`DEVICE_STRATEGIES`	dict	非持久化策略数据存储（结构：`{cycle_time: {device_id: strategy}}`）	是（需通过 `STORAGE_LOCK` 操作）
`CYCLE_STATUS`	dict	周期状态存储（结构：`{cycle_time: "running"/"completed"/"failed"}`）	是（需通过 `STORAGE_LOCK` 操作）
`STORAGE_LOCK`	threading.Lock	线程锁，保障多线程 / 多设备并发操作上述字典时的数据一致性	-

2.2 核心函数说明

函数名	功能说明	入参	返回值	关键逻辑
`get_current_cycle`	生成当前周期时间标识	无	str（ISO格式时间）	1. 基于基准时间戳和 `config.CYCLE_INTERVAL`（默认 120 秒）计算当前周期起始时间；2. 转换为 `config.TZ` 时区的 ISO 格式字符串（如 `2025-12-12T14:35:16.386067+11:00`）；3. 基准时间未初始化时自动补充
`is_upload_window_open`	判断当前周期上传窗口是否开启	cycle_time（周期标识）	bool	计算当前时间是否在「周期起始时间 + `config.UPLOAD_WINDOW`（默认 20 秒）」内
`clean_expired_data`	清理过期周期数据	无	无	保留最近 1 个周期数据，删除其他周期的 `DEVICE_DATA` / `DEVICE_STRATEGIES` / `CYCLE_STATUS`
`clean_cycle_data`	清理指定周期数据	cycle_time（周期标识）	无	精准删除指定周期的所有非持久化数据，用于周期结束后立即清理

3. 数据库工具（db.py）

封装数据库引擎初始化、会话管理、表创建等核心能力，是 Model 层操作数据库的基础，统一管理数据库连接池、字符集、异常处理。

3.1 核心组件说明

组件名	类型	功能说明
`Base`	declarative_base()	SQLAlchemy 基础模型类，所有数据模型（如 `YstcUser` / `Device`）均继承此类

组件名	类型	功能说明
`engine`	create_engine()	数据库连接引擎，配置：1. 连接池（pool_size=20，max_overflow=30）；2. 预检测连接有效性（pool_pre_ping=True）；3. 字符集 utf8mb4；4. 关闭 SQL 日志（echo=False）
`SessionLocal`	sessionmaker()	数据库会话工厂，每次请求生成独立会话，需手动关闭（或通过 `get_db` 自动关闭）
`init_db()`	函数	1. 测试数据库连接；2. 基于 `Base.metadata` 创建所有表；3. 异常时输出精准错误日志（如账号密码错误、数据库不存在）
`get_db()`	生成器函数	依赖注入用会话获取函数，自动管理会话的创建和关闭（适配 FastAPI 风格，Flask 也可复用）