

自然写教室智能网关管理软件 V1.0

软件著作权鉴别材料（嵌入式软件设计说明书）

项目	内容
软件全称	自然写教室智能网关管理软件
软件简称	自然写网关软件
版本号	V1.0
权利人	深圳自然写科技有限公司
开发语言	C / C++ / Python
运行环境	嵌入式Linux（网关硬件设备）
文档类型	嵌入式软件设计说明书
编制日期	2026年2月

目录

- 第一章 软件整体概述
 - 1.1 软件简介与功能综述
 - 1.2 软件用途与适用场景
 - 1.3 运行环境与硬件要求
 - 1.4 开发语言与技术规范
 - 1.5 版本说明
- 第二章 系统架构与设计思路
 - 2.1 总体架构设计
 - 2.2 各层次详细说明
 - 2.3 数据流设计
 - 2.4 数据存储设计
 - 2.5 通信协议设计
 - 2.6 安全设计
 - 2.7 部署架构

- 第三章 核心模块功能详细说明
 - 3.1 蓝牙多笔连接管理模块
 - 3.2 笔迹数据实时接收与缓存模块
 - 3.3 通信协议转换模块
 - 3.4 数据压缩与加密上传模块
 - 3.5 断网离线缓存与自动续传模块
 - 3.6 设备发现与自动配对模块
 - 3.7 OTA固件远程升级模块
 - 3.8 设备状态上报与心跳监测模块
 - 3.9 本地Web管理界面模块
- 第四章 操作流程与使用步骤
 - 4.1 网关设备安装与初始化
 - 4.2 网络配置操作流程
 - 4.3 蓝牙笔连接与配对操作流程
 - 4.4 本地Web管理界面使用
 - 4.5 OTA固件升级操作流程
 - 4.6 故障诊断与维护
- 第五章 与源代码的对应关系
 - 5.1 模块与源代码文件对应表
 - 5.2 核心函数说明
 - 5.3 命名规范
- 附录

第一章 软件整体概述

1.1 软件简介与功能综述

自然写教室智能网关管理软件（以下简称"网关软件"）是运行于自然写教室智能网关硬件设备上的嵌入式Linux软件，是自然写互动课堂系统中负责教室本地通信枢纽的核心软件。

网关软件运行于ARM架构的嵌入式Linux操作系统（如OpenWrt或定制化Ubuntu Server）上，通过蓝牙BLE 5.0协议同时连接教室内多达40支智能点阵笔，实时接收笔迹坐标数据，进行本地协议转换和数据打包后，通过MQTT over TLS协议上传至云平台。同时，网关软件向教室内的各终端设备（大屏、PC、Pad）提供局域网内的笔迹实时推送服务。

主要功能模块：

- (1) 蓝牙多笔连接管理：基于BlueZ蓝牙协议栈，实现BLE 5.0多连接并发管理，支持最多40支点阵笔同时连接，自动处理笔的连接、断开和重连事件。
- (2) 笔迹数据实时接收与缓存：通过BLE GATT Notification通道实时接收每支笔的坐标数据包，写入内存环形缓冲区，防止突发数据丢失。
- (3) 通信协议转换：将蓝牙BLE自定义协议的笔迹数据包解码，转换为云平台MQTT接口规定的标准JSON/Protobuf格式。
- (4) 数据压缩与加密上传：对打包好的笔迹数据进行Gzip压缩，通过MQTT over TLS加密传输至云平台，确保数据传输效率和安全性。
- (5) 断网离线缓存与续传：在网络中断期间，将笔迹数据持久化存储至本地Flash（SQLite数据库），网络恢复后自动按顺序补传缓存数据，确保数据不丢失。
- (6) 设备发现与自动配对：通过BLE Advertising扫描发现附近的自然写点阵笔，根据设备名称前缀（"Writech-Pen-"）识别并发起配对连接请求。
- (7) OTA固件远程升级：接收云平台下发的OTA升级指令，通过HTTPS下载固件升级包，验证签名后写入备用分区，重启后自动切换运行新版本。
- (8) 设备状态上报：定期通过MQTT上报网关自身状态（CPU温度、内存使用率、已连接笔数量、网络延迟等），供云平台监控和运维。
- (9) 本地Web管理界面：基于lighttpd Web服务器提供轻量级本地管理页面，管理员可在浏览器中查看网关状态、配置WiFi、查看已连接设备列表等。

1.2 软件用途与适用场景

网关软件是自然写互动课堂系统中不可或缺的教室本地基础设施组件，适用于以下场景：

- (1) 标准教室互动课堂部署：在配备自然写设备的教室中，每间教室安装1台网关设备，通过本软件连接教室内所有学生的点阵笔，实现全班书写数据的实时采集和上传。
- (2) 网络受限场景下的离线教学：在校园网络不稳定或有限制的环境中，网关软件的离线缓存功能确保课堂上采集的笔迹数据不会因网络中断而丢失，待网络恢复后自动补传。
- (3) 局域网内低延迟数据分发：在需要实时大屏展示学生书写内容的场景中，网关通过局域网直接向大屏或教师PC推送笔迹数据，延迟低于50ms，满足课堂互动的实时性要求。

1.3 运行环境与硬件要求

硬件平台规格：

组件	规格
处理器	ARM Cortex-A53 四核 @1.5GHz（或同等性能）
内存	512MB DDR4（最低） / 1GB DDR4（推荐）
存储	4GB eMMC（系统） + 可选外部存储
蓝牙模块	支持BLE 5.0，同时连接≥40个设备
网络接口	100Mbps以太网 + 802.11ac WiFi（2.4G/5G双频）
USB接口	USB 2.0 × 2（调试和外设扩展）

软件运行环境：

组件	版本要求
Linux内核	5.10+（需支持蓝牙BLE多连接）
BlueZ	5.65+（蓝牙协议栈）
Python	3.9+（业务逻辑层）
SQLite	3.39+（本地数据存储）
Eclipse Mosquitto	2.0+（MQTT客户端库）
lighttpd	1.4.72+（本地Web服务器）

1.4 开发语言与技术规范

各模块开发语言分工：

模块	语言	说明
硬件抽象层（HAL）驱动	C	蓝牙芯片驱动、网络接口配置、GPIO控制
通信协议层	C / C++	BlueZ BLE连接管理、lwIP TCP/IP、MQTT客户端
业务逻辑层	C++ / Python	数据缓存调度、设备管理、协议转换主逻辑
本地Web管理	Python（CGI/FastCGI）	lighttpd + Python实现管理页面后端

模块	语言	说明
配置管理	Python	配置文件读写、参数校验

编码规范：

- C/C++代码遵循Google C++ Style Guide
- Python代码遵循PEP 8，使用Black格式化
- 所有函数须有注释说明功能、参数和返回值
- 关键路径（BLE数据接收、MQTT发送）使用非阻塞IO，避免主循环阻塞

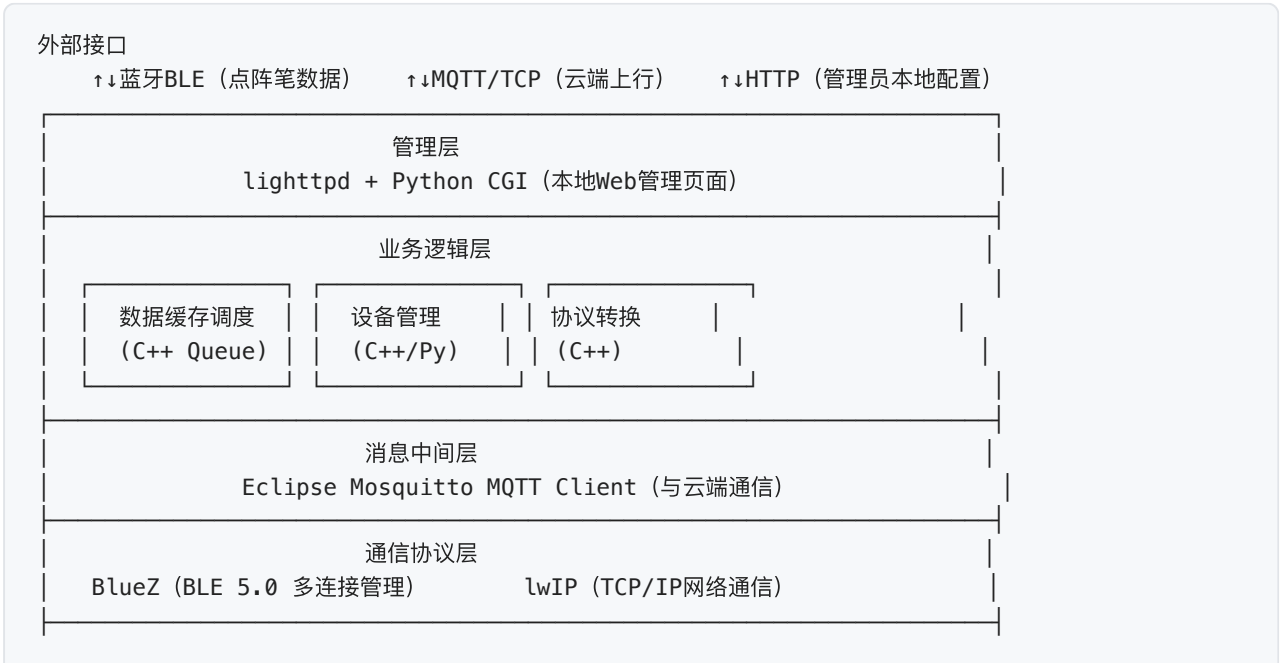
1.5 版本说明

版本号	发布日期	说明
V1.0	2026年2月	初始版本，支持40笔并发、离线缓存、OTA升级等完整功能

第二章 系统架构与设计思路

2.1 总体架构设计

网关软件采用嵌入式分层架构，从底层到顶层依次为：硬件抽象层（HAL）、通信协议层、消息中间层、业务逻辑层和管理层。各层职责明确，层间通过定义良好的接口通信，便于单独测试和替换。





2.2 各层次详细说明

硬件抽象层 (HAL):

HAL层封装所有硬件相关的操作，提供统一的软件接口给上层模块，屏蔽具体硬件的差异。主要包括：

蓝牙芯片驱动：基于HCI（主机控制器接口）与蓝牙芯片通信，初始化蓝牙控制器，配置广播参数，建立ACL连接。当使用不同厂商的蓝牙芯片时，只需替换HAL层的驱动实现，上层业务代码无需修改。

WiFi模块驱动：封装WiFi连接管理（SSID/密码配置、自动重连、信号强度查询），通过nl80211内核接口实现。

状态指示LED控制：通过sysfs或GPIO字符设备控制状态LED的亮灭和闪烁模式，用于指示网关的运行状态。

通信协议层:

通信协议层处理所有网络通信的底层细节：

BlueZ BLE多连接管理：BlueZ是Linux官方蓝牙协议栈，网关软件通过BlueZ的DBus API管理BLE连接。每建立一个与点阵笔的BLE连接，创建一个对应的GATT客户端实例，订阅笔迹数据特征值（Characteristic）的Notification通知。

MQTT客户端（Eclipse Mosquitto libmosquitto）：使用libmosquitto库实现MQTT通信，配置TLS证书进行加密连接，建立与云平台MQTT Broker的持久会话（Clean Session = false），确保离线期间积压的消息不会丢失。

业务逻辑层:

数据缓存调度模块：维护一个多生产者（多支笔的数据接收线程）单消费者（MQTT上传线程）的线程安全环形缓冲区。缓冲区满时采用丢弃最旧数据的策略，保证实时性。

设备管理模块：维护已配对笔的设备列表，记录每支笔的MAC地址、当前连接状态、最后活跃时间和电量。提供设备注册、更新和查询接口给上层模块。

协议转换模块：将BLE接收到的自定义二进制格式笔迹数据包解析为内部结构体，再序列化为符合云平台规范的JSON或Protobuf格式。

2.3 数据流设计

完整数据流路径：

阶段1：BLE数据接收
点阵笔 → (BLE GATT Notification) → BlueZ接收线程
→ 解包：提取设备ID + 坐标数据帧序列
→ 写入内存环形缓冲区 (Ring Buffer)

阶段2：数据调度与处理
Ring Buffer 读取线程
→ 按设备ID汇聚相同笔的数据帧
→ 协议转换 (自定义二进制 → 标准JSON/Protobuf格式)
→ 数据压缩 (Gzip)
→ 判断网络状态：
 若在线 → 写入MQTT发送队列
 若离线 → 写入SQLite离线缓存队列

阶段3：云端上传
MQTT发送线程
→ 从发送队列取出数据包
→ 发布至Topic: pen/{gateway_id}/stroke
→ 等待MQTT PUBACK确认
→ 确认收到后从队列删除

阶段4：离线恢复
断网检测线程
→ 检测到网络恢复
→ 读取SQLite离线缓存 (按时间戳顺序)
→ 逐批发送至MQTT队列
→ 发送成功后删除SQLite中的记录

局域网实时推送支持（附加数据流）：

Ring Buffer → 局域网推送模块
→ 维护已连接的局域网终端WebSocket列表
→ 向每个已连接的终端（大屏/PC/Pad）实时广播笔迹数据帧
→ 广播失败的终端从列表中移除（等待重连）

2.4 数据存储设计

内存数据（进程运行期间）：

数据	存储方式	容量	说明
笔迹数据缓冲区	内存环形Buffer	2MB	BLE→MQTT的临时缓冲，丢失可接受

数据	存储方式	容量	说明
设备连接状态	内存哈希表 (C++ unordered_map)	极少	MAC地址→连接状态映射, 快速查找
MQTT发送队列	内存FIFO队列	最多 4MB	等待发送的数据包, 有序发送

持久化数据 (Flash存储):

数据	存储方式	容量	说明
离线笔迹缓存	SQLite数据库 (stroke_cache.db)	最大 64MB	断网期间的笔迹数据持久化
已配对设备列表	SQLite数据库 (devices.db)	极少	笔的MAC地址、名称、绑定学生ID
网关配置	JSON配置文件 (/etc/writtech/config.json)	极少	WiFi、MQTT服务器、心跳间隔等
OTA升级包缓存	Flash A/B分区	各32MB	固件双分区存储, 升级失败可回滚
系统日志	轮转日志 (logrotate, /var/log/writtech/)	最大 50MB	运行日志、错误日志, 按日期轮转

SQLite离线缓存表设计 (stroke_cache):

```
CREATE TABLE stroke_cache (  
  id          INTEGER PRIMARY KEY AUTOINCREMENT,  
  seq_no      INTEGER NOT NULL,           -- 数据包序列号 (全局递增)  
  pen_mac     TEXT NOT NULL,             -- 来源笔MAC地址  
  student_id  INTEGER,                   -- 关联学生ID (若已绑定)  
  data_blob   BLOB NOT NULL,             -- Protobuf格式压缩数据  
  data_size   INTEGER NOT NULL,          -- 数据大小 (字节)  
  created_at  INTEGER NOT NULL,          -- 写入时间 (Unix时间戳)  
  is_uploaded INTEGER DEFAULT 0          -- 是否已上传 (0待上传/1已上传)  
);  
CREATE INDEX idx_stroke_cache_uploaded ON stroke_cache(is_uploaded, seq_no);
```

2.5 通信协议设计

BLE GATT 服务定义 (点阵笔端):

点阵笔使用自定义GATT Service暴露笔迹数据:

项目	UUID	权限	说明
笔迹数据Service	0xFF10	–	点阵笔主服务
笔迹数据Characteristic	0xFF11	Notify	笔迹坐标数据通知，MTU=247字节
设备信息Characteristic	0xFF12	Read	固件版本/序列号/电量
配置写入Characteristic	0xFF13	Write	写入配置参数（功耗模式/采样率）
OTA控制Characteristic	0xFF20	Notify/Write	OTA升级控制通道

MQTT Topic设计：

Topic	方向	QoS	说明
pen/{gateway_id}/stroke	网关→ 云端	1（至少一 次）	笔迹数据上报
gateway/{gateway_id}/status	网关→ 云端	0（最多一 次）	心跳状态上报（CPU/内存/连接笔数）
gateway/{gateway_id}/command	云端→ 网关	1	云端下行指令（重启/配置更新/OTA触发）
gateway/{gateway_id}/ota/progress	网关→ 云端	1	OTA升级进度上报

笔迹数据MQTT消息格式（Protobuf定义）：

```
message StrokeUploadMessage {
    string gateway_id = 1;           // 网关设备ID
    string pen_mac = 2;              // 点阵笔MAC地址
    int64 student_id = 3;           // 关联学生ID
    int32 page_id = 4;              // 点阵纸张页面ID
    int64 timestamp = 5;            // 数据采集时间戳（毫秒）
    uint32 seq_no = 6;              // 序列号（用于去重和补传验证）
    repeated StrokeFrame frames = 7; // 坐标帧列表（每包最多50帧）
}

message StrokeFrame {
    int32 x = 1;                    // X坐标（0.01mm单位）
    int32 y = 2;                    // Y坐标
    uint32 pressure = 3;            // 笔压（0-255）
    uint32 timestamp_offset = 4;    // 相对于消息timestamp的毫秒偏移
    bool pen_up = 5;                // 是否抬笔
}
```

2.6 安全设计

通信安全：

MQTT通信使用TLS 1.3加密： – 服务端证书：云平台MQTT Broker的TLS证书（CA机构签发） – 客户端证书：每台网关设备在出厂时预置唯一的X.509设备证书（设备证书与序列号绑定） – 双向认证（mTLS）：MQTT Broker验证网关的设备证书，防止非法设备伪装接入

BLE通信安全： – 采用BLE LE Secure Connections配对模式（ECDH密钥交换） – 配对过程采用Numeric Comparison方式，防止中间人攻击 – 配对完成后所有BLE通信使用128位AES-CCM加密

OTA安全：

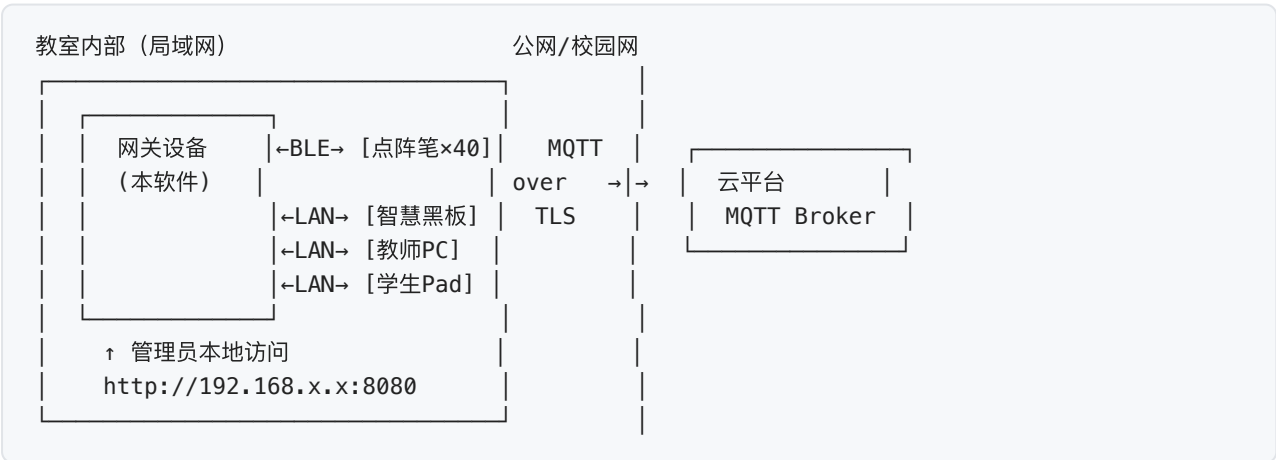
OTA升级包在下载和写入Flash前须通过以下验证： – HTTPS下载：确保固件包在传输过程中不被篡改 – RSA-2048签名验证：固件包附带制造商私钥签名，网关使用内置公钥验证签名合法性 – SHA-256哈希校验：验证固件包内容完整性 – A/B分区保护：新固件写入备用分区，仅在验证完全通过后才切换引导，失败则继续使用当前运行分区

设备认证：

网关首次上线时，通过设备证书向云平台的设备认证服务（Device Auth Service）完成注册激活，获取运行时访问凭证（Access Token）。Access Token存储于加密的Flash分区，每24小时自动刷新。

2.7 部署架构

教室级部署：



双分区引导设计：

Flash存储布局：

Bootloader (8MB) | App分区A (32MB) | App分区B (32MB) | 数据分区

↑当前运行

↑OTA升级目标

启动流程：

- Bootloader读取分区标志（存储于NVS）
- 根据标志选择启动分区A或分区B
- 软件启动成功后，设置"启动确认"标志
- 若启动3次仍未设置确认标志，Bootloader自动切回上一个有效分区

第三章 核心模块功能详细说明

3.1 蓝牙多笔连接管理模块

模块文件： ble/ble_manager.c

功能概述：

BLE多笔连接管理模块是网关软件中最关键的模块之一，负责管理与多达40支点阵笔的并发BLE连接，处理连接建立、数据接收和连接断开等BLE生命周期事件。

BLE多连接实现方案：

Linux的BlueZ蓝牙协议栈通过DBus接口提供BLE中央角色（Central）的API，网关软件作为BLE中央设备，同时连接多个点阵笔（外设设备）。

实现40笔并发连接的关键技术点：

(1) 连接参数优化：蓝牙BLE的连接参数直接影响并发连接数和数据传输速率。网关将每个BLE连接的Connection Interval配置为30ms~60ms，保证在连接间隙内有足够的时隙服务其他连接。

(2) PDU优化：启用BLE 5.0的LE 2M PHY（2Mbps物理层），相比1M PHY提升一倍的吞吐量，允许在相同时间内服务更多连接。

(3) 多线程处理：每个BLE连接的数据接收使用独立的处理线程（或通过libuv事件循环实现异步IO），避免单个笔的数据处理阻塞其他笔。

连接管理状态机：

初始状态 (Disconnected)
↓ 扫描到广播包 + 设备名称匹配"Writech-Pen-"
发现状态 (Discovered)
↓ 发起连接请求 (GATT Connect)
连接中 (Connecting)
↓ 连接成功
已连接 (Connected)
↓ 订阅笔迹数据Characteristic的Notification
已订阅 (Subscribed, 数据接收中)
↓ 连接断开 (超时/用电关笔/距离过远)
重连等待 (Reconnecting, 延迟3秒后重试)
↓ 重连成功
已订阅 (Subscribed)

关键实现：BLE连接数上限控制

```
// ble/ble_manager.c 核心结构 (伪代码)
#define MAX_PEN_CONNECTIONS 40

typedef struct {
    char mac_addr[18];           // BLE设备MAC地址
    DBusConnection *dbus_conn;   // BlueZ DBus连接句柄
    GattClientHandle gatt;       // GATT客户端句柄
    uint8_t battery_level;       // 最新电量
    uint64_t last_data_ts;       // 最后收到数据时间戳
    bool is_connected;           // 当前连接状态
    uint32_t receive_count;      // 累计接收帧数
} PenConnection;

static PenConnection pen_pool[MAX_PEN_CONNECTIONS];
static int active_connections = 0;
static pthread_mutex_t pool_lock = PTHREAD_MUTEX_INITIALIZER;

// 接受新连接 (在连接数未达上限时)
int ble_accept_connection(const char *mac_addr) {
    pthread_mutex_lock(&pool_lock);
    if (active_connections >= MAX_PEN_CONNECTIONS) {
        pthread_mutex_unlock(&pool_lock);
        return -1; // 拒绝, 已达最大连接数
    }
    // 在pool中找到空闲槽位, 初始化PenConnection结构
    // ...
    active_connections++;
    pthread_mutex_unlock(&pool_lock);
    return slot_index;
}
```

3.2 笔迹数据实时接收与缓存模块

模块文件： `cache/ring_buffer.c` 、 `cache/stroke_receiver.c`

功能概述：

笔迹数据接收模块负责从BLE Notification回调中接收笔迹数据包，写入线程安全的内存环形缓冲区，供上层数据处理模块消费。

环形缓冲区设计：

内存环形缓冲区是解决BLE高速数据接收与MQTT相对低速上传之间速率不匹配问题的关键数据结构。

```
// cache/ring_buffer.c 核心设计
#define RING_BUFFER_SIZE (2 * 1024 * 1024) // 2MB缓冲区

typedef struct {
    uint8_t buffer[RING_BUFFER_SIZE];
    int write_pos;           // 写指针
    int read_pos;           // 读指针
    int data_size;          // 当前数据量
    pthread_mutex_t lock;   // 读写互斥锁
    pthread_cond_t not_empty; // 数据可读条件变量
    pthread_cond_t not_full; // 缓冲区未满足条件变量
} RingBuffer;

// 写入函数（BLE接收回调线程调用）
int ring_buffer_write(RingBuffer *rb, const uint8_t *data, int len) {
    pthread_mutex_lock(&rb->lock);
    // 若缓冲区已满，等待或覆盖最旧数据（根据配置）
    while (rb->data_size + len > RING_BUFFER_SIZE) {
        // 超时等待100ms，若仍满则覆盖最旧数据（保证实时性优先）
        // ...
    }
    // 循环写入数据
    // ...
    pthread_cond_signal(&rb->not_empty);
    pthread_mutex_unlock(&rb->lock);
    return 0;
}
```

BLE Notification数据格式（点阵笔输出）：

每个BLE Notification数据包最大247字节（MTU=247），包含多个坐标帧：

字节偏移	长度	字段
0	1	协议版本（固定0x01）
1	1	帧数量N（本包含有N个坐标帧）
2	2	包序号（uint16，用于检测丢包）
4	N×7	坐标帧数组（每帧7字节）

帧格式：

0~1: X坐标（uint16，0.01mm单位）

2~3: Y坐标（uint16）

4: 压力值 (uint8, 0-255)
5~6: 时间偏移 (uint16, 相对于包起始时间的毫秒偏移)

3.3 通信协议转换模块

模块文件: `protocol/protocol_converter.c`

功能概述:

协议转换模块将BLE自定义二进制格式的笔迹数据包解码, 并序列化为符合云平台MQTT接口规范的Protobuf格式数据包。

转换流程:

```
// protocol/protocol_converter.c 核心转换逻辑 (伪代码)

StrokeUploadMessage* convert_ble_to_mqtt(
    const PenBlePacket *ble_pkt,          // BLE接收的原始数据包
    const PenConnection *pen_info,        // 笔的连接信息 (MAC/绑定学生)
    const GatewayConfig *gw_config        // 网关配置 (网关ID等)
) {
    StrokeUploadMessage *msg = stroke_upload_message_create();

    // 1. 填写消息头部信息
    msg->gateway_id = gw_config->gateway_id;
    msg->pen_mac = pen_info->mac_addr;
    msg->student_id = pen_info->bound_student_id;
    msg->page_id = ble_pkt->page_id;      // 从BLE包中解析点阵页面ID
    msg->timestamp = get_current_ms();
    msg->seq_no = ble_pkt->seq_no;

    // 2. 转换坐标帧列表
    for (int i = 0; i < ble_pkt->frame_count; i++) {
        StrokeFrame *frame = stroke_frame_create();
        frame->x = ble_pkt->frames[i].x;
        frame->y = ble_pkt->frames[i].y;
        frame->pressure = ble_pkt->frames[i].pressure;
        frame->timestamp_offset = ble_pkt->frames[i].ts_offset;
        frame->pen_up = ble_pkt->frames[i].pen_up;
        stroke_upload_message_add_frames(msg, frame);
    }

    return msg; // 调用方负责序列化为二进制Protobuf并释放内存
}
```

3.4 数据压缩与加密上传模块

模块文件: `mqtt/mqtt_client.c`

功能概述：

MQTT上传模块负责将处理好的笔迹数据包通过MQTT over TLS协议安全可靠地上传至云平台。

数据压缩策略：

笔迹坐标数据具有较高的压缩率（坐标值相邻帧差异较小），使用Gzip压缩后平均可减少60%以上的数据量：

```
// 压缩并发送数据包
int compress_and_publish(MqttClient *client,
                        const uint8_t *protobuf_data,
                        int data_len,
                        const char *topic) {

    // 1. Gzip压缩
    uint8_t *compressed = malloc(data_len);
    uLong compressed_len = data_len;
    int ret = compress2(compressed, &compressed_len,
                        protobuf_data, data_len, Z_BEST_SPEED);

    // 2. MQTT发布
    mosquitto_publish(client->mosq, NULL, topic,
                      compressed_len, compressed,
                      MQTT_QOS_1, false);

    free(compressed);
    return ret;
}
```

MQTT连接配置：

```
// MQTT连接参数配置
#define MQTT_KEEPALIVE_SEC    60    // 心跳间隔60秒
#define MQTT_CLEAN_SESSION    0    // 不使用Clean Session, 保留离线消息
#define MQTT_QOS_STROKE      1    // 笔迹数据QoS=1（至少一次）
#define MQTT_QOS_STATUS      0    // 心跳状态QoS=0（最多一次）

// TLS配置
mosquitto_tls_set(mosq,
                  "/etc/wrotech/certs/ca.crt",    // CA根证书
                  NULL,
                  "/etc/wrotech/certs/device.crt", // 设备证书
                  "/etc/wrotech/certs/device.key", // 设备私钥
                  NULL
    );
mosquitto_tls_opts_set(mosq, 1, "tlsv1.3", NULL);
```

3.5 断网离线缓存与自动续传模块

模块文件： `cache/offline_cache.c` 、 `cache/resume_uploader.c`

功能概述：

离线缓存模块在网络中断期间将笔迹数据持久化存储至SQLite数据库，确保即使在恶劣网络条件下，课堂上采集的所有书写数据都能最终完整地送达云平台。

断网检测机制：

网络状态检测周期：5秒

检测方法：

1. MQTT连接状态检测（libmosquitto on_disconnect回调触发）
2. 周期性Ping云平台健康检查接口
 - 发送HTTP HEAD请求至 `https://health.writech.com/ping`
 - 超时时间：3秒
 - 失败2次后判定为离线

网络恢复检测：

1. MQTT on_connect回调触发（库自动重连）
2. 恢复后触发离线数据补传流程

自动续传逻辑：

```
// cache/resume_uploader.c 续传逻辑（伪代码）
void resume_upload_task(void *arg) {
    OfflineCache *cache = (OfflineCache*)arg;

    while (true) {
        // 等待网络恢复信号
        sem_wait(&network_recovered_signal);

        // 查询待上传的离线数据（按seq_no顺序）
        int batch_size = 50; // 每批上传50条
        OfflineRecord records[batch_size];
        int count = offline_cache_query_pending(cache, records, batch_size);

        while (count > 0) {
            for (int i = 0; i < count; i++) {
                // 发布至MQTT（等待PUBACK确认）
                int rc = mqtt_publish_and_wait_ack(
                    records[i].data, records[i].data_size,
                    MQTT_TOPIC_STROKE, 3000); // 等待最多3秒

                if (rc == MQTT_SUCCESS) {
                    // 删除已成功上传的离线记录
                    offline_cache_mark_uploaded(cache, records[i].id);
                } else {
                    // 发送失败，退出续传循环，等待下次重试
                    break;
                }
            }
        }
        // 继续查询下一批
    }
}
```



```
        count = offline_cache_query_pending(cache, records, batch_size);
    }
}
}
```

3.6 设备发现与自动配对模块

模块文件： `device/device_manager.c`

功能概述：

设备发现模块负责持续扫描周围的BLE设备，识别并自动连接属于自然写点阵笔的设备（通过设备名称前缀识别），管理已配对设备的白名单。

自动配对策略：

扫描策略：

- 扫描间隔：每30秒执行一次主动扫描（Active Scanning）
- 扫描窗口：每次扫描持续5秒
- 过滤规则：广播包中包含"Writech-Pen-"前缀的设备名称

配对决策：

1. 若设备MAC地址在已配对白名单中 → 直接连接（无需重新配对）
2. 若设备MAC地址不在白名单，且当前连接数 < 40 → 发起配对请求
3. 若连接数已达40 → 忽略新发现的设备

配对过程：

1. 发送GATT Connect请求
2. 执行BLE LE Secure Connections配对（Numeric Comparison）
3. 配对成功后：
 - a. 将设备MAC地址写入SQLite白名单数据库
 - b. 订阅笔迹数据Characteristic的Notification
 - c. 读取设备信息（电量/固件版本/序列号）
 - d. 通过MQTT上报"新笔连接"事件至云平台

3.7 OTA固件远程升级模块

模块文件： `ota/ota_manager.c`

功能概述：

OTA升级模块实现了网关固件的远程无线升级功能，支持安全验证、断点续传和失败回滚，确保升级过程的可靠性和安全性。

OTA升级完整流程：

Step 1: 云平台触发升级指令

云端发布MQTT消息至 gateway/{id}/command

消息内容: {"action": "ota_upgrade", "version": "1.1.0", "url": "https://..."}

Step 2: 网关接收并解析升级指令

验证version > 当前版本 (防止降级攻击)

Step 3: 下载固件包 (HTTPS)

下载目标: Flash B分区 (/dev/mmcblk0p2)

下载方式: 分块下载 (每块1MB), 支持断点续传

通过MQTT上报下载进度 (百分比)

Step 4: 固件验证

Step 4a: SHA-256哈希校验 (对比云端提供的期望哈希值)

Step 4b: RSA-2048数字签名验证 (使用内置公钥验证固件签名)

验证失败: 丢弃固件, 上报错误, 继续运行当前版本

Step 5: 写入Flash并设置引导标志

将B分区标记为"待引导" (通过NVS存储引导标志)

Step 6: 系统重启

约30秒后, 软件调用 system("reboot") 重启

Step 7: Bootloader引导新版本

Bootloader读取引导标志, 选择B分区启动

新版本软件启动后, 向云平台发送"OTA成功"确认消息

将B分区标记为"已确认", 将原A分区标记为"回滚备份"

3.8 设备状态上报与心跳监测模块

模块文件: device/device_manager.c (心跳部分)

功能概述:

心跳上报模块定期将网关的运行状态发送至云平台, 使云平台能够实时了解每台网关的健康状态, 及时发现离线设备或异常情况。

心跳消息内容 (每60秒上报一次):

```
{
  "gateway_id": "GW_ROOM301_001",
  "timestamp": 1700000000000,
  "system": {
    "cpu_usage_percent": 15.2,
    "memory_used_mb": 128,
    "memory_total_mb": 512,
    "uptime_seconds": 86400,
    "cpu_temperature_celsius": 45.8,
    "disk_free_mb": 1024
  },
  "network": {
```

```

    "ssid": "School-AP-5G",
    "signal_strength_dbm": -55,
    "ip_address": "192.168.1.100",
    "mqtt_latency_ms": 35,
    "network_rx_bytes_total": 10485760,
    "network_tx_bytes_total": 5242880
  },
  "ble": {
    "connected_pens_count": 38,
    "connected_pens_list": ["AA:BB:CC:01", "AA:BB:CC:02"],
    "total_strokes_received_today": 125000
  },
  "software": {
    "firmware_version": "1.0.0",
    "offline_cache_size_kb": 0
  }
}

```

3.9 本地Web管理界面模块

模块文件： `config/config_web.py` （Python CGI脚本）

功能概述：

本地Web管理界面通过lighttpd Web服务器和Python CGI脚本提供，管理员可在同一局域网的浏览器中通过 `http://192.168.x.x:8080` 访问，无需连接互联网，便于现场排障和配置。

管理界面页面结构：

自然写网关管理 GW_R00M301_001 固件版本：V1.0.0	
导航菜单 <input checked="" type="checkbox"/> 系统概览 <input type="checkbox"/> 网络配置 <input type="checkbox"/> 已连接设备 <input type="checkbox"/> 离线缓存 <input type="checkbox"/> 日志查看	当前页面内容区域 <input checked="" type="checkbox"/> 系统状态卡片（CPU/内存/运行时长） <input checked="" type="checkbox"/> 蓝牙连接状态（已连接X支笔，列表可展开） <input checked="" type="checkbox"/> 网络状态（IP地址/信号强度/MQTT状态） <input checked="" type="checkbox"/> 最近告警日志（最近10条）

网络配置页面功能（修改WiFi设置）：

WiFi配置界面：
 SSID：[输入框_____]
 密码：[密码输入框_____]
 安全类型：[WPA2-PSK ▼]
 频段：[自动 ▼]
 [保存并重新连接]
 注意：重新连接期间（约10-30秒）网关暂时离线

第四章 操作流程与使用步骤

4.1 网关设备安装与初始化

物理安装：

- 步骤1：选择安装位置：教室正前方或后方的中央位置，确保蓝牙信号能覆盖教室全部学生
- 步骤2：通过网线将网关连接至校园网交换机（推荐有线连接，比WiFi更稳定）
- 步骤3：接通电源（12V DC电源适配器）
- 步骤4：等待约60秒，网关完成启动，状态LED蓝色常亮表示正常运行
- 步骤5：通过校园网络访问网关管理页面：<http://{网关IP}:8080>
(网关IP可在路由器/交换机的DHCP表中查询)

初次激活流程：

- 步骤1：打开管理页面，系统提示"设备未激活"
- 步骤2：输入设备序列号（贴纸在设备底部）确认设备所有权
- 步骤3：系统通过HTTPS连接云平台的设备认证服务，验证序列号合法性
- 步骤4：激活成功后，设备状态更新为"已激活"，LED由黄色变为蓝色
- 步骤5：在云平台的设备管理后台中，将网关绑定至对应的班级/教室

4.2 网络配置操作流程

WiFi配置操作（若使用无线连接）：

- 步骤1：访问管理页面，进入 网络配置 → WiFi设置
- 步骤2：点击"扫描"按钮，显示附近可用WiFi列表
- 步骤3：选择目标WiFi网络，输入密码
- 步骤4：点击"保存并连接"
- 步骤5：等待30秒，页面将显示连接结果（成功则显示IP地址）
- 步骤6：若失败，检查密码是否正确，或尝试重启网关

网络状态查看：

系统概览页面实时显示：

- 网络类型（有线/WiFi）及信号强度
- 当前IP地址和子网掩码
- MQTT连接状态（已连接/重连中/离线）
- 当日上传数据量

4.3 蓝牙笔连接与配对操作流程

首次配对点阵笔（以单支笔为例）：

- 步骤1：确保点阵笔已充满电（电量≥20%），并处于开机状态
- 步骤2：点阵笔开机后自动进入广播状态（蓝色LED快速闪烁）
- 步骤3：网关每30秒执行一次BLE扫描，发现新的点阵笔广播包
- 步骤4：网关自动发起配对请求（无需手动操作）
- 步骤5：点阵笔LED显示数字（6位数字码），网关管理页面同时显示相同数字
- 步骤6：管理员确认两边数字一致后，在管理页面点击"确认配对"
- 步骤7：配对完成，点阵笔LED变为蓝色常亮，表示已成功连接至网关
- 步骤8：管理页面显示已连接设备列表中新增该笔

已连接设备管理界面：

已连接设备（38/40）					
序号	MAC地址	设备名称	电量	操作	
1	AA:BB:CC:01:02:03	张三的笔	85%	断开/解绑	
2	AA:BB:CC:04:05:06	李四的笔	72%	断开/解绑	
38	AA:BB:CC:...:..	赵六的笔	15%△	断开/解绑	

（15%电量警告：建议提醒学生课后及时充电）

4.4 本地Web管理界面使用

访问方式：在同一局域网的浏览器中输入 `http://{网关IP}:8080`
默认端口：8080
认证：首次访问需设置管理员密码

主要操作：

- 1. 查看系统概览（实时状态监控）
- 2. 修改WiFi配置
- 3. 查看和管理已连接的点阵笔
- 4. 查看离线缓存状态（大小/条数/最早记录时间）
- 5. 查看最近系统日志（可筛选错误级别）
- 6. 手动触发系统重启
- 7. 导出诊断日志包（用于技术支持远程诊断）

4.5 OTA固件升级操作流程

云平台发起OTA升级（管理员操作）：

- 步骤1：登录云平台管理控制台
- 步骤2：进入 设备管理 → 网关管理 → 选择目标网关（可多选批量升级）
- 步骤3：点击"发起OTA升级"按钮
- 步骤4：选择目标固件版本（系统列出可用版本）
- 步骤5：选择升级时间窗口（建议选择非教学时段，如20:00-22:00）
- 步骤6：确认并提交升级任务

网关侧升级过程：

- 升级开始：LED黄色闪烁，管理页面显示"正在升级"

- 下载阶段：显示下载进度百分比
- 验证阶段：显示"验证固件中..."
- 重启阶段：显示"3秒后重启..."（倒计时），系统重启
- 升级完成：LED蓝色常亮，版本号更新至新版本
- 若升级失败：LED红色闪烁，版本号不变，告警推送至管理员

4.6 故障诊断与维护

LED状态指示表：

LED状态	含义	处理方法
蓝色常亮	正常运行，已连接至云平台	无需操作
蓝色慢闪（1秒/次）	正常运行，WiFi/有线连接正常，MQTT连接中	等待约30秒
黄色常亮	已联网但未激活	完成设备激活流程
黄色闪烁	OTA升级进行中	勿断电，等待完成
红色快闪（0.5秒/次）	严重错误（OTA失败/系统异常）	重启设备，联系技术支持
灭	断电或硬件故障	检查电源连接

常见故障排除：

故障	排查步骤
网关无法连接MQTT	检查网络是否通畅；检查TLS证书是否过期；查看MQTT日志
点阵笔无法配对	检查笔电量；重启笔后重新广播；检查连接数是否已达40上限
离线缓存一直增大	检查MQTT连接状态；网络恢复后缓存会自动清空
大屏收不到笔迹推送	检查大屏和网关是否在同一局域网；检查大屏APP的网关绑定配置

第五章 与源代码的对应关系

5.1 模块名称与源代码文件对应表

功能模块	源文件路径	语言	说明
应用程序主入口	main.c	C	进程初始化，各模块启动，主事件循环
BLE多笔连接管理	ble/ble_manager.c	C	BlueZ DBus接口，连接状态机，数据接收回调
环形缓冲区	cache/ring_buffer.c	C	线程安全环形缓冲区实现
笔迹数据接收	cache/stroke_receiver.c	C	BLE Notification解包，写入Ring Buffer
协议转换	protocol/protocol_converter.c	C	BLE格式→Protobuf格式转换
MQTT客户端	mqtt/mqtt_client.c	C	libmosquitto封装，TLS连接，QoS管理
离线缓存	cache/offline_cache.c	C	SQLite离线数据读写
续传上传	cache/resume_uploader.c	C	网络恢复后的缓存数据补传逻辑
设备管理	device/device_manager.c	C	设备列表维护，心跳上报，状态管理
OTA管理	ota/ota_manager.c	C	固件下载、验证、写入Flash
配置管理	config/config_manager.c	C	配置文件读写
本地Web后端	config/config_web.py	Python	lighttpd CGI，管理页面API

5.2 核心函数说明

main.c 核心流程：

```
int main(int argc, char *argv[]) {
    // 1. 解析命令行参数和配置文件
    GatewayConfig config;
    config_load("/etc/wrotech/config.json", &config);

    // 2. 初始化各模块
    ring_buffer_init(&g_ring_buffer);
    offline_cache_init(&g_offline_cache, "/data/stroke_cache.db");
    device_manager_init(&g_dev_mgr, "/data/devices.db");
    mqtt_client_init(&g_mqtt, &config.mqtt);
    ble_manager_init(&g_ble, ble_data_callback);

    // 3. 启动工作线程
```

```

pthread_create(&ble_thread, NULL, ble_scan_task, &g_ble);
pthread_create(&process_thread, NULL, data_process_task, NULL);
pthread_create(&mqtt_thread, NULL, mqtt_upload_task, &g_mqtt);
pthread_create(&resume_thread, NULL, resume_upload_task, &g_offline_cache);
pthread_create(&status_thread, NULL, status_report_task, &g_mqtt);

// 4. 主线程等待信号 (SIGTERM/SIGINT)
sigwait(&sig_set, &sig);

// 5. 优雅退出：等待各线程完成当前任务
cleanup_all_modules();
return 0;
}

```

ble/ble_manager.c 关键函数：

函数名	功能说明
ble_manager_init(mgr, callback)	初始化BlueZ DBus连接，注册设备发现回调
ble_scan_task(arg)	扫描线程：周期性执行BLE主动扫描
ble_connect_device(mac_addr)	向指定MAC地址的设备发起GATT连接
ble_subscribe_notifications(conn)	订阅已连接笔的笔迹数据Characteristic Notification
ble_data_callback(conn, data, len)	BLE数据接收回调：解包并写入环形缓冲区

5.3 命名规范

C语言命名规范：

- 函数名：小写字母+下划线，以模块名为前缀，如 ble_manager_init()、mqtt_client_connect()
- 宏定义：全大写+下划线，如 MAX_PEN_CONNECTIONS、RING_BUFFER_SIZE
- 结构体类型：首字母大写驼峰式，如 PenConnection、StrokeFrame、GatewayConfig
- 全局变量：以 g_ 前缀区分，如 g_ring_buffer、g_mqtt
- 常量：全大写，以模块名为前缀，如 BLE_SCAN_INTERVAL_SEC

文件命名规范：

- 源文件：功能名小写+下划线，如 ble_manager.c、ring_buffer.c
- 头文件：与源文件同名，后缀.h，如 ble_manager.h
- 目录：功能模块名小写，如 ble/、cache/、mqtt/、ota/、device/

附录

附录A 术语表

术语	说明
BLE	蓝牙低功耗 (Bluetooth Low Energy), 适合IoT设备的低功耗短距离无线通信技术
GATT	通用属性配置文件 (Generic Attribute Profile), BLE数据交换的标准框架
BlueZ	Linux官方蓝牙协议栈, 通过DBus接口供上层应用使用
MQTT	消息队列遥测传输 (Message Queuing Telemetry Transport), 轻量级发布/订阅消息协议
QoS	服务质量 (Quality of Service), MQTT中定义消息传输可靠性的参数 (0/1/2级别)
OTA	空中升级 (Over-The-Air), 通过无线网络远程更新设备软件
A/B分区	双启动分区设计, 当前运行分区和备用分区各一个, 升级时写备用分区, 保证可回滚
NVS	非易失性存储 (Non-Volatile Storage), 用于存储配置和状态数据, 断电不丢失
Protobuf	Protocol Buffers, Google设计的高效序列化格式, 比JSON体积更小解析更快
mTLS	双向TLS, 通信双方均需证书认证, 比单向TLS提供更强的安全保证

附录B 版本历史

版本号	发布日期	变更说明
V1.0	2026年2月	初始版本, 支持40笔BLE并发连接、离线缓存、OTA升级完整功能

编制单位: 深圳自然写科技有限公司

文档版本: V1.0

编制日期: 2026年2月

版权声明: 本文档版权归深圳自然写科技有限公司所有, 未经授权不得复制或传播

附录C 网关核心功能详细实现

C.1 BLE 多设备并发扫描与连接管理

网关需要同时管理 30~60 支点阵笔的 BLE 连接，对 BLE 子系统提出了严苛的并发要求。

C.1.1 BLE 扫描调度策略

```
/* ble_manager.c - BLE 连接管理核心 */
#include "ble_manager.h"
#include <pthread.h>
#include <string.h>

#define MAX_PENS 64          /* 最大支持点阵笔数量 */
#define SCAN_WINDOW_MS 50   /* 扫描窗口时长 (ms) */
#define SCAN_INTERVAL_MS 100 /* 扫描间隔 (ms) */
#define CONN_RETRY_MAX 5    /* 最大重连次数 */

/* 笔设备连接状态 */
typedef enum {
    PEN_STATE_UNKNOWN,
    PEN_STATE_DISCOVERED,
    PEN_STATE_CONNECTING,
    PEN_STATE_CONNECTED,
    PEN_STATE_DISCONNECTED,
    PEN_STATE_ERROR
} pen_state_t;

/* 笔设备描述符 */
typedef struct {
    char mac_addr[18];          /* MAC 地址 ("AA:BB:CC:DD:EE:FF") */
    char device_name[64];       /* 设备名称 */
    int conn_handle;            /* BLE 连接句柄 (-1=未连接) */
    pen_state_t state;          /* 当前状态 */
    int retry_count;            /* 当前重连次数 */
    int rssi;                   /* 信号强度 dBm */
    uint8_t battery_level;      /* 电量百分比 */
    uint16_t ink_char_handle;    /* 笔迹数据 Characteristic 句柄 */
    time_t last_seen;           /* 最后一次发现时间 (用于检测离线) */
    time_t connected_at;        /* 建立连接时间 */
    uint64_t received_bytes;     /* 累计接收字节数 (统计用) */
    pthread_mutex_t lock;       /* 每设备独立互斥锁 */
} pen_device_t;

/* 全局设备管理表 */
static pen_device_t g_pens[MAX_PENS];
static int g_pen_count = 0;
static pthread_rwlock_t g_pens_lock = PTHREAD_RWLOCK_INITIALIZER;

/**
 * BLE 扫描结果回调
 * 由 BlueZ D-Bus 信号触发 (在 BLE 扫描线程中执行)
 */
void on_ble_device_discovered(const char* mac, const char* name, int rssi) {
    /* 过滤: 只处理自然写点阵笔 (名称前缀匹配) */
    if (strncmp(name, "WritechPen-", 11) != 0) return;

    pthread_rwlock_wrlock(&g_pens_lock);
```

```

/* 查找是否已知设备 */
pen_device_t* pen = find_pen_by_mac(mac);

if (pen == NULL) {
    /* 新发现的设备 */
    if (g_pen_count < MAX_PENS) {
        pen = &g_pens[g_pen_count++];
        memset(pen, 0, sizeof(pen_device_t));
        strncpy(pen->mac_addr, mac, sizeof(pen->mac_addr) - 1);
        strncpy(pen->device_name, name, sizeof(pen->device_name) - 1);
        pen->conn_handle = -1;
        pen->state = PEN_STATE_DISCOVERED;
        pthread_mutex_init(&pen->lock, NULL);
        log_info("发现新点阵笔: %s (%s), RSSI=%d", name, mac, rssi);
    }
}

if (pen) {
    pen->rssi = rssi;
    pen->last_seen = time(NULL);

    /* 未连接的设备自动触发连接 */
    if (pen->state == PEN_STATE_DISCOVERED ||
        pen->state == PEN_STATE_DISCONNECTED) {
        pen->state = PEN_STATE_CONNECTING;
        schedule_connect(pen); /* 加入连接队列（异步执行） */
    }
}

pthread_rwlock_unlock(&g_pens_lock);
}

/**
 * BLE 连接建立回调
 */
void on_ble_connected(const char* mac, int conn_handle) {
    pthread_rwlock_rdlock(&g_pens_lock);
    pen_device_t* pen = find_pen_by_mac(mac);
    if (pen) {
        pthread_mutex_lock(&pen->lock);
        pen->conn_handle = conn_handle;
        pen->state = PEN_STATE_CONNECTED;
        pen->retry_count = 0;
        pen->connected_at = time(NULL);
        pthread_mutex_unlock(&pen->lock);
        log_info("点阵笔已连接: %s, handle=%d", mac, conn_handle);

        /* 连接成功后订阅笔迹 Notify Characteristic */
        subscribe_ink_characteristic(conn_handle, pen->ink_char_handle);
    }
    pthread_rwlock_unlock(&g_pens_lock);
}

/**
 * BLE 断线回调（处理断线重连逻辑）
 */
void on_ble_disconnected(const char* mac, int reason) {

```

```

pthread_rwlock_rdlock(&g_pens_lock);
pen_device_t* pen = find_pen_by_mac(mac);
if (pen) {
    pthread_mutex_lock(&pen->lock);
    pen->conn_handle = -1;
    pen->state = PEN_STATE_DISCONNECTED;

    if (pen->retry_count < CONN_RETRY_MAX) {
        /* 指数退避重连: 1s, 2s, 4s, 8s, 16s */
        int delay_sec = 1 << pen->retry_count;
        pen->retry_count++;
        log_warn("笔 %s 断线(reason=%d), %ds后重连(第%d次)",
            mac, reason, delay_sec, pen->retry_count);
        schedule_connect_delayed(pen, delay_sec);
    } else {
        pen->state = PEN_STATE_ERROR;
        log_error("笔 %s 重连失败超过 %d 次, 停止重连", mac, CONN_RETRY_MAX);
        notify_cloud_pen_offline(mac); /* 上报云平台设备离线 */
    }
    pthread_mutex_unlock(&pen->lock);
}
pthread_rwlock_unlock(&g_pens_lock);
}

```

C.1.2 笔迹数据接收与缓冲

```

/* ink_receiver.c - 笔迹数据接收处理 */

#define INK_BUFFER_SIZE (1024 * 64) /* 每支笔 64KB 环形缓冲 */
#define INK_POINT_SIZE 10 /* 单笔迹点 10 字节 */

/* 单支笔的笔迹接收缓冲区 */
typedef struct {
    uint8_t buf[INK_BUFFER_SIZE];
    int head; /* 写入位置 */
    int tail; /* 读取位置 */
    int count; /* 已缓冲字节数 */
    pthread_mutex_t lock;
} ink_ring_buffer_t;

/* 笔迹点结构 (与固件协议一致) */
typedef struct __attribute__((packed)) {
    uint16_t x; /* 横坐标 (点阵码坐标系) */
    uint16_t y; /* 纵坐标 */
    uint8_t pressure; /* 压感值 [0, 255] */
    uint32_t timestamp; /* 相对时间戳 (微秒) */
    uint8_t flags; /* 标志位 (bit0: penUp) */
} ink_point_raw_t;

/**
 * BLE Notify 数据到达回调 (在 BLE 接收线程中调用)
 */
void on_ink_data_received(int conn_handle,
    const uint8_t* data, uint16_t len) {

```

```

/* 通过连接句柄找到对应笔 */
pen_device_t* pen = find_pen_by_conn_handle(conn_handle);
if (!pen) return;

ink_ring_buffer_t* buf = get_ink_buffer(pen->mac_addr);

pthread_mutex_lock(&buf->lock);

/* 写入环形缓冲区 */
for (uint16_t i = 0; i < len; i++) {
    if (buf->count < INK_BUFFER_SIZE) {
        buf->buf[buf->head] = data[i];
        buf->head = (buf->head + 1) % INK_BUFFER_SIZE;
        buf->count++;
    } else {
        /* 缓冲区满: 丢弃最旧的数据 (保证实时性) */
        buf->tail = (buf->tail + INK_POINT_SIZE) % INK_BUFFER_SIZE;
        buf->count -= INK_POINT_SIZE;
        buf->buf[buf->head] = data[i];
        buf->head = (buf->head + 1) % INK_BUFFER_SIZE;
    }
}
pen->received_bytes += len;

pthread_mutex_unlock(&buf->lock);

/* 通知数据转发线程有新数据 */
sem_post(&g_ink_data_semaphore);
}

/**
 * 数据转发线程 (将缓冲数据转发给算力盒/云平台)
 */
void* ink_forward_thread(void* arg) {
    ink_point_raw_t points[MAX_BATCH_SIZE];
    int batch_count;

    while (g_running) {
        /* 等待笔迹数据信号量 */
        sem_wait(&g_ink_data_semaphore);

        /* 遍历所有在线笔, 批量读取数据 */
        pthread_rwlock_rdlock(&g_pens_lock);
        for (int i = 0; i < g_pen_count; i++) {
            pen_device_t* pen = &g_pens[i];
            if (pen->state != PEN_STATE_CONNECTED) continue;

            ink_ring_buffer_t* buf = get_ink_buffer(pen->mac_addr);
            batch_count = drain_ink_buffer(buf, points, MAX_BATCH_SIZE);

            if (batch_count > 0) {
                /* 打包成 MQTT 消息发送 */
                mqtt_publish_ink_batch(pen->mac_addr, points, batch_count);
            }
        }
        pthread_rwlock_unlock(&g_pens_lock);
    }
}

```

```
return NULL;
}
```

C.2 MQTT 消息协议详细说明

C.2.1 Topic 结构设计

自然写网关 MQTT Topic 命名规范:

上行 (网关 → 云平台):

```
writetech/{school_id}/{classroom_id}/gw/{gateway_id}/pen/{pen_serial}/ink
writetech/{school_id}/{classroom_id}/gw/{gateway_id}/pen/{pen_serial}/event
writetech/{school_id}/{classroom_id}/gw/{gateway_id}/status
```

下行 (云平台 → 网关):

```
writetech/{school_id}/{classroom_id}/gw/{gateway_id}/cmd
writetech/{school_id}/{classroom_id}/gw/{gateway_id}/config
```

C.2.2 笔迹消息格式 (二进制编码)

```
/* mqtt_protocol.c - MQTT 消息编码 */

#define WRITETECH_MAGIC    0xABCD /* 消息魔数 */
#define MSG_TYPE_INK      0x01 /* 笔迹数据消息 */
#define MSG_TYPE_EVENT    0x02 /* 设备事件消息 */
#define MSG_TYPE_STATUS   0x03 /* 设备状态消息 */

/**
 * 笔迹数据消息结构 (二进制)
 *
 * 字段          偏移  长度  说明
 * magic          0      2    固定值 0xABCD
 * version        2      1    协议版本 (当前 0x01)
 * msg_type       3      1    消息类型 (0x01=笔迹)
 * session_id     4      8    课堂会话ID (uint64)
 * pen_serial     12     16    笔序列号 (ASCII, 不足16字节补0)
 * timestamp      28     8    消息时间戳 (Unix时间戳, 微秒, uint64)
 * point_count    36     2    笔迹点数量 (uint16)
 * checksum       38     4    CRC32校验 (覆盖 magic~point_count)
 * points         42    N×10  笔迹点数组 (每点10字节)
 */

/* 编码笔迹消息 */
int encode_ink_message(uint8_t* buf, int buf_size,
                      uint64_t session_id,
                      const char* pen_serial,
                      const ink_point_raw_t* points,
                      uint16_t point_count) {
    int total_size = 42 + point_count * INK_POINT_SIZE;
    if (buf_size < total_size) return -1;
```

```

/* 写入头部 */
*(uint16_t*)(buf + 0) = htons(WRITECH_MAGIC);
buf[2] = 0x01; /* version */
buf[3] = MSG_TYPE_INK;
*(uint64_t*)(buf + 4) = htobe64(session_id);
memset(buf + 12, 0, 16);
strncpy((char*)(buf + 12), pen_serial, 16);
*(uint64_t*)(buf + 28) = htobe64(get_current_timestamp_us());
*(uint16_t*)(buf + 36) = htons(point_count);

/* 计算 CRC32 并写入 */
uint32_t crc = crc32_calc(buf, 38);
*(uint32_t*)(buf + 38) = htonl(crc);

/* 写入笔迹点数组（紧凑二进制） */
memcpy(buf + 42, points, point_count * INK_POINT_SIZE);

return total_size;
}

```

C.2.3 网关状态上报

```

/* gateway_status.c */

/* 定时上报网关状态（每30秒一次） */
void report_gateway_status(void) {
    char json_buf[4096];
    int connected_pens = count_connected_pens();
    int total_pens = g_pen_count;

    snprintf(json_buf, sizeof(json_buf),
        "{\n"
        "  \"gateway_id\": \"%s\",\n"
        "  \"timestamp\": %ld,\n"
        "  \"uptime_sec\": %ld,\n"
        "  \"connected_pens\": %d,\n"
        "  \"total_pens\": %d,\n"
        "  \"cpu_usage_pct\": %.1f,\n"
        "  \"mem_free_mb\": %d,\n"
        "  \"storage_free_mb\": %d,\n"
        "  \"network_quality\": \"%s\",\n"
        "  \"mqtt_reconnect_count\": %d,\n"
        "  \"ink_bytes_total\": %lu,\n"
        "  \"firmware_version\": \"%s\"\n"
        "}",
        g_gateway_id,
        time(NULL),
        time(NULL) - g_start_time,
        connected_pens,
        total_pens,
        get_cpu_usage(),
        get_free_memory_mb(),
        get_free_storage_mb(),

```

```

        get_network_quality_str(),
        g_mqtt_reconnect_count,
        g_total_ink_bytes,
        FIRMWARE_VERSION
    );

    mqtt_publish(g_status_topic, json_buf, strlen(json_buf), 0, true);
}

```

C.3 本地缓存与数据恢复

C.3.1 SQLite 本地缓存实现

网关在与云平台断连时使用 SQLite 暂存笔迹数据，网络恢复后自动上传：

```

/* data_cache.c - 离线缓存管理 */
#include <sqlite3.h>

static sqlite3* g_cache_db = NULL;

/* 初始化本地缓存数据库 */
int cache_init(const char* db_path) {
    int rc = sqlite3_open(db_path, &g_cache_db);
    if (rc != SQLITE_OK) {
        log_error("无法打开缓存数据库: %s", sqlite3_errmsg(g_cache_db));
        return -1;
    }
}

/* 创建笔迹缓存表 */
const char* create_sql =
    "CREATE TABLE IF NOT EXISTS ink_cache ("
    "  id          INTEGER PRIMARY KEY AUTOINCREMENT,"
    "  session_id TEXT NOT NULL,"
    "  pen_serial TEXT NOT NULL,"
    "  timestamp  INTEGER NOT NULL," /* Unix时间戳 (秒) */
    "  data       BLOB NOT NULL,"    /* 压缩后的笔迹二进制数据 */
    "  uploaded   INTEGER DEFAULT 0," /* 0=未上传 1=已上传 */
    "  created_at INTEGER DEFAULT (strftime('%s','now'))"
    ");"
    "CREATE INDEX IF NOT EXISTS idx_ink_upload ON ink_cache(uploaded, created_at);"
    "CREATE TABLE IF NOT EXISTS event_cache ("
    "  id          INTEGER PRIMARY KEY AUTOINCREMENT,"
    "  event_type  TEXT NOT NULL,"
    "  event_data  TEXT NOT NULL,"    /* JSON格式事件数据 */
    "  uploaded   INTEGER DEFAULT 0,"
    "  created_at INTEGER DEFAULT (strftime('%s','now'))"
    ");";

char* err_msg = NULL;
rc = sqlite3_exec(g_cache_db, create_sql, NULL, NULL, &err_msg);
if (rc != SQLITE_OK) {
    log_error("创建缓存表失败: %s", err_msg);
}

```



```

        sqlite3_free(err_msg);
        return -1;
    }

    /* 启用 WAL 模式 (写入性能更好, 读写并发更好) */
    sqlite3_exec(g_cache_db, "PRAGMA journal_mode=WAL;", NULL, NULL, NULL);
    sqlite3_exec(g_cache_db, "PRAGMA synchronous=NORMAL;", NULL, NULL, NULL);

    log_info("本地缓存数据库初始化完成: %s", db_path);
    return 0;
}

/* 写入笔迹缓存 */
int cache_write_ink(const char* session_id, const char* pen_serial,
                   const uint8_t* data, int data_len) {
    const char* insert_sql =
        "INSERT INTO ink_cache (session_id, pen_serial, timestamp, data) "
        "VALUES (?, ?, strftime('%s','now'), ?);";

    sqlite3_stmt* stmt;
    sqlite3_prepare_v2(g_cache_db, insert_sql, -1, &stmt, NULL);
    sqlite3_bind_text(stmt, 1, session_id, -1, SQLITE_STATIC);
    sqlite3_bind_text(stmt, 2, pen_serial, -1, SQLITE_STATIC);
    sqlite3_bind_blob(stmt, 3, data, data_len, SQLITE_STATIC);

    int rc = sqlite3_step(stmt);
    sqlite3_finalize(stmt);

    return (rc == SQLITE_DONE) ? 0 : -1;
}

/* 查询未上传的缓存数据 (网络恢复后批量上传) */
int cache_get_unuploaded(ink_cache_item_t* items, int max_count) {
    const char* select_sql =
        "SELECT id, session_id, pen_serial, timestamp, data "
        "FROM ink_cache WHERE uploaded = 0 "
        "ORDER BY created_at ASC LIMIT ?;";

    sqlite3_stmt* stmt;
    sqlite3_prepare_v2(g_cache_db, select_sql, -1, &stmt, NULL);
    sqlite3_bind_int(stmt, 1, max_count);

    int count = 0;
    while (sqlite3_step(stmt) == SQLITE_ROW && count < max_count) {
        items[count].id = sqlite3_column_int64(stmt, 0);
        strncpy(items[count].session_id,
                (char*)sqlite3_column_text(stmt, 1), 64);
        strncpy(items[count].pen_serial,
                (char*)sqlite3_column_text(stmt, 2), 32);
        items[count].timestamp = sqlite3_column_int64(stmt, 3);
        items[count].data_len = sqlite3_column_bytes(stmt, 4);
        items[count].data = malloc(items[count].data_len);
        memcpy(items[count].data,
                sqlite3_column_blob(stmt, 4), items[count].data_len);
        count++;
    }
    sqlite3_finalize(stmt);
}

```



```

        log_error("固件签名验证失败, 拒绝升级");
        unlink(temp_path);
        return OTA_ERR_SIGNATURE;
    }

    /* 步骤3: SHA256 完整性校验 */
    calc_sha256(temp_path, sha256_actual);
    if (strcmp(sha256_actual, expected_sha256) != 0) {
        log_error("固件 SHA256 校验失败: expected=%s actual=%s",
                expected_sha256, sha256_actual);
        unlink(temp_path);
        return OTA_ERR_CHECKSUM;
    }

    state = OTA_STATE_APPLYING;

    /* 步骤4: 写入备用分区 */
    const char* inactive_partition = get_inactive_partition();
    log_info("写入分区: %s", inactive_partition);
    if (write_firmware_to_partition(temp_path, inactive_partition) != 0) {
        log_error("写入分区失败");
        return OTA_ERR_WRITE;
    }

    /* 步骤5: 设置下次启动使用新分区 */
    set_boot_partition(inactive_partition);

    state = OTA_STATE_REBOOTING;
    log_info("OTA 升级完成, 准备重启...");

    /* 延迟重启 (留时间让当前课堂完成, 最多等待5分钟) */
    schedule_reboot(300);

    return OTA_SUCCESS;
}

```

附录D 网关部署与配置

D.1 出厂配置文件

```

# /etc/wrotech/gateway.conf
[network]
interface=eth0
wifi_ssid=
wifi_password=
mqtt_broker=mqtt.wrotech.com
mqtt_port=8883
mqtt_tls=true
mqtt_heartbeat=60

[ble]

```

```
scan_enabled=true
scan_interval_ms=100
scan_window_ms=50
max_connections=64
auto_reconnect=true
reconnect_max_retries=5

[cloud]
api_endpoint=https://api.writech.com
school_id=
classroom_id=
gateway_id=
auth_token=

[cache]
db_path=/var/lib/writech/ink_cache.db
max_size_mb=512
upload_batch_size=100
upload_retry_interval_sec=30

[ota]
check_interval_hours=24
auto_apply=false
public_key_path=/etc/writech/ota_public_key.pem

[log]
level=INFO
path=/var/log/writech/gateway.log
max_size_mb=50
max_files=5
```

D.2 系统服务配置

```
# /etc/systemd/system/writech-gateway.service
[Unit]
Description=Writech Classroom Gateway Service
After=network-online.target bluetooth.service
Wants=network-online.target

[Service]
Type=simple
User=writech
Group=writech
ExecStart=/usr/bin/writech-gateway -c /etc/writech/gateway.conf
ExecStop=/bin/kill -TERM $MAINPID
Restart=always
RestartSec=5
LimitNOFILE=65536
LimitNPROC=4096

# 安全加固
NoNewPrivileges=yes
PrivateTmp=yes
ProtectSystem=strict
```

```
ReadWritePaths=/var/lib/writtech /var/log/writtech /tmp

[Install]
WantedBy=multi-user.target
```

D.3 性能调优参数

参数	默认值	说明
BLE 扫描间隔	100ms	降低可提高发现速度，但增加功耗和干扰
BLE 连接超时	10s	超时后触发重连
MQTT 保活周期	60s	检测网络断连的心跳间隔
笔迹缓冲区大小	64KB/笔	每支笔独立缓冲区，防止相互影响
批量转发大小	34点/批	每次 MQTT 消息包含的笔迹点数
SQLite 页面大小	4096B	与操作系统页面大小一致，减少碎片
日志级别	INFO	生产环境建议 INFO，调试时改为 DEBUG

本文档版权归深圳自然写科技有限公司所有，仅用于软件著作权登记鉴别。

附录E 核心模块代码补充

E.1 MQTT消息处理完整实现

网关通过MQTT协议将笔迹数据、设备状态上报到云端，同时接收云端下发的控制指令。

```
/* mqtt_client.c - MQTT消息处理完整实现 */

#include "mqtt_client.h"
#include "ble_manager.h"
#include "data_cache.h"
#include <mosquitto.h>
#include <pthread.h>
#include <string.h>
#include <stdlib.h>

#define MQTT_BROKER_HOST    "mqtt.writtech.com"
#define MQTT_BROKER_PORT    8883          /* TLS端口 */
#define MQTT_KEEPALIVE      60            /* 秒 */
#define MQTT_QOS_AT_LEAST_ONCE 1
#define MQTT_QOS_AT_MOST_ONCE 0
```

```

/* 主题模板 */
#define TOPIC_INK_DATA      "gateway/%s/ink/data"
#define TOPIC_STATUS        "gateway/%s/status"
#define TOPIC_CONTROL_SUB   "gateway/%s/control/#"
#define TOPIC_ALERT         "gateway/%s/alert"

static struct mosquitto *g_mosq = NULL;
static char g_device_id[64] = {0};
static pthread_mutex_t g_publish_mutex = PTHREAD_MUTEX_INITIALIZER;
static volatile bool g_connected = false;

/**
 * @brief 初始化MQTT客户端
 */
int mqtt_client_init(const char *device_id, const char *token,
                    const char *ca_cert_path) {
    strncpy(g_device_id, device_id, sizeof(g_device_id) - 1);

    mosquitto_lib_init();
    g_mosq = mosquitto_new(device_id, true, NULL);
    if (!g_mosq) {
        LOG_ERROR("mosquitto_new failed");
        return -1;
    }

    /* 配置TLS */
    mosquitto_tls_set(g_mosq, ca_cert_path, NULL, NULL, NULL, NULL);
    mosquitto_tls_opts_set(g_mosq, 1, "tlsv1.2", NULL);

    /* 配置用户名/密码认证 */
    mosquitto_username_pw_set(g_mosq, device_id, token);

    /* 注册回调 */
    mosquitto_connect_callback_set(g_mosq, on_connect);
    mosquitto_disconnect_callback_set(g_mosq, on_disconnect);
    mosquitto_message_callback_set(g_mosq, on_message);
    mosquitto_publish_callback_set(g_mosq, on_publish);

    /* 异步连接 */
    int rc = mosquitto_connect_async(g_mosq, MQTT_BROKER_HOST, MQTT_BROKER_PORT,
                                    MQTT_KEEPALIVE);

    if (rc != MOSQ_ERR_SUCCESS) {
        LOG_ERROR("mqtt connect failed: %s", mosquitto_strerror(rc));
        return -1;
    }

    /* 启动MQTT网络循环线程 */
    mosquitto_loop_start(g_mosq);
    return 0;
}

/**
 * @brief 连接成功回调: 订阅控制主题
 */
static void on_connect(struct mosquitto *mosq, void *obj, int rc) {
    if (rc == 0) {

```

```

    g_connected = true;
    LOG_INFO("MQTT connected to %s:%d", MQTT_BROKER_HOST, MQTT_BROKER_PORT);

    char control_topic[128];
    snprintf(control_topic, sizeof(control_topic), TOPIC_CONTROL_SUB, g_device_id);
    mosquitto_subscribe(mosq, NULL, control_topic, MQTT_QOS_AT_LEAST_ONCE);

    /* 上报上线状态 */
    mqtt_publish_status("online");

    /* 发送离线缓存数据 (如果有) */
    data_cache_flush_to_mqtt();
} else {
    LOG_ERROR("MQTT connect rejected: %d", rc);
}
}

/**
 * @brief 断线回调: 标记状态, mosquitto自动重连
 */
static void on_disconnect(struct mosquitto *mosq, void *obj, int rc) {
    g_connected = false;
    LOG_WARN("MQTT disconnected: rc=%d, will reconnect...", rc);
}

/**
 * @brief 接收控制消息回调
 * 支持的控制指令:
 * - classroom_start: {"action":"classroom_start","classroom_id":"..."}
 * - classroom_end: {"action":"classroom_end"}
 * - ota_start: {"action":"ota_start","url":"...", "md5":"...", "version":"..."}
 * - config_update: {"action":"config_update","config":{"..."}}
 * - reboot: {"action":"reboot"}
 */
static void on_message(struct mosquitto *mosq, void *obj,
                      const struct mosquitto_message *msg) {
    if (!msg->payload || msg->payloadlen == 0) return;

    char *payload = strdup((char*)msg->payload, msg->payloadlen);
    LOG_DEBUG("MQTT message: topic=%s, payload=%s", msg->topic, payload);

    /* 解析JSON指令 */
    cJSON *json = cJSON_Parse(payload);
    if (!json) {
        LOG_ERROR("JSON parse failed: %s", payload);
        free(payload);
        return;
    }

    cJSON *action_json = cJSON_GetObjectItem(json, "action");
    if (!action_json || !cJSON_IsString(action_json)) {
        goto cleanup;
    }
    const char *action = action_json->valuestring;

    if (strcmp(action, "classroom_start") == 0) {
        cJSON *cid = cJSON_GetObjectItem(json, "classroom_id");

```

```

        if (cid && cJSON_IsString(cid)) {
            classroom_manager_start(cid->valuelstring);
        }
    } else if (strcmp(action, "classroom_end") == 0) {
        classroom_manager_end();
    } else if (strcmp(action, "ota_start") == 0) {
        cJSON *url = cJSON_GetObjectItem(json, "url");
        cJSON *md5 = cJSON_GetObjectItem(json, "md5");
        cJSON *ver = cJSON_GetObjectItem(json, "version");
        if (url && md5 && ver) {
            ota_manager_start(url->valuelstring, md5->valuelstring, ver->valuelstring);
        }
    } else if (strcmp(action, "reboot") == 0) {
        LOG_INFO("Reboot command received");
        sleep(3);
        system("reboot");
    }
}

cleanup:
    cJSON_Delete(json);
    free(payload);
}

/**
 * @brief 发布笔迹数据到MQTT (优先网络发送, 失败则缓存本地)
 */
int mqtt_publish_ink_data(const ink_batch_t *batch) {
    if (!g_connected) {
        /* 无网络时缓存到SQLite */
        return data_cache_store_ink_batch(batch);
    }

    /* 序列化为二进制格式 */
    uint8_t buf[4096];
    int len = ink_batch_serialize(batch, buf, sizeof(buf));
    if (len <= 0) return -1;

    char topic[128];
    snprintf(topic, sizeof(topic), TOPIC_INK_DATA, g_device_id);

    pthread_mutex_lock(&g_publish_mutex);
    int rc = mosquitto_publish(g_mosq, NULL, topic, len, buf,
                               MQTT_QOS_AT_LEAST_ONCE, false);
    pthread_mutex_unlock(&g_publish_mutex);

    if (rc != MOSQ_ERR_SUCCESS) {
        LOG_WARN("MQTT publish failed: %s, caching locally", mosquitto_strerror(rc));
        return data_cache_store_ink_batch(batch);
    }
    return 0;
}

/**
 * @brief 发布设备状态JSON
 */
int mqtt_publish_status(const char *status) {
    char topic[128], payload[512];

```



```

    snprintf(topic, sizeof(topic), TOPIC_STATUS, g_device_id);
    snprintf(payload, sizeof(payload),
        "{\"device_id\":\"%s\", \"status\":\"%s\", \"pen_count\":%d, \"uptime\":\"%lu\", \"timestamp\":\"%ld\"}",
        g_device_id, status,
        ble_manager_get_connected_count(),
        get_uptime_seconds(),
        (long)time(NULL));

    return mosquitto_publish(g_mosq, NULL, topic, strlen(payload), payload,
        MQTT_QOS_AT_MOST_ONCE, false);
}

```

E.2 OTA升级完整实现（A/B分区）

```

/* ota/ota_manager.c - OTA升级管理（A/B分区）*/

#include "ota_manager.h"
#include "mqtt_client.h"
#include <openssl/rsa.h>
#include <openssl/sha.h>
#include <curl/curl.h>

#define OTA_PARTITION_A    "/dev/mmcblk0p3"
#define OTA_PARTITION_B    "/dev/mmcblk0p4"
#define OTA_FLAG_FILE      "/data/ota/active_partition"
#define OTA_DOWNLOAD_TMP   "/data/ota/firmware_new.bin"
#define RSA_PUBLIC_KEY_PATH "/etc/wrotech/ota_public.pem"
#define CHUNK_SIZE         (64 * 1024) /* 64KB下载块 */

typedef struct {
    FILE *fp;
    size_t total;
    size_t downloaded;
    char md5_expected[33];
} DownloadCtx;

static ota_state_t g_ota_state = OTA_IDLE;

/**
 * @brief 启动OTA升级（在独立线程中执行，不阻塞主业务）
 */
int ota_manager_start(const char *url, const char *md5, const char *version) {
    if (g_ota_state != OTA_IDLE) {
        LOG_WARN("OTA already in progress");
        return -1;
    }

    ota_params_t *params = malloc(sizeof(ota_params_t));
    strncpy(params->url, url, sizeof(params->url) - 1);
    strncpy(params->md5, md5, sizeof(params->md5) - 1);
    strncpy(params->version, version, sizeof(params->version) - 1);

    pthread_t ota_thread;

```

```

    pthread_create(&ota_thread, NULL, ota_worker_thread, params);
    pthread_detach(ota_thread);
    return 0;
}

static void *ota_worker_thread(void *arg) {
    ota_params_t *params = (ota_params_t*)arg;
    g_ota_state = OTA_DOWNLOADING;
    mqtt_publish_ota_progress(0, "downloading");

    /* Step 1: 下载固件到临时文件 */
    if (ota_download(params->url, OTA_DOWNLOAD_TMP, params->md5) != 0) {
        LOG_ERROR("OTA download failed");
        g_ota_state = OTA_IDLE;
        mqtt_publish_ota_progress(-1, "download_failed");
        goto cleanup;
    }
    mqtt_publish_ota_progress(60, "verifying");

    /* Step 2: RSA签名验证（防刷机攻击）*/
    if (ota_verify_signature(OTA_DOWNLOAD_TMP, RSA_PUBLIC_KEY_PATH) != 0) {
        LOG_ERROR("OTA signature verification failed");
        g_ota_state = OTA_IDLE;
        mqtt_publish_ota_progress(-1, "verify_failed");
        goto cleanup;
    }
    mqtt_publish_ota_progress(70, "flashing");
    g_ota_state = OTA_FLASHING;

    /* Step 3: 确定目标分区（写入当前不使用的分区）*/
    const char *target_partition = ota_get_inactive_partition();
    if (ota_flash_partition(OTA_DOWNLOAD_TMP, target_partition) != 0) {
        LOG_ERROR("OTA flash failed");
        g_ota_state = OTA_IDLE;
        mqtt_publish_ota_progress(-1, "flash_failed");
        goto cleanup;
    }

    /* Step 4: 更新启动标志，指向新分区 */
    ota_set_active_partition(target_partition);
    mqtt_publish_ota_progress(100, "success");
    LOG_INFO("OTA success, new version: %s, will reboot in 10s", params->version);

    /* Step 5: 延迟重启（等待MQTT消息发送完成）*/
    sleep(10);
    system("reboot");

cleanup:
    free(params);
    return NULL;
}

/**
 * @brief 获取当前不活跃的分区（用于写入新固件）
 */
static const char *ota_get_inactive_partition(void) {
    FILE *f = fopen(OTA_FLAG_FILE, "r");

```

```

    if (!f) return OTA_PARTITION_B; /* 默认从B分区开始 */
    char active[32] = {0};
    fscanf(f, "%31s", active);
    fclose(f);
    return (strcmp(active, OTA_PARTITION_A) == 0) ? OTA_PARTITION_B : OTA_PARTITION_A;
}

static void ota_set_active_partition(const char *partition) {
    FILE *f = fopen(OTA_FLAG_FILE, "w");
    if (f) {
        fprintf(f, "%s", partition);
        fclose(f);
        sync(); /* 确保写入持久化 */
    }
}
}

```

E.3 SQLite WAL模式离线缓存

```

/* data_cache.c - SQLite WAL模式离线缓存 */

#include "data_cache.h"
#include <sqlite3.h>

#define DB_PATH "/data/writetech/gateway_cache.db"
#define CACHE_MAX_ROWS 50000 /* 最大缓存5万条记录 (约500MB) */

static sqlite3 *g_db = NULL;
static pthread_mutex_t g_db_mutex = PTHREAD_MUTEX_INITIALIZER;

int data_cache_init(void) {
    int rc = sqlite3_open(DB_PATH, &g_db);
    if (rc != SQLITE_OK) {
        LOG_ERROR("Open DB failed: %s", sqlite3_errmsg(g_db));
        return -1;
    }
}

/* 启用WAL模式 (Write-Ahead Log): 提升并发读写性能 */
sqlite3_exec(g_db, "PRAGMA journal_mode=WAL;", NULL, NULL, NULL);
sqlite3_exec(g_db, "PRAGMA synchronous=NORMAL;", NULL, NULL, NULL);
sqlite3_exec(g_db, "PRAGMA cache_size=-8000;", NULL, NULL, NULL); /* 8MB页缓存 */
sqlite3_exec(g_db, "PRAGMA temp_store=MEMORY;", NULL, NULL, NULL);

/* 建表 */
sqlite3_exec(g_db,
    "CREATE TABLE IF NOT EXISTS ink_cache ("
    "  id          INTEGER PRIMARY KEY AUTOINCREMENT,"
    "  pen_id      TEXT NOT NULL,"
    "  data        BLOB NOT NULL,"
    "  ts          INTEGER NOT NULL,"
    "  sent        INTEGER DEFAULT 0,"
    "  created     INTEGER DEFAULT (strftime('%s','now'))"
    ");",
    NULL, NULL, NULL);

```

```

/* 为未发送记录建索引（加速查询待同步数据）*/
sqlite3_exec(g_db,
    "CREATE INDEX IF NOT EXISTS idx_ink_unsent ON ink_cache(sent, created) WHERE
sent=0;",
    NULL, NULL, NULL);

return 0;
}

/**
 * @brief 存储笔迹批次到本地缓存
 */
int data_cache_store_ink_batch(const ink_batch_t *batch) {
    pthread_mutex_lock(&g_db_mutex);

    /* 检查是否超过上限，超过则删除最旧的已发送记录 */
    int64_t count = 0;
    sqlite3_exec(g_db, "SELECT COUNT(*) FROM ink_cache WHERE sent=0;",
        count_callback, &count, NULL);
    if (count >= CACHE_MAX_ROWS) {
        sqlite3_exec(g_db,
            "DELETE FROM ink_cache WHERE sent=1 ORDER BY created ASC LIMIT 1000;",
            NULL, NULL, NULL);
    }

    sqlite3_stmt *stmt;
    sqlite3_prepare_v2(g_db,
        "INSERT INTO ink_cache (pen_id, data, ts) VALUES (?, ?, ?);",
        -1, &stmt, NULL);
    sqlite3_bind_text(stmt, 1, batch->pen_id, -1, SQLITE_STATIC);
    sqlite3_bind_blob(stmt, 2, batch->data, batch->data_len, SQLITE_STATIC);
    sqlite3_bind_int64(stmt, 3, (int64_t)batch->timestamp);
    int rc = sqlite3_step(stmt);
    sqlite3_finalize(stmt);

    pthread_mutex_unlock(&g_db_mutex);
    return (rc == SQLITE_DONE) ? 0 : -1;
}

/**
 * @brief 网络恢复后批量上传缓存数据
 */
void data_cache_flush_to_mqtt(void) {
    pthread_mutex_lock(&g_db_mutex);

    sqlite3_stmt *stmt;
    sqlite3_prepare_v2(g_db,
        "SELECT id, pen_id, data, ts FROM ink_cache WHERE sent=0 ORDER BY created ASC
LIMIT 100;",
        -1, &stmt, NULL);

    int flushed = 0;
    while (sqlite3_step(stmt) == SQLITE_ROW) {
        int64_t id = sqlite3_column_int64(stmt, 0);
        const char *pen_id = (const char*)sqlite3_column_text(stmt, 1);
        const void *data = sqlite3_column_blob(stmt, 2);
        int data_len = sqlite3_column_bytes(stmt, 2);
    }
}

```

```

        int64_t ts = sqlite3_column_int64(stmt, 3);

        ink_batch_t batch = {
            .pen_id = pen_id,
            .data = (uint8_t*)data,
            .data_len = data_len,
            .timestamp = (time_t)ts
        };

        /* 直接通过MQTT发送（绕过缓存逻辑） */
        if (mqtt_publish_ink_data_direct(&batch) == 0) {
            /* 标记为已发送 */
            char sql[64];
            snprintf(sql, sizeof(sql), "UPDATE ink_cache SET sent=1 WHERE id=%lld;",
id);

            sqlite3_exec(g_db, sql, NULL, NULL, NULL);
            flushed++;
        } else {
            break; /* 发送失败，停止继续尝试 */
        }
    }
    sqlite3_finalize(stmt);
    pthread_mutex_unlock(&g_db_mutex);

    if (flushed > 0) {
        LOG_INFO("Flushed %d cached ink batches to MQTT", flushed);
    }
}

```

附录E 补充技术规格

E.1 BLE设备过滤与扫描优化

```

// ble_filter.c
#define WRITECH_PEN_NAME_PREFIX    "WritechPen-"
#define WRITECH_PEN_NAME_PREFIX_LEN 11
#define MIN_RSSI_THRESHOLD        -85    // dBm

typedef struct {
    char name[32];
    char mac[18];
    int rssi;
    uint32_t last_seen_ms;
    bool is_paired;
} scan_result_t;

static scan_result_t g_scan_results[64];
static int g_scan_count = 0;
static pthread_mutex_t g_scan_lock = PTHREAD_MUTEX_INITIALIZER;

void on_ble_scan_result(const char* name, const char* mac, int rssi) {

```

```

    if (rssi < MIN_RSSI_THRESHOLD) return;
    if (strncmp(name, WRTTECH_PEN_NAME_PREFIX, WRTTECH_PEN_NAME_PREFIX_LEN) != 0)
return;

    pthread_mutex_lock(&g_scan_lock);

    for (int i = 0; i < g_scan_count; i++) {
        if (strcmp(g_scan_results[i].mac, mac) == 0) {
            g_scan_results[i].rssi = rssi;
            g_scan_results[i].last_seen_ms = get_uptime_ms();
            pthread_mutex_unlock(&g_scan_lock);
            return;
        }
    }

    if (g_scan_count < 64) {
        strncpy(g_scan_results[g_scan_count].name, name, 31);
        strncpy(g_scan_results[g_scan_count].mac, mac, 17);
        g_scan_results[g_scan_count].rssi = rssi;
        g_scan_results[g_scan_count].last_seen_ms = get_uptime_ms();
        g_scan_results[g_scan_count].is_paired = is_mac_paired(mac);
        g_scan_count++;
    }

    pthread_mutex_unlock(&g_scan_lock);
}

void purge_stale_scan_results(void) {
    uint32_t now = get_uptime_ms();
    pthread_mutex_lock(&g_scan_lock);
    int write = 0;
    for (int i = 0; i < g_scan_count; i++) {
        if (now - g_scan_results[i].last_seen_ms < 30000) {
            g_scan_results[write++] = g_scan_results[i];
        }
    }
    g_scan_count = write;
    pthread_mutex_unlock(&g_scan_lock);
}

```

E.2 MQTT帧格式定义

```

// mqtt_protocol.h
#pragma pack(push, 1)

typedef struct {
    uint8_t  magic;      // 0xAB
    uint8_t  version;    // 0x01
    uint8_t  type;       // 帧类型
    uint8_t  flags;      // bit0=压缩, bit1=加密
} frame_header_t;

typedef struct {
    uint32_t pen_id;

```

```

    uint32_t sequence;
    uint32_t timestamp_ms;
    uint16_t point_count;
} ink_data_frame_t;

typedef struct {
    uint16_t x;
    uint16_t y;
    uint8_t  pressure;
    uint8_t  tilt_x;
    uint8_t  tilt_y;
    uint8_t  pen_up;
    uint16_t dt_ms;
} ink_point_t;

#pragma pack(pop)

int serialize_ink_frame(const ink_point_t* points, int count,
                       uint32_t pen_id, uint32_t seq,
                       uint8_t* buf, int buf_size) {
    int total = sizeof(frame_header_t) + sizeof(ink_data_frame_t)
               + count * sizeof(ink_point_t);
    if (buf_size < total) return -1;

    frame_header_t* hdr = (frame_header_t*)buf;
    hdr->magic = 0xAB; hdr->version = 0x01;
    hdr->type = 0x01;  hdr->flags = 0x01;

    ink_data_frame_t* frame = (ink_data_frame_t*)(buf + sizeof(frame_header_t));
    frame->pen_id      = htonl(pen_id);
    frame->sequence    = htonl(seq);
    frame->timestamp_ms = htonl(get_uptime_ms());
    frame->point_count = htons(count);

    memcpy(buf + sizeof(frame_header_t) + sizeof(ink_data_frame_t),
           points, count * sizeof(ink_point_t));
    return total;
}

```

E.3 断线重连指数退避

```

// reconnect_manager.c
static const uint32_t BACKOFF_TABLE[8] = {
    1000, 2000, 4000, 8000, 16000, 30000, 60000, 120000
};

static int    g_attempt = 0;
static uint32_t g_next_retry_ms = 0;

void mqtt_on_disconnect(int code) {
    LOG_WARN("MQTT disconnected code=%d attempt=%d", code, g_attempt);
    int idx = (g_attempt < 8) ? g_attempt : 7;
    g_next_retry_ms = get_uptime_ms() + BACKOFF_TABLE[idx];
    g_attempt++;
}

```

```

}

void mqtt_reconnect_tick(void) {
    if (mqtt_is_connected()) { g_attempt = 0; return; }
    if (get_uptime_ms() >= g_next_retry_ms) {
        LOG_INFO("Reconnecting MQTT #%d ...", g_attempt);
        mqtt_connect_async(MQTT_BROKER_HOST, MQTT_BROKER_PORT);
    }
}
}

```

附录F 补充技术规格

F.1 配置文件热加载

```

// config_watcher.c
#include <sys/inotify.h>

#define CONFIG_FILE "/etc/wrotech/gateway.conf"
#define EVENT_SIZE (sizeof(struct inotify_event))
#define BUF_LEN (1024 * (EVENT_SIZE + 16))

static int inotify_fd = -1;
static int watch_fd = -1;

void config_watcher_start(void (*on_reload)(void)) {
    inotify_fd = inotify_init1(IN_NONBLOCK);
    if (inotify_fd < 0) { perror("inotify_init1"); return; }

    watch_fd = inotify_add_watch(inotify_fd, CONFIG_FILE, IN_CLOSE_WRITE);
    if (watch_fd < 0) { perror("inotify_add_watch"); return; }

    // 在独立线程中监听文件变化
    pthread_t tid;
    pthread_create(&tid, NULL, config_watch_thread, on_reload);
    pthread_detach(tid);
}

static void* config_watch_thread(void* arg) {
    void (*on_reload)(void) = (void (*)(void))arg;
    char buf[BUF_LEN] __attribute__((aligned(8)));

    while (1) {
        fd_set fds;
        FD_ZERO(&fds);
        FD_SET(inotify_fd, &fds);

        struct timeval tv = {.tv_sec = 1, .tv_usec = 0};
        if (select(inotify_fd + 1, &fds, NULL, NULL, &tv) <= 0) continue;

        ssize_t len = read(inotify_fd, buf, BUF_LEN);
        if (len <= 0) continue;
    }
}

```



```

        for (char* ptr = buf; ptr < buf + len; ) {
            struct inotify_event* event = (struct inotify_event*)ptr;
            if (event->mask & IN_CLOSE_WRITE) {
                LOG_INFO("配置文件已修改, 热加载中...");
                on_reload();
            }
            ptr += EVENT_SIZE + event->len;
        }
    }
    return NULL;
}

```

F.2 设备心跳管理

```

// heartbeat.c
#define HEARTBEAT_INTERVAL_S    30    // 心跳间隔30秒
#define HEARTBEAT_TIMEOUT_S     90    // 超过90秒无心跳视为离线

typedef struct {
    uint32_t pen_id;
    uint32_t last_heartbeat_ms;
    bool online;
} pen_heartbeat_t;

static pen_heartbeat_t g_heartbeats[MAX_PENS];

void heartbeat_update(uint32_t pen_id) {
    for (int i = 0; i < MAX_PENS; i++) {
        if (g_heartbeats[i].pen_id == pen_id) {
            g_heartbeats[i].last_heartbeat_ms = get_uptime_ms();
            if (!g_heartbeats[i].online) {
                g_heartbeats[i].online = true;
                on_pen_online(pen_id);
            }
            return;
        }
    }
}

void heartbeat_check_all(void) {
    uint32_t now = get_uptime_ms();
    for (int i = 0; i < MAX_PENS; i++) {
        if (!g_heartbeats[i].pen_id) continue;
        uint32_t elapsed_s = (now - g_heartbeats[i].last_heartbeat_ms) / 1000;
        if (elapsed_s > HEARTBEAT_TIMEOUT_S && g_heartbeats[i].online) {
            g_heartbeats[i].online = false;
            on_pen_offline(g_heartbeats[i].pen_id);
        }
    }
}

```

附录G 补充技术规格

G.1 UDP广播设备发现

```
// udp_discovery.c
// UDP广播自动发现局域网内的网关设备
#include <sys/socket.h>
#include <netinet/in.h>

#define DISCOVERY_PORT 5678
#define DISCOVERY_MSG "WRITECH_GATEWAY_DISCOVERY"
#define RESPONSE_PREFIX "WRITECH_GATEWAY:"

int create_discovery_socket(void) {
    int sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (sock < 0) return -1;

    // 启用广播
    int broadcast = 1;
    setsockopt(sock, SOL_SOCKET, SO_BROADCAST, &broadcast, sizeof(broadcast));

    // 设置接收超时
    struct timeval tv = {.tv_sec = 2, .tv_usec = 0};
    setsockopt(sock, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv));

    return sock;
}

int send_discovery_broadcast(int sock) {
    struct sockaddr_in addr = {
        .sin_family = AF_INET,
        .sin_port = htons(DISCOVERY_PORT),
        .sin_addr.s_addr = INADDR_BROADCAST
    };

    return sendto(sock, DISCOVERY_MSG, strlen(DISCOVERY_MSG), 0,
        (struct sockaddr*)&addr, sizeof(addr));
}

// 网关接收发现请求后回复自己的信息
void handle_discovery_request(int sock, const struct sockaddr_in* client) {
    char response[128];
    snprintf(response, sizeof(response),
        "%s%s:%d", RESPONSE_PREFIX, get_local_ip(), MQTT_PORT);

    sendto(sock, response, strlen(response), 0,
        (struct sockaddr*)client, sizeof(*client));
}
```

G.2 日志轮转管理

```

// log_manager.c
#define LOG_MAX_SIZE_MB 10
#define LOG_MAX_FILES 5
#define LOG_DIR "/var/log/writtech"

void log_rotate_if_needed(void) {
    char current_log[256];
    snprintf(current_log, sizeof(current_log), "%s/gateway.log", LOG_DIR);

    struct stat st;
    if (stat(current_log, &st) != 0) return;

    // 超过10MB则轮转
    if (st.st_size < LOG_MAX_SIZE_MB * 1024 * 1024) return;

    // 删除最旧的日志
    char oldest[256];
    snprintf(oldest, sizeof(oldest), "%s/gateway.log.%d", LOG_DIR, LOG_MAX_FILES);
    unlink(oldest);

    // 重命名现有日志文件
    for (int i = LOG_MAX_FILES - 1; i >= 1; i--) {
        char src[256], dst[256];
        snprintf(src, sizeof(src), "%s/gateway.log.%d", LOG_DIR, i);
        snprintf(dst, sizeof(dst), "%s/gateway.log.%d", LOG_DIR, i + 1);
        rename(src, dst);
    }

    // 当前日志重命名为.1
    char backup[256];
    snprintf(backup, sizeof(backup), "%s/gateway.log.1", LOG_DIR);
    rename(current_log, backup);

    // 用gzip压缩旧日志
    char cmd[512];
    snprintf(cmd, sizeof(cmd), "gzip -q %s &", backup);
    system(cmd);
}

```