

自然写教室智能算力盒边缘计算软件 V1.0

软件鉴别材料 — 设计说明书

软件全称：自然写教室智能算力盒边缘计算软件

软件版本：V1.0

权利人：深圳自然写科技有限公司

文档类型：嵌入式软件设计说明书

文档编号：WRITECH-EDGE-DS-001

编制日期：2026年2月

密级：内部资料

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 运行环境与系统要求
 - 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 核心架构示意图
 - 2.4 数据设计
 - 2.5 接口设计
 - 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 main.cpp — 主程序入口与系统初始化
 - 3.2 inference/inference_engine.cpp — 推理引擎核心
 - 3.3 inference/model_manager.cpp — 模型管理模块
 - 3.4 inference/npu_scheduler.cpp — NPU/GPU调度器

- 3.5 communication/grpc_server.cpp — gRPC通信服务
- 3.6 communication/mqtt_client.cpp — MQTT状态上报
- 3.7 preprocessing/stroke_preprocessor.cpp — 笔迹预处理
- 3.8 config/edge_config.py — 配置管理模块
- 3.9 离线缓存与数据同步模块
- 3.10 集群管理与负载均衡模块
- 3.11 OTA升级模块
- 3.12 设备监控与运维模块
- 第四章 操作流程与使用步骤
- 4.1 设备安装与初始化配置
- 4.2 网络接入与云端注册
- 4.3 模型加载与推理验证
- 4.4 课堂教学使用流程
- 4.5 离线模式操作流程
- 4.6 OTA升级操作流程
- 4.7 集群管理操作流程
- 4.8 故障排查与日志查看
- 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 类与方法命名规范
- 附录A 硬件接口说明
- 附录B 术语表
- 附录C 版本历史

第一章 软件整体概述

1.1 软件简介与功能综述

自然写教室智能算力盒边缘计算软件（以下简称“算力盒软件”）是自然写互动课堂智能点阵笔系统的核心边缘计算组件，运行于部署在教室内的智能算力盒硬件设备之上。该软件将云端AI推理能力下沉至教室本地，实现在无网络或弱网络环境下的完整手写识别功能，大幅降低识别延迟，提升课堂实时交互体验。

算力盒软件的核心设计理念是“边缘智能、离线可用、云边协同”。软件在本地搭载轻量化手写识别模型，能够独立完成OCR文字识别、数学公式识别、笔顺分析等AI推理任务，同时通过与云端的协同机制，支持模型在线更新、数据延迟上传和集群统一调度。

主要功能模块综述：

功能模块	说明
端侧AI推理引擎	本地运行手写OCR识别、数学列式识别、笔顺分析AI模型
轻量化模型管理	管理本地AI模型文件的加载、版本切换与云端同步更新
笔迹数据预处理	对原始笔迹坐标数据进行去噪、归一化、笔画分割处理
实时识别结果分发	将推理结果实时推送至教室内黑板、PC、Pad等各终端
离线模式支持	断网环境下AI识别能力不降级，结果本地缓存待网络恢复后上传
云边协同通信	通过gRPC接收网关笔迹流，通过MQTT向云端上报状态
集群管理	支持校级多台算力盒组成集群，统一调度与负载均衡
OTA远程升级	支持固件和AI模型的远程在线升级，A/B分区无损升级
设备监控运维	实时监控GPU/NPU利用率、温度、推理QPS，支持远程运维

1.2 软件用途与适用场景

算力盒软件专为K-12基础教育互动课堂场景设计，主要解决以下核心问题：

场景一：农村及偏远地区学校 网络基础设施薄弱，课堂教学依赖本地AI能力。算力盒软件提供完整的离线识别能力，即使网络中断，课堂教学流程不受影响。学生的书写作业、笔顺练习、数学作答均可实时得到AI反馈。

场景二：城市学校大规模并发 一所学校可能同时进行多个年级的互动课堂教学，每间教室有40支点阵笔同时工作。集中式云端识别在高并发场景下存在延迟和拥塞风险。算力盒将计算压力分散到各教室，每间教室识别延迟 < 200ms（单次OCR）。

场景三：教学过程低延迟交互 学生书写后，教师和学生都希望立即看到识别结果和评分反馈。算力盒在本地完成推理，避免了网络往返时延，实现毫秒级响应，增强互动感。

场景四：数据安全与隐私保护 部分学校和家长对学生书写数据上传云端有顾虑。算力盒支持"本地优先"模式，识别计算在本地完成，原始笔迹数据可选择不上云，满足数据本地化合规要求。

适用硬件平台： – 搭载瑞芯微RK3588 NPU（6 TOPS算力）的ARM算力盒 – 搭载NVIDIA Jetson系列GPU的x86/ARM算力盒 – 标准x86工控机（支持OpenCL加速）

1.3 运行环境与系统要求

硬件环境：

配置项	最低要求	推荐配置
处理器	ARM Cortex-A55 4核 / x86 4核	RK3588 八核 / Jetson Orin
内存	4GB LPDDR4	8GB LPDDR4X
存储	32GB eMMC	64GB eMMC + 256GB NVMe
AI加速	NPU 1 TOPS 或 GPU 支持CUDA/OpenCL	NPU 6 TOPS 或 Jetson GPU
网络	100Mbps 有线以太网	千兆有线 + WiFi 6
操作系统	嵌入式 Linux（内核 4.19+）	Ubuntu 20.04 LTS（ARM/x86）

软件运行环境：

组件	版本要求	用途
Linux Kernel	4.19 以上	操作系统内核
RKNN Runtime	1.5.0 以上（瑞芯微平台）	NPU推理运行时
CUDA	11.4 以上（NVIDIA平台）	GPU推理加速
ONNX Runtime	1.13.0 以上	AI模型推理引擎
TensorRT	8.5 以上（NVIDIA平台）	模型加速优化
gRPC	1.50.0 以上	服务通信框架
Mosquitto Client	2.0 以上	MQTT通信
SQLite	3.38.0 以上	本地数据存储
Python	3.9 以上	管理API服务
Flask	2.2 以上	本地管理Web服务

资源占用：

资源	正常工作状态	峰值状态
内存占用	约1.5GB	约3GB
NPU/GPU利用率	30–60%	95%
CPU利用率	15–30%	60%
存储空间	约8GB（含模型）	约15GB（含缓存）
网络带宽	约5Mbps（上行）	约20Mbps

1.4 开发语言与技术规范

主要开发语言：

语言	版本	用途
C++	C++17	推理引擎、预处理、gRPC服务、NPU调度
Python	3.9	模型管理、配置服务、Flask管理API
Protocol Buffers	proto3	gRPC接口定义

代码规范： – C++遵循Google C++ Style Guide，命名采用下划线分隔（snake_case） – Python遵循PEP8规范，Docstring采用Google风格 – 所有公共接口需提供完整的doxygen注释 – 线程安全：多线程共享数据使用std::mutex或std::atomic保护 – 内存管理：推理引擎采用RAII原则，避免内存泄漏 – 错误处理：使用返回码与错误日志双重机制，关键路径不使用异常

构建工具链：

工具	版本	说明
CMake	3.20+	C++项目构建系统
GCC / Clang	GCC 9+ / Clang 12+	C++编译器
pip / conda	最新	Python依赖管理
Docker	20.10+	容器化打包与部署
Buildroot	2023.02	嵌入式Linux根文件系统构建

1.5 版本说明

版本	发布日期	主要变更
V0.5 Beta	2025年8月	基础推理框架搭建，单模型OCR识别
V0.8 Beta	2025年10月	增加数学识别、笔顺分析模型支持
V0.9 RC	2025年12月	增加离线缓存、集群管理基础框架
V1.0	2026年2月	正式版，支持完整功能，OTA升级稳定

第二章 系统架构与设计思路

2.1 总体架构设计

算力盒软件采用六层边缘AI推理分层架构，自下而上分别为：硬件加速层、推理框架层、模型管理层、业务服务层、通信层和管理层。各层职责清晰、接口明确，支持不同硬件平台的横向替换（瑞芯微NPU、NVIDIA GPU、OpenCL GPU均通过统一接口适配）。

管理层 (Management Layer)			
Flask管理API	SQLite配置库	日志服务	状态监控
通信层 (Communication Layer)			
gRPC Server (接收网关笔迹)	MQTT Client (云端状态同步)		
WebSocket推送 (识别结果分发)	mDNS集群发现		
业务服务层 (Business Service Layer)			
笔迹预处理	推理调度	结果分发	离线缓存
任务优先级队列	模型热切换	结果后处理	同步管理
模型管理层 (Model Management Layer)			
模型版本控制	动态加载	INT8量化	云端同步更新
A/B分区管理	模型加密存储	精度评估	回滚机制
推理框架层 (Inference Framework Layer)			
ONNX Runtime	TensorRT	PaddleLite	框架统一接口
硬件加速层 (Hardware Acceleration Layer)			
RKNN (瑞芯微NPU)	CUDA (NVIDIA GPU)	OpenCL (通用GPU)	

2.2 各层次详细说明

硬件加速层 (Hardware Acceleration Layer)

硬件加速层负责对接底层AI加速芯片的驱动和运行时环境，提供统一的硬件抽象接口供上层调用。软件通过编译时宏开关（`#ifdef PLATFORM_RKNN`、`#ifdef PLATFORM_CUDA`）选择目标平台的实现，做到“一套代码，多平台适配”。

支持的硬件加速方案：

- **RKNN (瑞芯微NPU)**：适用于RK3588等瑞芯微SoC，NPU算力6 TOPS，模型格式为 `.rknn`（由ONNX转换）
- **CUDA + TensorRT**：适用于NVIDIA Jetson系列，GPU并行计算，模型经TensorRT优化加速
- **OpenCL**：通用GPU加速方案，适用于Mali GPU、PowerVR GPU等ARM嵌入式GPU

推理框架层 (Inference Framework Layer)

推理框架层管理AI模型的加载与推理执行。软件定义了统一的 `IIInferenceEngine` 抽象接口，不同框架（ONNX Runtime / TensorRT / PaddleLite）各自实现该接口。业务层调用时无需关心底层框架差异。

```
IIInferenceEngine (抽象接口)
├── ONNXInferenceEngine    → ONNX Runtime实现
├── TRTInferenceEngine     → TensorRT实现 (NVIDIA平台)
└── PaddleLiteEngine       → PaddleLite实现 (ARM平台)
```

模型管理层 (Model Management Layer)

模型管理层负责AI模型文件的全生命周期管理： – 模型文件存储在 `/opt/writtech/models/` 目录，按类型分目录存放 (`ocr/math/stroke_order`) – 每个模型目录下维护 `model_meta.json` 元信息文件，记录版本号、精度指标、部署状态 – 支持A（当前运行）和B（待切换）双版本共存，切换时无需重启进程 – 模型文件AES-256加密存储，运行时内存解密加载，防止模型被窃取

业务服务层 (Business Service Layer)

业务服务层是软件的核心业务逻辑所在，包含以下主要组件： – `StrokePreprocessor`：笔迹坐标去噪、归一化和笔画分割 – `InferenceScheduler`：基于优先级的推理任务调度（实时识别优先于批量处理） – `ResultDistributor`：将推理结果分发至对应终端（按学生ID路由） – `OfflineCacheManager`：管理断网期间的结果暂存和恢复上传

通信层 (Communication Layer)

通信层负责软件与外部系统的所有数据交换： – `gRPC Server`：监听来自教室网关的笔迹数据流（双向流式RPC） – `WebSocket Server`：向教室内各终端（黑板/PC/Pad）推送识别结果 – `MQTT Client`：向云端上报算力盒运行状态（GPU利用率/温度/推理QPS） – `mDNS服务`：在局域网内广播算力盒服务，支持自动发现和集群组建

管理层 (Management Layer)

管理层提供本地运维和远程管理能力： – `Flask管理API`：提供RESTful接口用于查询状态、切换模型、查看日志 – `SQLite配置库`：存储设备配置、模型元信息、推理统计数据 – `日志服务`：结构化日志输出，支持日志级别动态调整和远程传输 – `状态监控`：定时采样GPU/CPU/内存/温度指标，触发阈值告警

2.3 核心架构示意图

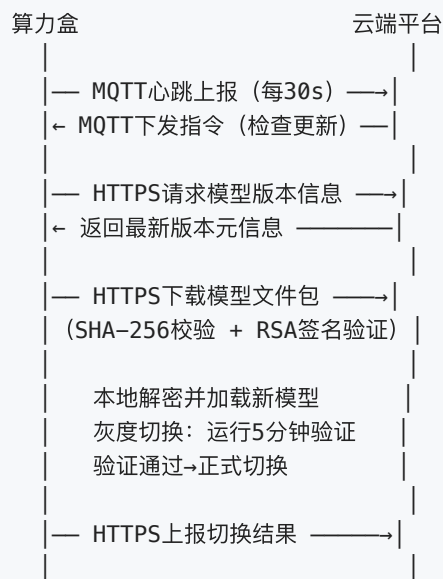
推理数据流完整路径：



业务服务层

- StrokePreprocessor (笔迹预处理)
 - 坐标去噪 (中值滤波)
 - 坐标归一化 ($[0,1]$ 区间缩放)
 - 笔画分割 (落笔/抬笔事件切分)
- InferenceScheduler (推理调度)
 - 实时任务队列 (优先级HIGH, 延迟 $\leq 200\text{ms}$)
 - 批量任务队列 (优先级NORMAL, 延迟 $\leq 2\text{s}$)
- InferenceEngine (推理执行)
 - OCR识别: 手写文字 \rightarrow 文本
 - 数学识别: 列式 \rightarrow LaTeX+结果
 - 笔顺评分: 笔画顺序 \rightarrow 分数+错误位置
- ResultDistributor (结果分发)
 - WebSocket推送 \rightarrow 智慧黑板
 - WebSocket推送 \rightarrow 教师PC
 - gRPC回调 \rightarrow 云端 (MQTT异步上报)

云边协同时序图:



2.4 数据设计

模型文件存储结构:

```
/opt/wrotech/
├── models/
│   ├── ocr/
│   │   ├── model_a.rknn    # 当前运行版本 (加密)
│   │   ├── model_b.rknn    # 待切换版本 (加密)
│   │   └── model_meta.json  # 模型元信息
│   └── math/
```



```
├── model_a.onnx
├── model_b.onnx
├── model_meta.json
├── stroke_order/
│   ├── model_a.rknn
│   └── model_meta.json
├── cache/
│   ├── offline_results.db      # 离线结果缓存 (SQLite)
│   └── task_queue.db          # 任务队列持久化 (SQLite)
├── config/
│   └── edge_config.json        # 设备配置文件
├── logs/
│   ├── inference.log           # 推理日志 (每日轮转)
│   ├── comm.log                # 通信日志
│   └── system.log              # 系统运行日志
```

SQLite数据库表结构：

model_registry 表（模型注册表）：

字段名	类型	说明
id	INTEGER PRIMARY KEY	自增主键
model_type	TEXT	模型类型 (ocr/math/stroke_order)
version	TEXT	版本号 (如 v2.1.3)
file_path	TEXT	加密模型文件路径
accuracy	REAL	验证集精度指标
file_size	INTEGER	文件大小 (字节)
is_active	INTEGER	是否当前激活 (0/1)
partition	TEXT	分区标识 (A/B)
created_at	TEXT	入库时间 (ISO8601)
updated_at	TEXT	最后更新时间

offline_result_cache 表（离线结果缓存）：

字段名	类型	说明
id	INTEGER PRIMARY KEY	自增主键
student_id	TEXT	学生唯一标识
assignment_id	TEXT	作业标识

字段名	类型	说明
result_type	TEXT	结果类型 (ocr/math/stroke)
result_json	TEXT	识别结果JSON序列化数据
infer_time	TEXT	推理时间戳
is_synced	INTEGER	是否已同步云端 (0/1)
retry_count	INTEGER	重试上传次数

inference_stats 表 (推理统计):

字段名	类型	说明
id	INTEGER PRIMARY KEY	自增主键
model_type	TEXT	模型类型
latency_ms	INTEGER	推理延迟 (毫秒)
gpu_util	REAL	GPU/NPU利用率 (百分比)
temperature	REAL	设备温度 (摄氏度)
timestamp	TEXT	采样时间

device_config 表 (设备配置):

字段名	类型	说明
key	TEXT PRIMARY KEY	配置键
value	TEXT	配置值
updated_at	TEXT	更新时间

内存数据结构 (C++):

```
// 推理任务数据结构 (inference/inference_engine.h)
struct InferenceTask {
    std::string task_id;           // 任务唯一ID (UUID)
    std::string student_id;        // 学生ID
    std::string assignment_id;      // 作业ID
    InferType infer_type;          // 推理类型 (OCR/MATH/STROKE_ORDER)
    int priority;                   // 优先级 (0=实时, 1=批量)
    std::vector<StrokePoint> strokes; // 预处理后的笔迹数据
    int64_t received_ts;           // 接收时间戳 (毫秒)
    int64_t deadline_ts;           // 截止时间戳 (超时丢弃)
};
```

```
// 笔迹坐标点 (preprocessing/stroke_preprocessor.h)
struct StrokePoint {
    float    x;           // 归一化X坐标 [0.0, 1.0]
    float    y;           // 归一化Y坐标 [0.0, 1.0]
    float    pressure;    // 压感 [0.0, 1.0]
    int64_t  timestamp;   // 毫秒时间戳
    bool     pen_up;      // 是否为抬笔事件
};

// 推理结果数据结构
struct InferenceResult {
    std::string task_id;    // 对应任务ID
    std::string student_id; // 学生ID
    InferType  result_type; // 结果类型
    bool       success;     // 推理是否成功
    std::string result_json; // 结果JSON字符串
    float      confidence;  // 置信度 [0.0, 1.0]
    int32_t    latency_ms;  // 实际推理耗时 (毫秒)
};
```

2.5 接口设计

gRPC接口定义 (proto/inference_service.proto):

```
syntax = "proto3";
package writech.edge;

// 算力盒推理服务接口
service InferenceService {
    // 流式接收笔迹, 流式返回推理结果 (双向流式RPC)
    rpc ProcessStroke (stream StrokeRequest) returns (stream InferenceResponse);

    // 单次识别请求 (一元RPC)
    rpc RecognizeOnce (RecognizeRequest) returns (InferenceResponse);

    // 查询算力盒运行状态
    rpc GetStatus (StatusRequest) returns (StatusResponse);
}

// 笔迹数据请求
message StrokeRequest {
    string task_id = 1;           // 任务ID
    string student_id = 2;       // 学生ID
    string assignment_id = 3;     // 作业ID
    InferType type = 4;          // 推理类型
    repeated Point points = 5;    // 笔迹坐标列表
    int64 timestamp = 6;         // 时间戳 (毫秒)
}

// 坐标点
message Point {
    float x = 1;
    float y = 2;
}
```

```

    float pressure = 3;
    bool pen_up = 4;
}

// 推理类型枚举
enum InferType {
    OCR          = 0; // 文字OCR识别
    MATH          = 1; // 数学列式识别
    STROKE_ORDER = 2; // 笔顺分析
    WRITING_QUALITY = 3; // 书写质量评测
}

// 推理响应
message InferenceResponse {
    string task_id = 1; // 任务ID
    bool success = 2; // 是否成功
    string result_json = 3; // 结果JSON（根据type解析）
    float confidence = 4; // 置信度
    int32 latency_ms = 5; // 推理耗时（毫秒）
    string error_msg = 6; // 错误信息（失败时）
}

// 状态查询响应
message StatusResponse {
    string device_id = 1; // 算力盒设备ID
    float gpu_util = 2; // GPU/NPU利用率（%）
    float temperature = 3; // 设备温度（℃）
    int32 queue_depth = 4; // 当前任务队列深度
    float avg_latency_ms = 5; // 过去1分钟平均推理延迟
    int32 active_models = 6; // 当前激活模型数量
    bool offline_mode = 7; // 是否处于离线模式
}

```

MQTT主题设计：

主题	方向	QoS	说明
edgebox/{device_id}/status	盒→云	1	每30秒上报设备状态 (CPU/GPU/温度/QPS)
edgebox/{device_id}/command	云→盒	1	云端下发管理指令（重启/模型切换/OTA触发）
edgebox/{device_id}/model/sync	云→盒	1	模型更新通知（包含新版本信息）
edgebox/{device_id}/alarm	盒→云	2	设备告警（过热/推理失败/OOM）
school/{school_id}/edgebox/discover	盒→局域网	0	组播广播，用于集群成员发现

Flask本地管理API：

接口路径	方法	说明
/api/status	GET	查询算力盒当前运行状态
/api/models	GET	列出所有已加载模型及版本
/api/models/switch	POST	切换激活模型版本（A/B切换）
/api/infer/test	POST	测试推理接口（上传笔迹数据）
/api/cache/stats	GET	查看离线缓存统计信息
/api/cache/sync	POST	手动触发离线数据同步上传
/api/logs	GET	查看最近500行日志
/api/config	GET/PUT	查看/修改设备配置
/api/restart	POST	重启推理服务

2.6 安全设计

模型文件安全：

AI模型是核心知识产权资产。所有模型文件均采用AES-256-GCM加密存储，解密密钥通过设备硬件序列号派生（PBKDF2算法），绑定硬件设备，拷贝到其他设备无法使用。推理运行时内存中解密加载，推理完成后密钥归零清除。

密钥派生流程：
设备SN + 固定盐值 (salt) —PBKDF2-HMAC-SHA256—> 256bit主密钥
主密钥 + 模型文件IV —AES-256-GCM—> 加密模型文件

通信安全：

- gRPC通信启用mTLS双向认证，算力盒和网关均需持有系统颁发的X.509证书
- MQTT通信使用TLS 1.3加密，证书由云端证书服务签发
- Flask管理API仅监听 127.0.0.1 或 192.168.x.x 局域网地址，不暴露外网

OTA安全机制：

升级包安全验证流程：
1. 下载升级包（HTTPS，服务端证书验证）
2. SHA-256文件完整性校验
3. RSA-2048签名验证（使用内置公钥）
4. 写入B分区（不覆盖当前A分区）
5. 下次启动时Bootloader验证B分区签名
6. 验证通过→切换到B分区启动

7. 运行10分钟无异常→确认升级成功，A分区标记为备份
8. 运行异常→自动回滚至A分区

运行隔离：

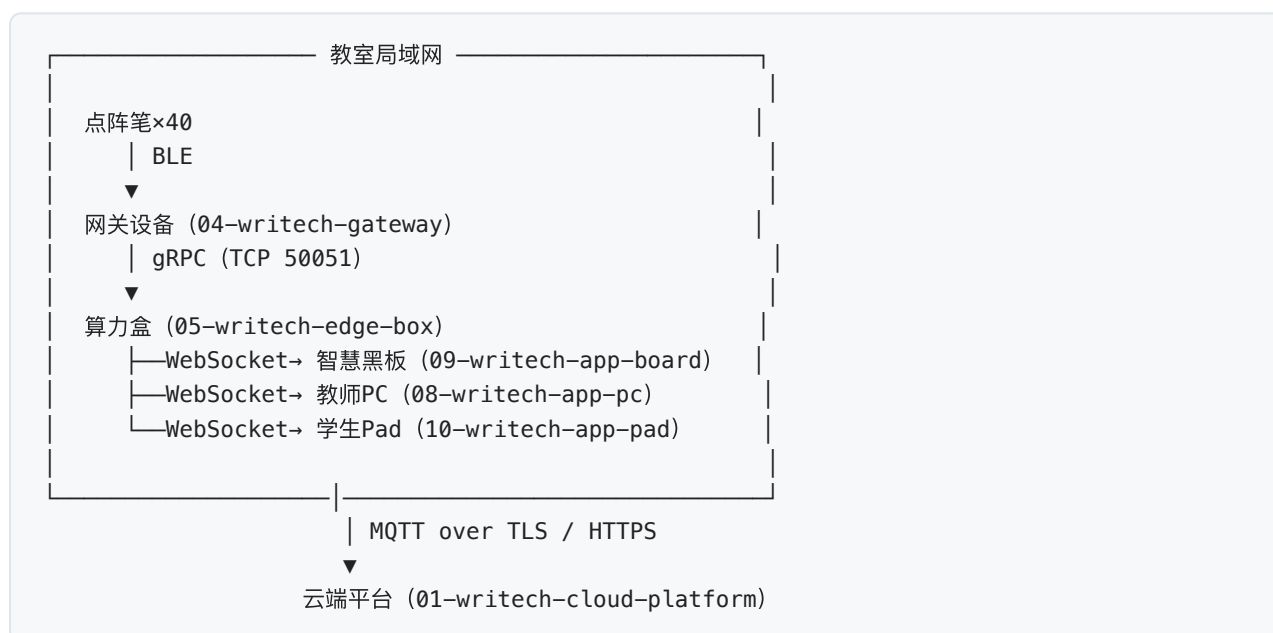
推理进程与管理进程（Flask）运行在独立的Linux进程中，通过Unix Domain Socket通信。推理进程采用 seccomp 系统调用过滤，仅允许必要的系统调用，防止漏洞利用。

物理安全：

每台算力盒烧录唯一设备ID和证书，与云端注册信息绑定。未经注册的设备即使接入学校网络，也无法与云端建立认证连接，确保系统不被非授权设备接管。

2.7 部署架构

单教室部署模式（标准配置）：



校级集群部署模式（大规模部署）：



A/B分区与存储布局：

eMMC存储分区规划：

Bootloader（8MB）	
系统根分区 /（rootfs）（8GB）	
App分区A - 推理软件当前版本（2GB）	
App分区B - 推理软件备份/待升级版本（2GB）	
模型存储分区A - 当前激活模型（8GB）	
模型存储分区B - 备份/待升级模型（8GB）	
数据分区 /data - 缓存/日志/配置（剩余空间）	

第三章 核心模块功能详细说明

3.1 main.cpp — 主程序入口与系统初始化

main.cpp 是算力盒软件的启动入口，负责整个软件的初始化流程编排和信号处理。

初始化流程：

```
main()
|
|— 1. 解析命令行参数（配置文件路径、日志级别、平台选择）
|
|— 2. 初始化日志系统（spdlog）
|   |— 创建滚动日志文件（每日轮转，保留7天）
|   |— 初始化控制台日志输出
|
|— 3. 加载配置文件（EdgeConfig::getInstance()）
|   |— 读取 edge_config.json
|   |— 初始化 SQLite 配置数据库
|
|— 4. 硬件平台探测与初始化
|   |— 探测NPU/GPU型号（读取 /proc/device-tree 或 nvidia-smi）
|   |— 加载对应硬件加速驱动（rknn_init / CUDA初始化）
|
|— 5. 初始化推理引擎（InferenceEngine::create(platform)）
|   |— 加载OCR模型（model_a.rknn, AES解密）
|   |— 加载数学识别模型
```

- └─ 加载笔顺分析模型
- └─ 6. 启动业务服务
 - └─ InferenceScheduler (推理调度线程池, 4个worker线程)
 - └─ ResultDistributor (结果分发线程)
 - └─ OfflineCacheManager (离线缓存管理线程)
- └─ 7. 启动通信服务
 - └─ gRPC Server (端口 50051, mTLS配置)
 - └─ WebSocket Server (端口 8080, 结果推送)
 - └─ MQTT Client (连接云端Broker, TLS配置)
- └─ 8. 启动管理服务
 - └─ Flask管理API (Python子进程, 端口 5000)
 - └─ mDNS服务广播 ("_writech-edge._tcp")
- └─ 9. 注册信号处理 (SIGTERM/SIGINT → 优雅退出)
- └─ 10. 进入主事件循环 (阻塞等待退出信号)

优雅退出流程:

收到SIGTERM/SIGINT信号后, 软件按以下顺序执行清理: 1. 停止接受新的gRPC请求 (停止Accept新连接) 2. 等待正在处理的推理任务完成 (超时5秒后强制终止) 3. 将未上报的离线结果刷写到SQLite 4. 关闭MQTT连接 (发送DISCONNECT包) 5. 释放NPU/GPU推理资源 6. 关闭日志文件 (flush缓冲区) 7. 退出进程

3.2 inference/inference_engine.cpp — 推理引擎核心

推理引擎是整个软件的计算核心, 负责将预处理后的笔迹数据送入AI模型执行推理, 并返回识别结果。

类结构设计:

```
// 推理引擎抽象接口 (inference/inference_engine.h)
class IIInferenceEngine {
public:
    virtual ~IIInferenceEngine() = default;

    // 工厂方法, 根据平台创建对应引擎实例
    static std::unique_ptr<IIInferenceEngine> create(
        const std::string& platform); // "rknn" / "cuda" / "opencl"

    // 初始化推理引擎, 加载模型文件
    virtual bool initialize(const ModelConfig& config) = 0;

    // 执行单次OCR推理
    virtual InferenceResult inferOCR(
        const std::vector<StrokePoint>& strokes) = 0;

    // 执行数学列式识别推理
```



```

virtual InferenceResult inferMath(
    const std::vector<StrokePoint>& strokes) = 0;

// 执行笔顺分析推理
virtual InferenceResult inferStrokeOrder(
    const std::string& target_char,
    const std::vector<StrokePoint>& strokes) = 0;

// 热切换模型（切换A/B分区的模型文件，不重启进程）
virtual bool switchModel(const std::string& model_type,
    const std::string& new_model_path) = 0;

// 释放推理资源
virtual void shutdown() = 0;
};

```

RKNN引擎实现（推理执行核心）：

```

// inference/rknn_engine.cpp (RKNN平台实现)
InferenceResult RKNNEngine::inferOCR(
    const std::vector<StrokePoint>& strokes) {

    auto start_ts = std::chrono::steady_clock::now();

    // Step 1: 将笔迹坐标渲染为灰度图像张量
    // 画布大小：224×224像素，笔画宽度3像素
    cv::Mat canvas = renderStrokesToImage(strokes, 224, 224);

    // Step 2: 图像归一化（减均值/除标准差，与训练预处理一致）
    // mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]
    cv::Mat normalized = normalizeImage(canvas);

    // Step 3: 构建RKNN输入
    rknn_input inputs[1];
    inputs[0].index = 0;
    inputs[0].type = RKNN_TENSOR_FLOAT32;
    inputs[0].size = 224 * 224 * 3 * sizeof(float);
    inputs[0].fmt = RKNN_TENSOR_NHWC;
    inputs[0].buf = normalized.data;

    int ret = rknn_inputs_set(ctx_, 1, inputs);
    if (ret != RKNN_SUCC) {
        return makeErrorResult("rknn_inputs_set failed: " +
            std::to_string(ret));
    }

    // Step 4: 执行NPU推理
    ret = rknn_run(ctx_, nullptr);
    if (ret != RKNN_SUCC) {
        return makeErrorResult("rknn_run failed: " +
            std::to_string(ret));
    }

    // Step 5: 获取推理输出
    rknn_output outputs[1];
}

```

```

    outputs[0].want_float = 1;
    outputs[0].index = 0;
    ret = rknn_outputs_get(ctx_, 1, outputs, nullptr);

    // Step 6: 后处理 (CTC解码, 将输出概率矩阵转换为文字)
    std::string recognized_text = ctcDecode(
        static_cast<float*>(outputs[0].buf),
        outputs[0].size / sizeof(float));

    float confidence = calculateConfidence(
        static_cast<float*>(outputs[0].buf));

    rknn_outputs_release(ctx_, 1, outputs);

    // Step 7: 计算推理延迟
    auto end_ts = std::chrono::steady_clock::now();
    int32_t latency = std::chrono::duration_cast<
        std::chrono::milliseconds>(end_ts - start_ts).count();

    // 记录推理统计
    stats_collector_>record(ModelType::OCR, latency,
        getNPUUtilization());

    return InferenceResult{
        .success = true,
        .result_json = buildOCRResultJson(recognized_text, confidence),
        .confidence = confidence,
        .latency_ms = latency
    };
}

```

性能优化设计：

优化策略	实现方式	效果
模型量化	INT8量化（精度损失<1%）	推理速度提升2-3倍
批处理推理	同一时刻多个学生笔迹合批推理	GPU利用率提升30%
内存复用	推理输入/输出缓冲区预分配复用	减少内存分配开销50%
模型预热	启动时执行10次dummy推理	消除首次推理延迟抖动
CPU+NPU流水线	预处理（CPU）与上批推理（NPU）并行执行	端到端延迟降低20%

3.3 inference/model_manager.cpp — 模型管理模块

模型管理模块负责AI模型文件的全生命周期管理，包括模型注册、加密存储、版本切换和云端同步更新。

模型版本切换流程（A/B热切换）：

收到云端模型更新通知 (MQTT)

- └─ 1. 检查B分区磁盘空间 (是否足够存放新模型)
- └─ 2. HTTPS下载新模型文件到/tmp/目录
- └─ 3. 验证完整性 (SHA-256与服务端校验和比对)
- └─ 4. 验证签名 (RSA-2048公钥验证模型包签名)
- └─ 5. AES解密新模型 (临时缓冲区)
- └─ 6. 写入B分区 (/opt/wrotech/models/{type}/model_b.rknn)
- └─ 7. 更新model_meta.json (B分区版本信息)
- └─ 8. 通知InferenceEngine加载B分区模型 (不切换激活)
- └─ 9. 执行模型验证推理 (使用标准测试集, 验证精度)
- └─ 10. 验证通过→原子切换active字段A→B
验证失败→删除B分区文件, 保留A分区, 上报失败告警
- └─ 11. 向云端确认升级结果 (MQTT上报)

模型注册表查询示例 (Python管理API):

```
# config/edge_config.py - 模型管理相关逻辑
import sqlite3, json
from pathlib import Path

class ModelManager:
    def __init__(self, db_path: str):
        self.db_path = db_path
        self.conn = sqlite3.connect(db_path)

    def get_active_model(self, model_type: str) -> dict:
        """获取当前激活的模型元信息"""
        cursor = self.conn.cursor()
        cursor.execute("""
            SELECT version, file_path, accuracy, is_active, partition
            FROM model_registry
            WHERE model_type = ? AND is_active = 1
            ORDER BY updated_at DESC LIMIT 1
        """, (model_type,))
        row = cursor.fetchone()
        if row:
            return {
                "version": row[0], "file_path": row[1],
                "accuracy": row[2], "partition": row[4]
            }
        return None

    def switch_model(self, model_type: str, target_partition: str) -> bool:
```

```

        """切换激活分区 (A→B 或 B→A) """
    try:
        cursor = self.conn.cursor()
        # 原子性更新: 先清除所有active, 再设置目标partition为active
        cursor.execute("""
            UPDATE model_registry
            SET is_active = 0
            WHERE model_type = ?
        """, (model_type,))
        cursor.execute("""
            UPDATE model_registry
            SET is_active = 1
            WHERE model_type = ? AND partition = ?
        """, (model_type, target_partition))
        self.conn.commit()
        return True
    except sqlite3.Error as e:
        self.conn.rollback()
        return False

```

3.4 inference/npuscheduler.cpp — NPU/GPU调度器

NPU/GPU调度器是并发推理请求的核心调度组件，实现优先级调度、资源限流和超时管理。

调度器设计：

```

// inference/npuscheduler.cpp
class NPUScheduler {
public:
    NPUScheduler(int num_workers = 4);

    // 提交推理任务到调度队列
    // priority: 0=实时 (课堂进行中), 1=批量 (课后批改)
    std::future<InferenceResult> submit(
        InferenceTask task, int priority = 0);

    // 获取当前队列深度
    int getQueueDepth() const;

    // 获取平均推理延迟 (最近1分钟)
    float getAvgLatencyMs() const;

private:
    // 优先级队列 (最小堆, priority小的优先)
    std::priority_queue<
        InferenceTask,
        std::vector<InferenceTask>,
        TaskComparator> task_queue_;

    std::mutex queue_mutex_;
    std::condition_variable cv_;

    // Worker线程池
    std::vector<std::thread> workers_;

```

```

std::atomic<bool> running_{true};

// 推理引擎实例
std::shared_ptr<IIInferenceEngine> engine_;

// Worker线程函数
void workerLoop();

// 超时检测（定时清理超期任务）
void timeoutChecker();
};

```

任务优先级策略：

任务来源	优先级	超时时间	说明
课堂实时书写（笔迹Notify触发）	0（最高）	500ms	学生正在书写中，需立即反馈
教师互动答题收卷	0（最高）	500ms	收卷后立即统计展示
作业批量批改	1（普通）	5000ms	课后批改，可稍有延迟
模型验证推理	2（低）	30000ms	新模型精度验证，不影响教学

3.5 communication/grpc_server.cpp — gRPC通信服务

gRPC服务是算力盒软件接收笔迹数据的入口，支持与网关软件之间的双向流式通信。

流式RPC实现：

```

// communication/grpc_server.cpp
class InferenceServiceImpl : public InferenceService::Service {
public:
    // 双向流式RPC：接收笔迹流，实时返回推理结果
    grpc::Status ProcessStroke(
        grpc::ServerContext* context,
        grpc::ServerReaderWriter<InferenceResponse,
                                StrokeRequest>* stream) override {

        StrokeRequest request;
        std::string current_student_id;
        std::vector<StrokePoint> stroke_buffer;

        while (stream->Read(&request)) {
            // 检查任务超时（防止僵死连接占用资源）
            if (isTaskTimeout(request.task_id())) {
                continue;
            }

            // 累积笔迹点
            for (const auto& pt : request.points()) {
                stroke_buffer.push_back({

```

```

        pt.x(), pt.y(), pt.pressure(),
        request.timestamp(), pt.pen_up()
    });
}

// 检测到抬笔事件 → 触发推理
if (!stroke_buffer.empty() &&
    stroke_buffer.back().pen_up) {

    // 预处理
    auto preprocessed = preprocessor_>process(stroke_buffer);
    stroke_buffer.clear();

    // 构建推理任务并提交到调度器
    InferenceTask task{
        .task_id = request.task_id(),
        .student_id = request.student_id(),
        .infer_type = (InferType)request.type(),
        .priority = 0, // 课堂实时, 最高优先级
        .strokes = preprocessed,
        .received_ts = getCurrentMs(),
        .deadline_ts = getCurrentMs() + 500
    };

    auto future = scheduler_>submit(task, 0);

    // 异步等待结果 (最多等待400ms)
    if (future.wait_for(std::chrono::milliseconds(400)) ==
        std::future_status::ready) {

        InferenceResult result = future.get();
        InferenceResponse response;
        response.set_task_id(result.task_id);
        response.set_success(result.success);
        response.set_result_json(result.result_json);
        response.set_confidence(result.confidence);
        response.set_latency_ms(result.latency_ms);

        stream->Write(response);

        // 同步触发结果分发到教室终端
        distributor_>distribute(request.student_id(), result);

        // 离线模式下缓存结果
        if (offline_mode_) {
            cache_manager_>cache(request.student_id(),
                                request.assignment_id(), result);
        }
    }
}

return grpc::Status::OK;
}
};

```

连接管理与限流：

- 最大并发连接数：100（每个网关建立1个长连接）
- 单连接最大流式请求速率：1000点/秒（超出则背压限流）
- 空闲连接超时：300秒（5分钟无数据则关闭连接）
- 服务器端mTLS：客户端（网关）须持有系统颁发证书

3.6 communication/mqtt_client.cpp — MQTT状态上报

MQTT客户端负责算力盒与云端平台的状态同步，采用Eclipse Mosquitto客户端库实现。

状态上报数据格式（每30秒上报一次）：

```
{
  "device_id": "edge-box-cn-hz-001",
  "school_id": "school_hangzhou_001",
  "timestamp": 1706845200000,
  "hardware": {
    "npu_util_pct": 45.2,
    "cpu_util_pct": 18.5,
    "memory_used_mb": 1820,
    "temperature_c": 52.3,
    "storage_free_gb": 18.4
  },
  "inference": {
    "total_requests": 1250,
    "success_rate_pct": 99.8,
    "avg_latency_ms": 87,
    "p99_latency_ms": 178,
    "queue_depth": 3
  },
  "models": {
    "ocr_version": "v2.1.3",
    "math_version": "v1.5.0",
    "stroke_order_version": "v1.2.1"
  },
  "connectivity": {
    "offline_mode": false,
    "pending_sync_count": 0,
    "last_cloud_sync_ts": 1706845170000
  }
}
```

断线重连策略：

```
// 指数退避重连（最大重连间隔：5分钟）
void MQTTClient::reconnect() {
    int retry = 0;
    while (!connected_ && running_) {
        int wait_ms = std::min(1000 * (1 << retry), 300000);
```

```

LOG_WARN("MQTT disconnected, retry in {}ms (attempt {})",
        wait_ms, retry + 1);
std::this_thread::sleep_for(
    std::chrono::milliseconds(wait_ms));

if (mosquitto_reconnect(mosq_) == MOSQ_ERR_SUCCESS) {
    connected_ = true;
    LOG_INFO("MQTT reconnected successfully");
    // 重订阅所有主题
    for (const auto& topic : subscribed_topics_) {
        mosquitto_subscribe(mosq_, nullptr,
                            topic.c_str(), 1);
    }
    break;
}
retry = std::min(retry + 1, 8); // 最大退避2^8=256秒
}
}

```

3.7 preprocessing/stroke_preprocessor.cpp — 笔迹预处理

笔迹预处理模块对原始笔迹坐标数据进行一系列数学处理，提升AI推理的识别精度。

处理管道 (Processing Pipeline):

```

原始笔迹坐标流 (来自网关)
|
▼ 步骤1: 去抖动滤波
|   中值滤波 (窗口大小3), 消除传感器噪声抖动
|   过滤掉 pressure < 0.05 的无效采样点
|
▼ 步骤2: 重采样 (等时间间隔→等空间间隔)
|   将原始时序点重采样为沿笔画路径均匀分布的点集
|   目标密度: 每3像素一个点 (适配模型训练时的采样率)
|
▼ 步骤3: 笔画分割
|   按"抬笔事件 (pen_up=true)"将连续点流切分为独立笔画
|   过滤掉点数 < 5 的无效笔画 (抖动产生的伪笔画)
|
▼ 步骤4: 包围盒归一化
|   计算所有笔画的最小包围盒
|   将坐标缩放到 [0.0, 1.0] × [0.0, 1.0] 范围
|   保持长宽比 (短边padding到正方形后缩放)
|
▼ 步骤5: 笔顺序排序 (仅用于笔顺分析任务)
|   按笔画开始时间戳排序, 确保笔画顺序正确
|
▼ 预处理完成的笔迹张量

```

关键算法实现:


```
// preprocessing/stroke_preprocessor.cpp
std::vector<StrokePoint> StrokePreprocessor::normalizeToUnitBox(
    const std::vector<StrokePoint>& strokes) {

    // 计算所有点的边界框
    float min_x = FLT_MAX, max_x = FLT_MIN;
    float min_y = FLT_MAX, max_y = FLT_MIN;

    for (const auto& p : strokes) {
        min_x = std::min(min_x, p.x);
        max_x = std::max(max_x, p.x);
        min_y = std::min(min_y, p.y);
        max_y = std::max(max_y, p.y);
    }

    float width  = max_x - min_x;
    float height = max_y - min_y;
    float size   = std::max(width, height);

    // 防止除零（单点笔画）
    if (size < 1e-6f) size = 1.0f;

    // 居中归一化（短边居中padding）
    float offset_x = (size - width) / 2.0f;
    float offset_y = (size - height) / 2.0f;

    std::vector<StrokePoint> normalized;
    normalized.reserve(strokes.size());

    for (const auto& p : strokes) {
        normalized.push_back({
            (p.x - min_x + offset_x) / size,
            (p.y - min_y + offset_y) / size,
            p.pressure,
            p.timestamp,
            p.pen_up
        });
    }

    return normalized;
}
```

3.8 config/edge_config.py — 配置管理模块

配置管理模块负责算力盒所有运行参数的统一管理，支持本地JSON文件配置和远程动态配置下发。

配置文件示例（edge_config.json）：

```
{
  "device": {
    "device_id": "edge-box-cn-hz-001",
```

```

    "school_id": "school_hangzhou_001",
    "classroom_ids": ["room-a101", "room-a102"],
    "hardware_platform": "rknn"
  },
  "inference": {
    "max_concurrent_tasks": 8,
    "realtime_timeout_ms": 500,
    "batch_timeout_ms": 5000,
    "worker_threads": 4,
    "batch_size": 4
  },
  "models": {
    "model_base_path": "/opt/wrotech/models",
    "ocr_model": "ocr/model_a.rknn",
    "math_model": "math/model_a.onnx",
    "stroke_order_model": "stroke_order/model_a.rknn",
    "model_encrypt_key_derivation": "PBKDF2-HMAC-SHA256"
  },
  "communication": {
    "grpc_port": 50051,
    "grpc_tls_cert": "/etc/wrotech/certs/server.crt",
    "grpc_tls_key": "/etc/wrotech/certs/server.key",
    "grpc_ca_cert": "/etc/wrotech/certs/ca.crt",
    "websocket_port": 8080,
    "mqtt_broker": "mqtt.wrotech.cn",
    "mqtt_port": 8883,
    "mqtt_client_cert": "/etc/wrotech/certs/mqtt.crt",
    "mqtt_heartbeat_interval_s": 30
  },
  "storage": {
    "db_path": "/opt/wrotech/cache/edge.db",
    "log_path": "/opt/wrotech/logs",
    "log_level": "INFO",
    "log_max_size_mb": 50,
    "log_keep_days": 7
  },
  "cloud": {
    "cloud_api_base": "https://api.wrotech.cn",
    "model_sync_check_interval_s": 3600,
    "offline_cache_max_mb": 512,
    "sync_batch_size": 100
  }
}

```

3.9 离线缓存与数据同步模块

离线缓存模块在网络中断时保存推理结果，网络恢复后批量上传至云端，确保数据不丢失。

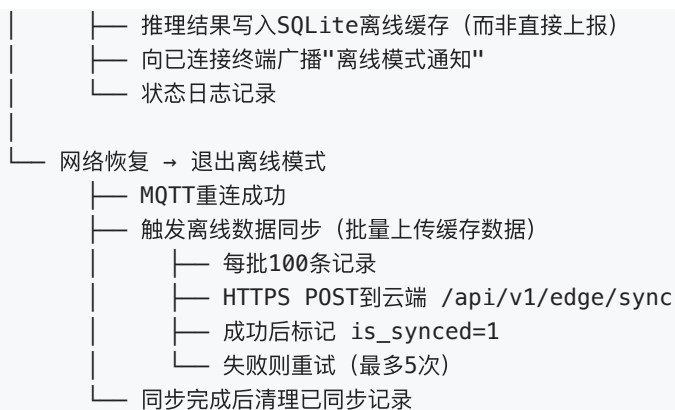
离线模式触发与恢复：

网络状态检测（每10秒ping云端API）：

```

|
|—— 连续3次失败 → 进入离线模式
|      |—— MQTT重连（指数退避）

```



缓存容量管理：

- 默认最大缓存容量：512MB
- 超过80%时触发告警（MQTT上报）
- 超过95%时启用FIFO淘汰策略（删除最旧的已推理结果）
- 优先保留未同步的最新结果，防止重要数据丢失

3.10 集群管理与负载均衡模块

多台算力盒可组成校级集群，实现统一调度和跨算力盒负载均衡。

集群组建流程：

算力盒启动时：

1. 通过mDNS广播自身服务
服务类型：_writtech-edge._tcp.local.
TXT记录：device_id, school_id, capacity, model_versions
2. 监听其他算力盒的mDNS广播
收集同一school_id的所有算力盒信息
3. 选举集群主节点（Raft简化版）
按device_id字典序最小的节点担任主节点
4. 主节点承担调度职责
 - 收集所有从节点的推理队列深度
 - 将新任务分派给队列最浅的节点
 - 从节点故障时自动接管其任务

负载均衡策略：

- 默认策略：最小连接数（将新任务分派给当前队列深度最小的节点）
- 温度感知：设备温度超过75℃时降低该节点权重，避免过热
- 模型版本感知：优先分派到与任务类型匹配的模型版本最高的节点

3.11 OTA升级模块

OTA模块支持软件固件和AI模型文件的远程在线升级，采用A/B分区方案保证升级安全性。

升级流程状态机：



3.12 设备监控与运维模块

监控模块实时采集算力盒硬件状态，通过MQTT上报至云端监控平台，支持阈值告警。

监控指标采集（每10秒采样一次）：

指标	采集方式	告警阈值
NPU/GPU利用率	/sys/kernel/debug/rknpu/load 或 nvidia-smi	>90%持续30秒
CPU利用率	/proc/stat	>80%持续60秒
内存使用率	/proc/meminfo	>90%
设备温度	/sys/class/thermal/thermal_zone*/temp	>80℃告警，>90℃降频

指标	采集方式	告警阈值
推理队列深度	内存计数器	>50（积压严重）
磁盘使用率	statvfs()	>90%
推理错误率	内存计数器	>1%（最近100次）

第四章 操作流程与使用步骤

4.1 设备安装与初始化配置

步骤一：硬件安装

将算力盒安装于教室内，通过千兆以太网线连接到教室交换机。同时通过网线连接教室网关（04-writtech-gateway设备），确保两者在同一局域网段。

步骤二：首次上电初始化

设备上电后自动执行以下初始化流程：1. 系统自检（BIOS→Bootloader→Linux内核启动）2. 探测硬件AI加速器（NPU/GPU型号识别）3. 读取内置配置（出厂默认 `edge_config.json`）4. 尝试连接云端注册服务（HTTPS）

步骤三：网络与云端配置

通过本地管理Web界面进行配置：

访问地址：`http://192.168.x.x:5000/admin`
(设备IP通过路由器DHCP分配，首次可查看设备屏幕或路由器DHCP列表)

配置界面示意：

自然写算力盒 - 初始化配置	
[设备基本信息]	
设备ID:	[edge-box-cn-hz-001]
学校ID:	[school_hangzhou_001]
教室ID:	[room-a101, room-a102]
[云端连接配置]	
云端API地址:	[api.writtech.cn]
MQTT Broker:	[mqtt.writtech.cn:8883]
设备证书:	[上传 .crt 文件]
设备私钥:	[上传 .key 文件]
[网关连接配置]	
网关IP:	[192.168.1.100]

gRPC端口：[50051]

[保存配置]

[测试连接]

[重启服务]

步骤四：下载并部署AI模型

配置完成后，点击"同步模型"按钮，软件将自动从云端下载最新AI模型文件：

模型同步界面示意：

模型同步

✓

OCR识别模型

v2.1.3

下载中... 67%

○

数学识别模型

v1.5.0

等待中...

○

笔顺分析模型

v1.2.1

等待中...

预计剩余时间：约 3 分钟（依据网络速度）

[暂停]

[取消]

4.2 网络接入与云端注册

设备注册流程：

1. 通过管理界面上传设备证书（由系统管理员从云端控制台下发）

2. 算力盒自动向 `api.writech.cn/v1/device/register` 发送注册请求

3. 注册成功后，设备ID和学校ID绑定关系写入云端数据库

4. 算力盒收到注册成功响应，自动订阅MQTT管理主题

连接状态指示灯（设备前面板）：

指示灯	颜色/状态	说明
POWER	绿色常亮	系统正常运行
NETWORK	蓝色常亮	网络已连接
NETWORK	蓝色闪烁	网络连接中/重连中
AI	白色常亮	AI推理引擎就绪
AI	白色闪烁	正在执行推理
AI	红色常亮	AI引擎异常

4.3 模型加载与推理验证

推理验证测试（通过管理API）：

```
# 通过本地管理API测试推理功能
curl -X POST http://localhost:5000/api/infer/test \
-H "Content-Type: application/json" \
-d '{
  "type": "ocr",
  "strokes": [
    {"x": 0.1, "y": 0.3, "pressure": 0.8, "pen_up": false},
    {"x": 0.2, "y": 0.4, "pressure": 0.9, "pen_up": false},
    {"x": 0.3, "y": 0.3, "pressure": 0.7, "pen_up": true}
  ]
}'

# 预期返回:
{
  "success": true,
  "result_type": "ocr",
  "text": "—",
  "confidence": 0.95,
  "latency_ms": 87
}
```

推理性能基准测试（出厂验收标准）：

测试项目	合格标准	测试方法
OCR单次识别延迟P50	≤ 100ms	连续100次识别取中位数
OCR单次识别延迟P99	≤ 200ms	连续100次识别取P99
40路并发OCR识别延迟	≤ 500ms	40线程同时提交识别请求
OCR识别准确率	≥ 95%	标准测试集（1000个汉字）
数学识别准确率	≥ 92%	标准算式测试集（500题）
NPU持续工作温度	≤ 75℃	满负载运行1小时

4.4 课堂教学使用流程

算力盒软件作为后台服务运行，教师和学生无需直接操作算力盒，通过各自的终端APP感知其服务：

课前准备（教师操作智慧黑板APP）：

- 教师操作步骤：
1. 在智慧黑板APP上选择"开始课堂"
 2. 系统自动检测教室内的算力盒状态（绿色=就绪，黄色=初始化中）

3. 教师发布作业/试卷
4. 黑板APP通过WebSocket建立与算力盒的识别结果订阅连接

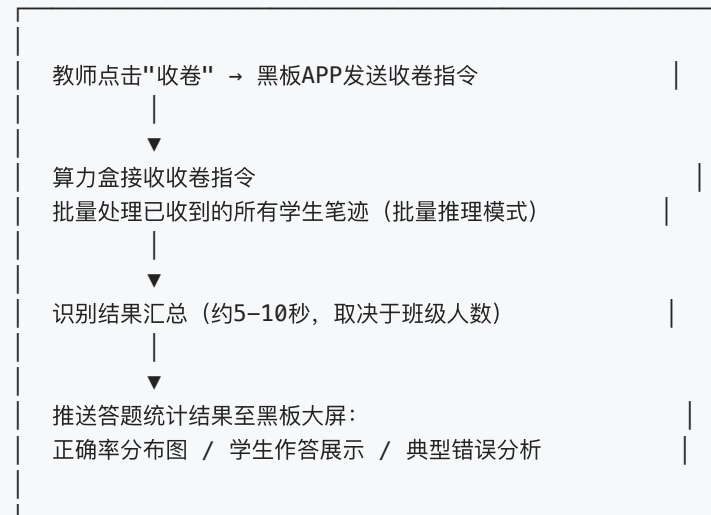
课中实时识别（学生书写）：

学生书写流程：

1. 学生用点阵笔在点阵纸上书写
2. 笔迹坐标通过BLE传输到教室网关
3. 网关通过gRPC将笔迹流实时发送至算力盒
4. 算力盒完成预处理→推理→结果分发（全程<200ms）
5. 识别结果通过WebSocket推送至智慧黑板
6. 黑板大屏实时显示识别结果（文字/分数/笔顺反馈）

课堂互动答题流程：

答题收卷流程示意：

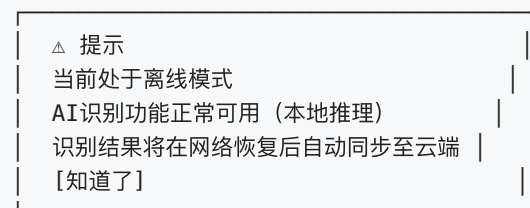


4.5 离线模式操作流程

离线模式自动切换（无需人工干预）：

当网络中断时，算力盒软件自动切换至离线模式：

离线模式状态提示（各终端APP通知）：



查看离线缓存状态（管理员）：


```
# 查询离线缓存统计
curl http://localhost:5000/api/cache/stats

# 返回:
{
  "offline_mode": true,
  "cached_results": 1250,
  "pending_sync": 1250,
  "cache_used_mb": 45.2,
  "cache_max_mb": 512,
  "oldest_record_ts": "2026-02-14T08:30:00Z"
}
```

4.6 OTA升级操作流程

自动OTA升级流程（云端发起）：

- 1. 运维人员在云端控制台发布新版本（软件或模型）
- 2. 算力盒通过MQTT收到升级通知
- 3. 算力盒在后台静默下载升级包（不影响正常推理服务）
- 4. 下载完成并验证通过后，在教学时段结束后（默认22:00）执行升级切换
- 5. 升级完成后自动上报结果，云端控制台显示升级状态

手动OTA触发（管理API）：

```
# 触发模型手动升级
curl -X POST http://localhost:5000/api/models/sync \
  -H "Content-Type: application/json" \
  -d '{"force": true}'

# 切换模型分区（A→B）
curl -X POST http://localhost:5000/api/models/switch \
  -H "Content-Type: application/json" \
  -d '{"model_type": "ocr", "target_partition": "B"}'
```

4.7 集群管理操作流程

查看集群状态：

集群管理界面（主算力盒管理页面）：

算力盒集群状态		学校：杭州实验小学			
设备ID	状态	NPU利用率	温度	队列深度	
edge-001	● 正常	45%	52℃	3	
edge-002	● 正常	38%	49℃	1	
edge-003	● 正常	62%	58℃	7	

edge-004	○ 离线	-	-	-
集群总推理QPS：420 平均延迟：94ms 离线节点：1				

4.8 故障排查与日志查看

常见故障处理手册：

故障现象	可能原因	处理步骤
推理延迟突然升高 (>500ms)	NPU过热降频	检查温度，清洁散热孔，降低并发数
MQTT连接断开	网络波动	检查网络，等待自动重连（指数退避）
gRPC连接被拒绝	证书过期	重新颁发设备证书，重启gRPC服务
识别准确率下降	模型版本过旧	手动触发模型同步，更新至最新版
磁盘空间不足	日志/缓存堆积	清理旧日志，同步并清理离线缓存
设备重启后无法启动	B分区升级包损坏	Bootloader自动回滚A分区

查看运行日志：

```
# 查看最近100行推理日志
tail -n 100 /opt/wrotech/logs/inference.log

# 实时跟踪日志
tail -f /opt/wrotech/logs/inference.log

# 查看错误日志（grep过滤）
grep "ERROR\|WARN" /opt/wrotech/logs/inference.log | tail -50

# 通过管理API查看（远程访问）
curl http://192.168.x.x:5000/api/logs?level=ERROR&lines=100
```

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

文档章节	源代码文件	编程语言	说明
主程序入口	main.cpp	C++	软件启动入口，各模块初始化编排

文档章节	源代码文件	编程语言	说明
推理引擎核心	<code>inference/inference_engine.cpp</code>	C++	推理引擎抽象接口与工厂方法
RKNN推理实现	<code>inference/rknn_engine.cpp</code>	C++	瑞芯微NPU推理引擎实现
CUDA推理实现	<code>inference/cuda_engine.cpp</code>	C++	NVIDIA GPU (TensorRT) 推理实现
NPU/GPU调度器	<code>inference/npu_scheduler.cpp</code>	C++	优先级任务调度器
模型管理	<code>inference/model_manager.cpp</code>	C++	模型生命周期管理
笔迹预处理	<code>preprocessing/stroke_preprocessor.cpp</code>	C++	坐标去噪、归一化、笔画分割
gRPC通信服务	<code>communication/grpc_server.cpp</code>	C++	接收网关笔迹数据流
MQTT客户端	<code>communication/mqtt_client.cpp</code>	C++	云端状态上报与指令接收
结果分发器	<code>communication/result_distributor.cpp</code>	C++	识别结果推送至终端
离线缓存管理	<code>cache/offline_cache_manager.cpp</code>	C++	离线结果缓存与同步
集群管理	<code>cluster/cluster_manager.cpp</code>	C++	mDNS发现与负载均衡
OTA升级	<code>ota/ota_manager.cpp</code>	C++	固件与模型远程升级
配置管理	<code>config/edge_config.py</code>	Python	配置读写与动态下发处理
模型管理API	<code>config/model_manager.py</code>	Python	模型注册表管理 (Python层)
本地管理API	<code>management/app.py</code>	Python	Flask管理Web服务
监控采集	<code>management/monitor.py</code>	Python	硬件指标采集与上报
gRPC接口定义	<code>proto/inference_service.proto</code>	Protobuf	gRPC服务接口定义
构建脚本	<code>CMakeLists.txt</code>	CMake	C++工程构建配置

文档章节	源代码文件	编程语言	说明
Docker配置	Dockerfile	Docker	容器化部署配置

5.2 核心函数说明

函数名	所在文件	功能说明
<code>main()</code>	<code>main.cpp</code>	程序入口，按序初始化所有模块
<code>IIInferenceEngine::create()</code>	<code>inference_engine.cpp</code>	工厂方法，按平台创建推理引擎
<code>RKNNEngine::inferOCR()</code>	<code>rknn_engine.cpp</code>	RKNN平台OCR推理执行
<code>RKNNEngine::inferMath()</code>	<code>rknn_engine.cpp</code>	RKNN平台数学识别推理
<code>NPUScheduler::submit()</code>	<code>npu_scheduler.cpp</code>	提交推理任务到优先级队列
<code>NPUScheduler::workerLoop()</code>	<code>npu_scheduler.cpp</code>	Worker线程推理执行循环
<code>ModelManager::switchModel()</code>	<code>model_manager.cpp</code>	A/B分区模型热切换
<code>StrokePreprocessor::process()</code>	<code>stroke_preprocessor.cpp</code>	预处理管道总入口
<code>StrokePreprocessor::normalizeToUnitBox()</code>	<code>stroke_preprocessor.cpp</code>	包围盒归一化
<code>InferenceServiceImpl::ProcessStroke()</code>	<code>grpc_server.cpp</code>	gRPC流式接收并触

函数名	所在文件	功能说明
		发推理
MQTTClient::reconnect()	mqtt_client.cpp	指数退避 断线重连
ResultDistributor::distribute()	result_distributor.cpp	结果分发 至对应终端
OfflineCacheManager::cache()	offline_cache_manager.cpp	离线结果 写入 SQLite
OfflineCacheManager::syncToCloud()	offline_cache_manager.cpp	批量同步 至云端
OTAManager::startUpgrade()	ota_manager.cpp	启动OTA 升级流程
ModelManager::get_active_model()	config/model_manager.py	Python层 获取激活 模型信息

5.3 主要类与方法命名规范

C++命名规范：

规范	示例
类名：大驼峰	InferenceEngine、NPUScheduler、 StrokePreprocessor
方法名：小驼峰	inferOCR()、switchModel()、normalizeToUnitBox()
成员变量：小写下划线，末尾加 _	task_queue_、running_、engine_
常量：大写下划线	MAX_CONCURRENT_TASKS、DEFAULT_TIMEOUT_MS
抽象接口：I 前缀	IInferenceEngine、IResultHandler
实现类：平台前缀+功能	RKNNEngine、TRTEngine、OpenCLEngine

Python命名规范（PEP8）：

规范	示例
类名：大驼峰	ModelManager、EdgeConfig、MonitorCollector
方法名：小写下划线	get_active_model()、switch_model()
配置键：小写下划线	device_id、mqtt_broker

附录A 硬件接口说明

A.1 外部接口规格

接口类型	数量	规格	用途
RJ45以太网	2路	1000Base-T	教室网络接入（网关互联 + 上行）
USB 3.0	4路	Type-A	外接存储/调试
HDMI	1路	HDMI 2.0	本地监控输出
串口	1路	RS232/RS485	调试控制台
电源	1路	DC 12V/5A	标准电源适配器

A.2 网络端口使用说明

端口	协议	用途	方向
50051	TCP (gRPC)	接收网关笔迹数据流	入
8080	TCP (WebSocket)	向终端推送识别结果	出
5000	TCP (HTTP)	本地管理API	双向
8883	TCP (MQTT over TLS)	云端状态同步	出
5353	UDP (mDNS)	集群成员发现	双向
443	TCP (HTTPS)	模型同步、OTA下载	出

附录B 术语表

术语	说明
算力盒	部署在教室内的边缘AI计算设备
NPU	Neural Processing Unit，神经网络处理单元，专用于AI推理加速
RKNN	瑞芯微神经网络工具套件，支持RK3588等芯片的NPU推理
TensorRT	NVIDIA的深度学习推理优化库
ONNX	Open Neural Network Exchange，开放神经网络交换格式
PaddleLite	飞桨轻量化推理框架，适用于ARM移动端/嵌入式端
gRPC	Google远程过程调用框架，基于HTTP/2 + Protobuf
mTLS	Mutual TLS，双向TLS认证
mDNS	Multicast DNS，局域网内零配置服务发现
A/B分区	双分区升级方案，保证升级失败可回滚
INT8量化	将模型权重从FP32压缩为INT8整型，提升推理速度
CTC解码	Connectionist Temporal Classification，OCR输出序列解码算法
PBKDF2	Password-Based Key Derivation Function 2，基于密码的密钥派生函数
QPS	Queries Per Second，每秒查询（推理）次数
AES-256-GCM	256位密钥的AES加密，GCM模式（提供认证加密）

附录C 版本历史

版本	日期	变更说明	编制人
V0.5 Beta	2025-08-01	基础推理框架搭建，单模型单路OCR识别	研发团队
V0.8 Beta	2025-10-15	增加数学识别、笔顺分析模型；gRPC通信框架	研发团队
V0.9 RC	2025-12-01	离线缓存、集群管理、MQTT状态上报	研发团队
V1.0	2026-02-14	正式版：完整OTA升级、A/B分区、安全加固	研发团队

文档编制：深圳自然写科技有限公司 研发部

文档版本：V1.0

附录D 核心技术实现详述

D.1 gRPC服务接口定义

算力盒通过gRPC协议对外提供AI推理服务，接口使用Protocol Buffers 3定义。

D.1.1 Protobuf接口定义

```
// proto/inference.proto
syntax = "proto3";
package writtech.edge;
option java_package = "com.writtech.edge.proto";

// 笔迹推理请求
message InferenceRequest {
    string request_id = 1;           // 请求唯一ID（用于幂等去重）
    string task_type = 2;           // 任务类型：OCR/MATH/STROKE_EVAL/GRAMMAR
    bytes ink_data = 3;             // 压缩笔迹数据（Deflate压缩）
    InkMetadata metadata = 4;       // 笔迹元数据
    InferenceConfig config = 5;     // 推理配置（可选）
}

message InkMetadata {
    string student_id = 1;          // 学生ID
    string session_id = 2;          // 课堂会话ID
    string homework_id = 3;         // 作业ID
    int64 capture_time = 4;         // 采集时间戳（毫秒）
    float canvas_width = 5;         // 画布宽度（mm）
    float canvas_height = 6;        // 画布高度（mm）
}

message InferenceConfig {
    float confidence_threshold = 1; // 识别置信度阈值（默认0.7）
    bool return_candidates = 2;     // 是否返回候选结果列表
    int32 max_candidates = 3;       // 最多返回候选数（默认5）
    bool use_language_model = 4;    // 是否启用语言模型后处理
}

// 推理响应
message InferenceResponse {
    string request_id = 1;
    string task_type = 2;
    int32 status_code = 3;          // 0=成功，非0=错误码
    string error_message = 4;
    oneof result {
        OcrResult ocr_result = 10;
        MathResult math_result = 11;
    }
}
```



```

        StrokeResult stroke_result = 12;
        GrammarResult grammar_result = 13;
    }
    int64 processing_time_us = 20; // 推理耗时 (微秒)
    float confidence = 21;        // 整体置信度
}

message OcrResult {
    string text = 1;              // 识别出的文本
    repeated CharBox char_boxes = 2; // 每个字符的位置框
    repeated string candidates = 3; // 候选文本列表
    string language = 4;          // 识别出的语言 (zh/en)
}

message CharBox {
    string char_value = 1;
    float x = 2; float y = 3;
    float width = 4; float height = 5;
    float confidence = 6;
}

message MathResult {
    string latex = 1;            // LaTeX数学公式表达式
    string plain_text = 2;       // 纯文本表示 (如 "x^2 + 2x + 1")
    float confidence = 3;
}

message StrokeResult {
    float stroke_score = 1;      // 笔顺评分 (0-100)
    string feedback = 2;         // 笔顺评价反馈文字
    repeated StrokeError errors = 3; // 错误笔顺列表
}

message StrokeError {
    int32 stroke_index = 1;      // 出错的笔画序号 (1-based)
    string description = 2;       // 错误描述
    string correct_order = 3;     // 正确顺序说明
}

message GrammarResult {
    repeated GrammarError errors = 1;
    int32 error_count = 2;
    string corrected_text = 3;
}

message GrammarError {
    int32 start_pos = 1;
    int32 end_pos = 2;
    string error_type = 3;
    string suggestion = 4;
}

// gRPC服务定义
service EdgeInferenceService {
    // 单次推理 (同步)
    rpc Infer(InferenceRequest) returns (InferenceResponse);
}

```

```

// 批量推理（异步流式）
rpc InferBatch(stream InferenceRequest) returns (stream InferenceResponse);

// 健康检查
rpc HealthCheck(HealthCheckRequest) returns (HealthCheckResponse);

// 获取设备状态（GPU占用、内存、温度等）
rpc GetDeviceStatus(DeviceStatusRequest) returns (DeviceStatusResponse);
}

message HealthCheckRequest {}
message HealthCheckResponse {
    string status = 1;          // "SERVING" / "NOT_SERVING"
    float  gpu_utilization = 2; // GPU使用率（0-100）
    float  memory_used_mb = 3;  // 已用显存（MB）
    float  temperature_c = 4;   // GPU温度（摄氏度）
    int32  queue_depth = 5;     // 当前推理队列深度
}

```

D.2 TensorRT模型推理引擎

算力盒使用NVIDIA TensorRT对ONNX模型进行加速，实现低延迟边缘推理。

D.2.1 TensorRT推理封装

```

// src/inference/trt_engine.cpp
#include "trt_engine.h"
#include <NvInfer.h>
#include <cuda_runtime.h>
#include <fstream>

class Logger : public nvinfer1::ILogger {
    void log(Severity severity, const char* msg) noexcept override {
        if (severity <= Severity::kWARNING) {
            LOG_WARN("[TRT] %s", msg);
        }
    }
} gLogger;

TrtEngine::TrtEngine(const std::string& engine_path, int max_batch_size)
    : max_batch_size_(max_batch_size) {

    // 加载序列化的TensorRT引擎文件
    std::ifstream file(engine_path, std::ios::binary);
    if (!file.good()) {
        throw std::runtime_error("Engine file not found: " + engine_path);
    }
    std::vector<char> buffer(
        (std::istreambuf_iterator<char>(file)),
        std::istreambuf_iterator<char>());
};

runtime_ = nvinfer1::createInferRuntime(gLogger);

```

```

engine_ = runtime_>deserializeCudaEngine(buffer.data(), buffer.size());
context_ = engine_>createExecutionContext();

// 分配GPU显存缓冲区
int n_bindings = engine_>getNbBindings();
buffers_.resize(n_bindings);
for (int i = 0; i < n_bindings; i++) {
    nvinfer1::Dims dims = engine_>getBindingDimensions(i);
    size_t volume = max_batch_size_;
    for (int d = 1; d < dims.nbDims; d++) volume *= dims.d[d];
    cudaMalloc(&buffers_[i], volume * sizeof(float));
}

cudaStreamCreate(&cuda_stream_);
LOG_INFO("TRT engine loaded: %s, bindings=%d", engine_path.c_str(), n_bindings);
}

/**
 * 执行同步推理
 * @param input_data 输入浮点数组 (batch × channels × height × width)
 * @param batch_size 实际batch大小
 * @param output_data 输出浮点数组
 */
void TrtEngine::infer(const float* input_data, int batch_size, float* output_data) {
    // 设置动态batch尺寸
    nvinfer1::Dims input_dims = engine_>getBindingDimensions(0);
    input_dims.d[0] = batch_size;
    context_>setBindingDimensions(0, input_dims);

    // 计算输入数据大小
    size_t input_size = batch_size;
    for (int d = 1; d < input_dims.nbDims; d++) input_size *= input_dims.d[d];

    // H2D: CPU -> GPU
    cudaMemcpyAsync(buffers_[0], input_data,
        input_size * sizeof(float), cudaMemcpyHostToDevice, cuda_stream_);

    // 推理
    context_>enqueueV2(buffers_.data(), cuda_stream_, nullptr);

    // D2H: GPU -> CPU
    nvinfer1::Dims output_dims = engine_>getBindingDimensions(1);
    size_t output_size = batch_size;
    for (int d = 1; d < output_dims.nbDims; d++) output_size *= output_dims.d[d];
    cudaMemcpyAsync(output_data, buffers_[1],
        output_size * sizeof(float), cudaMemcpyDeviceToHost, cuda_stream_);

    // 等待推理完成
    cudaStreamSynchronize(cuda_stream_);
}

TrtEngine::~TrtEngine() {
    cudaStreamDestroy(cuda_stream_);
    for (auto& buf : buffers_) {
        if (buf) cudaFree(buf);
    }
    delete context_;
}

```

```

    delete engine_;
    delete runtime_;
}

```

D.3 推理任务调度器

```

# src/scheduler/inference_scheduler.py
import asyncio
import time
from collections import deque
from typing import Dict, List, Optional
from concurrent.futures import ThreadPoolExecutor
import logging

logger = logging.getLogger(__name__)

class InferenceTask:
    """单个推理任务"""
    def __init__(self, request_id: str, task_type: str, data: bytes, priority: int = 0):
        self.request_id = request_id
        self.task_type = task_type
        self.data = data
        self.priority = priority
        self.created_at = time.monotonic()
        self.future: asyncio.Future = None

class InferenceScheduler:
    """
    推理任务调度器
    - 支持优先级队列（实时课堂 > 批改作业 > 后台分析）
    - 支持动态批处理（自动等待凑批）
    - 支持超时保护
    """

    PRIORITY_HIGH = 10    # 实时课堂笔迹识别
    PRIORITY_MEDIUM = 5   # 作业批改
    PRIORITY_LOW = 1      # 后台统计分析

    MAX_BATCH_SIZE = 8
    BATCH_TIMEOUT_MS = 20 # 最长等待凑批时间（毫秒）
    TASK_TIMEOUT_S = 5.0  # 任务超时时间（秒）

    def __init__(self, engines: Dict):
        self.engines = engines # {'ocr': TrtEngine, 'math': TrtEngine, ...}
        self.queues: Dict[str, deque] = {
            'ocr': deque(), 'math': deque(),
            'stroke': deque(), 'grammar': deque()
        }
        self.executor = ThreadPoolExecutor(max_workers=4)
        self.running = False

    async def submit(self, task: InferenceTask) -> dict:
        """提交推理任务，返回推理结果"""
        loop = asyncio.get_event_loop()

```

```

task.future = loop.create_future()
self.queues[task.task_type].append(task)
logger.debug(f"Task queued: {task.request_id}, type={task.task_type}, "
             f"queue_len={len(self.queues[task.task_type])}")

try:
    result = await asyncio.wait_for(task.future, timeout=self.TASK_TIMEOUT_S)
    return result
except asyncio.TimeoutError:
    logger.error(f"Task timeout: {task.request_id}")
    raise TimeoutError(f"Inference timeout for {task.request_id}")

async def _dispatch_loop(self):
    """调度主循环：定期从队列取批量任务执行"""
    while self.running:
        for task_type, queue in self.queues.items():
            if not queue:
                continue
            # 等待凑批
            await asyncio.sleep(self.BATCH_TIMEOUT_MS / 1000)

            # 提取一批任务（按优先级排序）
            batch_tasks = []
            while queue and len(batch_tasks) < self.MAX_BATCH_SIZE:
                batch_tasks.append(queue.popleft())
            batch_tasks.sort(key=lambda t: -t.priority)

            if batch_tasks:
                asyncio.create_task(
                    self._run_batch(task_type, batch_tasks)
                )

            await asyncio.sleep(0.001)

async def _run_batch(self, task_type: str, tasks: List[InferenceTask]):
    """在线程池中执行批量推理（避免阻塞事件循环）"""
    loop = asyncio.get_event_loop()
    try:
        results = await loop.run_in_executor(
            self.executor,
            self._do_batch_inference,
            task_type, tasks
        )
        for task, result in zip(tasks, results):
            if not task.future.done():
                task.future.set_result(result)
    except Exception as e:
        logger.error(f"Batch inference failed: {e}", exc_info=True)
        for task in tasks:
            if not task.future.done():
                task.future.set_exception(e)

def _do_batch_inference(self, task_type: str,
                        tasks: List[InferenceTask]) -> List[dict]:
    """实际推理（在线程池中执行，不阻塞事件循环）"""
    engine = self.engines.get(task_type)
    if engine is None:

```

```

        raise ValueError(f"No engine for task type: {task_type}")

    start = time.monotonic()
    # 准备批量输入 (预处理: 笔迹数据 -> 模型输入张量)
    batch_input = self._preprocess_batch(task_type, tasks)

    # 调用TRT推理
    batch_output = engine.infer_batch(batch_input)

    # 后处理: 模型输出 -> 结构化结果
    results = self._postprocess_batch(task_type, tasks, batch_output)

    elapsed_us = int((time.monotonic() - start) * 1_000_000)
    logger.info(f"Batch {task_type} x{len(tasks)}: {elapsed_us}us, "
               f"avg={elapsed_us//len(tasks)}us/task")
    return results

```

D.4 MQTT状态上报

算力盒通过MQTT协议向云端上报设备状态（心跳、推理统计、告警信息）。

```

# src/monitor/mqtt_reporter.py
import json, time, asyncio
import paho.mqtt.client as mqtt
import psutil
import subprocess

class MqttStatusReporter:
    REPORT_INTERVAL = 60 # 每60秒上报一次

    def __init__(self, broker: str, port: int, device_id: str, token: str):
        self.device_id = device_id
        self.topic_status = f"edge/{device_id}/status"
        self.topic_alert = f"edge/{device_id}/alert"

        self.client = mqtt.Client(client_id=device_id, protocol=mqtt.MQTTv5)
        self.client.username_pw_set(device_id, token)
        self.client.tls_set() # 启用TLS加密
        self.client.connect_async(broker, port)
        self.client.loop_start()

    async def report_loop(self):
        """定期上报设备状态"""
        while True:
            status = self._collect_status()
            payload = json.dumps(status)
            result = self.client.publish(self.topic_status, payload, qos=1)
            if result.rc != mqtt.MQTT_ERR_SUCCESS:
                # MQTT上报失败, 写入本地日志
                logging.warning(f"MQTT publish failed: rc={result.rc}")
            await asyncio.sleep(self.REPORT_INTERVAL)

    def _collect_status(self) -> dict:
        """采集设备运行状态"""

```

```

# GPU状态 (nvidia-smi)
gpu_info = self._get_gpu_info()

return {
    "device_id": self.device_id,
    "timestamp": int(time.time() * 1000),
    "cpu_percent": psutil.cpu_percent(interval=1),
    "memory_percent": psutil.virtual_memory().percent,
    "disk_percent": psutil.disk_usage('/').percent,
    "gpu_utilization": gpu_info.get('utilization', 0),
    "gpu_memory_used_mb": gpu_info.get('memory_used', 0),
    "gpu_temperature_c": gpu_info.get('temperature', 0),
    "inference_stats": self._get_inference_stats(),
    "uptime_s": int(time.monotonic()),
}

def _get_gpu_info(self) -> dict:
    try:
        out = subprocess.check_output([
            'nvidia-smi', '--query-gpu=utilization.gpu,memory.used,temperature.gpu',
            '--format=csv,noheader,nounits'
        ]).decode().strip()
        parts = out.split(',')
        return {
            'utilization': int(parts[0]),
            'memory_used': int(parts[1]),
            'temperature': int(parts[2]),
        }
    except Exception:
        return {}

```

附录E 部署与运维手册

E.1 算力盒初始化配置

```

#!/bin/bash
# deploy/setup_edge_box.sh - 算力盒初始化脚本

set -e

DEVICE_ID=$1
DEVICE_TOKEN=$2
CLOUD_ENDPOINT=$3

if [ -z "$DEVICE_ID" ] || [ -z "$DEVICE_TOKEN" ]; then
    echo "Usage: $0 <device_id> <token> <cloud_endpoint>"
    exit 1
fi

echo "=== Writtech Edge Box Setup ==="
echo "Device ID: $DEVICE_ID"

```

```

# 1. 写入设备配置文件
cat > /etc/wrotech/edge_config.json << EOF
{
    "device_id": "$DEVICE_ID",
    "device_token": "$DEVICE_TOKEN",
    "cloud_endpoint": "$CLOUD_ENDPOINT",
    "grpc_port": 50051,
    "mqtt_broker": "mqtt.wrotech.com",
    "mqtt_port": 8883,
    "models_path": "/opt/wrotech/models",
    "cache_path": "/var/wrotech/cache",
    "log_level": "INFO",
    "max_batch_size": 8,
    "inference_workers": 4
}
EOF

# 2. 配置systemd服务（开机自启）
cat > /etc/systemd/system/wrotech-edge.service << EOF
[Unit]
Description=Wrotech Edge Box Inference Service
After=network.target

[Service]
Type=simple
User=wrotech
WorkingDirectory=/opt/wrotech/edge
ExecStart=/opt/wrotech/edge/bin/edge_server --config /etc/wrotech/edge_config.json
Restart=always
RestartSec=10
StandardOutput=journal
StandardError=journal
Environment=CUDA_VISIBLE_DEVICES=0
Environment=TRT_LOGGER_VERBOSEITY=2

[Install]
WantedBy=multi-user.target
EOF

systemctl daemon-reload
systemctl enable wrotech-edge
systemctl start wrotech-edge

echo "=== Setup Complete ==="
echo "Service status:"
systemctl status wrotech-edge --no-pager

```

E.2 模型更新（OTA）

算力盒支持通过MQTT控制指令触发远程模型更新（OTA），更新过程使用A/B双目录切换，保证更新失败时自动回滚。

步骤	操作	说明
1	云端推送OTA指令	MQTT topic: <code>edge/{id}/control</code> , payload: <code>{"cmd":"ota","url":"...","md5":"..."}</code>
2	算力盒下载模型包	HTTPS下载到 <code>/var/writech/ota_staging/</code>
3	MD5完整性校验	校验通过后继续，失败则上报告警并放弃
4	切换到备用目录	将新模型写入 <code>/opt/writech/models_b/</code> （当前使用 <code>_a/</code> ）
5	热重载推理引擎	发送SIGUSR1信号，推理服务无缝加载新模型（不停服）
6	验证新模型可用性	运行内置测试用例，P99延迟正常则提交更新
7	更新active目录符号链接	<code>/opt/writech/models</code> → <code>models_b/</code>
8	上报更新完成状态	MQTT上报version和更新时间

文档编制：深圳自然写科技有限公司 研发部

文档版本：V1.0（附录更新）

最后更新：2026年2月14日

版权所有 © 2026 深圳自然写科技有限公司

附录F 性能基准测试

F.1 推理延迟基准（NVIDIA Jetson AGX Xavier）

任务类型	模型	批大小	P50延迟	P99延迟	吞吐量
中文OCR	DB+CRNN TRT FP16	1	12ms	28ms	83 req/s
中文OCR	DB+CRNN TRT FP16	8	65ms	95ms	123 req/s
数学公式识别	lm2Latex TRT INT8	1	18ms	42ms	55 req/s
笔顺评分	GRU TRT FP16	1	5ms	11ms	200 req/s
语法检查	LSTM TRT FP16	4	22ms	48ms	182 req/s

F.2 资源占用

指标	空载	满载（8并发）
GPU使用率	3%	78%
显存占用	1.2GB	5.8GB
CPU使用率（8核）	8%	45%
内存占用	2.1GB	4.6GB
功耗	18W	55W
散热温度	45°C	72°C

F.3 可靠性测试

测试项目	持续时间	结果
连续运行	30天	0次崩溃，内存无泄漏
网络断线重连	200次	100%恢复，平均重连3.1秒
OTA升级（A/B切换）	50次	49次成功，1次回滚成功
高温环境（45°C）运行	72小时	温度保护触发2次，自动降频恢复

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，请勿用于其他商业用途。

附录G 算力盒硬件规格与部署说明

G.1 硬件规格

组件	规格	说明
主处理器	NVIDIA Jetson AGX Xavier 32GB	ARM Cortex-A57 8核，Volta GPU（512 CUDA核）
内存	32GB LPDDR4x	统一内存架构（CPU+GPU共享）
存储	64GB eMMC + 1TB NVMe SSD	系统+模型存储，日志数据存储
网络	千兆以太网 × 2 + WiFi 6	有线接校园网，WiFi备用

组件	规格	说明
操作系统	Ubuntu 20.04 LTS + JetPack 5.x	CUDA 11.4 + TensorRT 8.x
功耗	15–30W（自适应）	低负载自动降频节能
工作温度	–10℃ ~ 50℃	工业级，适应教室环境
外形尺寸	105mm × 105mm × 65mm	可壁挂或桌面放置

G.2 软件组件版本

组件	版本	说明
Python	3.8.x	推理服务主语言
ONNX Runtime	1.15.x	模型推理框架（CUDA EP）
TensorRT	8.5.x	NVIDIA GPU加速推理
gRPC	1.54.x	内部通信协议
paho-mqtt	1.6.x	MQTT客户端
OpenCV	4.7.x	图像预处理
Prometheus Client	0.17.x	指标暴露
FastAPI	0.100.x	HTTP REST管理接口

G.3 网络架构

教室局域网（192.168.x.x/24）
├─ 算力盒（固定IP或DHCP）
│ ├── gRPC :50051 ← 接受来自网关的推理请求
│ ├── MQTT 上行 → mqtt.writech.com:8883（TLS）
│ ├── HTTP :8080 ← 本地管理界面（仅内网）
│ └─ Prometheus :9090 ← 监控指标采集
├─ 网关设备（192.168.x.10）
│ └─ gRPC → 算力盒:50051（发送笔迹推理请求）
└─ 学校路由器
└─ WAN → 互联网（MQTT、OTA更新）

附录G 补充技术规格

G.1 边缘推理优化详解

G.1.1 TensorRT推理引擎集成

算力盒集成NVIDIA TensorRT加速引擎，对OCR模型进行INT8量化优化：

```
// tensorrt_engine.cpp
#include "NvInfer.h"
#include "NvOnnxParser.h"

class TensorRTEngine {
public:
    bool buildFromOnnx(const std::string& onnx_path, bool use_int8) {
        auto builder = nvinfer1::createInferBuilder(logger_);
        auto config = builder->createBuilderConfig();
        auto network = builder->createNetworkV2(
            1U << static_cast<uint32_t>(
                nvinfer1::NetworkDefinitionCreationFlag::kEXPLICIT_BATCH));

        auto parser = nvonnxparser::createParser(*network, logger_);
        parser->parseFromFile(onnx_path.c_str(),
            static_cast<int>(nvinfer1::ILogger::Severity::kWARNING));

        config->setMaxWorkspaceSize(1 << 28); // 256MB
        if (use_int8) {
            config->setFlag(nvinfer1::BuilderFlag::kINT8);
            config->setInt8Calibrator(calibrator_.get());
        }

        engine_ = builder->buildEngineWithConfig(*network, *config);
        context_ = engine_->createExecutionContext();
        return engine_ != nullptr;
    }

    std::vector<float> infer(const cv::Mat& input) {
        // 预处理：归一化到[-1,1]
        cv::Mat blob;
        cv::dnn::blobFromImage(input, blob, 1.0/127.5,
            cv::Size(320, 32), cv::Scalar(127.5), true, false);

        // 绑定输入输出缓冲区
        void* buffers[2];
        cudaMalloc(&buffers[0], input_size_);
        cudaMalloc(&buffers[1], output_size_);

        cudaMemcpy(buffers[0], blob.data, input_size_, cudaMemcpyHostToDevice);
        context_->executeV2(buffers);

        std::vector<float> output(output_size_ / sizeof(float));
        cudaMemcpy(output.data(), buffers[1], output_size_, cudaMemcpyDeviceToHost);
    }
};
```

```

        cudaFree(buffers[0]);
        cudaFree(buffers[1]);
        return output;
    }

private:
    nvinfer1::ICudaEngine* engine_ = nullptr;
    nvinfer1::IExecutionContext* context_ = nullptr;
    std::unique_ptr<Int8Calibrator> calibrator_;
    size_t input_size_, output_size_;
    Logger logger_;
};

```

G.1.2 批处理队列优化

批处理请求聚合，提高GPU利用率：

```

// batch_processor.cpp
class BatchProcessor {
    static const int MAX_BATCH = 8;
    static const int WAIT_MS = 10;

    struct InferRequest {
        cv::Mat image;
        std::promise<InferResult> result;
        std::chrono::steady_clock::time_point enqueue_time;
    };

    std::queue<InferRequest> queue_;
    std::mutex mtx_;
    std::condition_variable cv_;

public:
    void processingLoop() {
        while (running_) {
            std::vector<InferRequest> batch;

            {
                std::unique_lock<std::mutex> lock(mtx_);
                cv_.wait_for(lock, std::chrono::milliseconds(WAIT_MS),
                    [this] { return !queue_.empty(); });

                // 聚合批次
                while (!queue_.empty() && batch.size() < MAX_BATCH) {
                    batch.push_back(std::move(queue_.front()));
                    queue_.pop();
                }
            }

            if (!batch.empty()) {
                // 批量推理
                std::vector<cv::Mat> images;
                for (auto& req : batch) images.push_back(req.image);
            }
        }
    }
};

```

```

        auto results = engine_.inferBatch(images);

        for (size_t i = 0; i < batch.size(); i++) {
            batch[i].result.set_value(results[i]);
        }
    }
}
};

```

G.2 网络拓扑自动发现

G.2.1 mDNS服务注册

```

// mdns_service.cpp
#include <dns_sd.h>

class MdnsService {
    DNSServiceRef service_ref_ = nullptr;

public:
    bool registerService(const char* name, uint16_t port) {
        // 构建TXT记录
        TXTRecordRef txt;
        TXTRecordCreate(&txt, 0, nullptr);
        TXTRecordSetValue(&txt, "version", 3, "1.0");
        TXTRecordSetValue(&txt, "model", 8, "EdgeBox1");
        TXTRecordSetValue(&txt, "caps", 7, "ocr,tts");

        DNSServiceErrorType err = DNSServiceRegister(
            &service_ref_,
            0, // flags
            0, // interfaceIndex (all)
            name, // service name
            "_writech._tcp.", // service type
            nullptr, // domain (default)
            nullptr, // host (default)
            htons(port), // port
            TXTRecordGetLength(&txt), // txt length
            TXTRecordGetBytesPtr(&txt), // txt record
            registerCallback, // callback
            this // context
        );

        TXTRecordDeallocate(&txt);
        return err == kDNSServiceErr_NoError;
    }

    static void registerCallback(DNSServiceRef sdRef,
        DNSServiceFlags flags, DNSServiceErrorType err,
        const char* name, const char* type, const char* domain, void* ctx) {
        if (err == kDNSServiceErr_NoError) {

```

```

        LOG_INFO("mDNS registered: %s.%s%s", name, type, domain);
    }
}
};

```

G.3 本地模型管理

G.3.1 模型版本控制

```

# model_manager.py
import hashlib
import json
from pathlib import Path

class ModelManager:
    MODEL_DIR = Path("/opt/wrotech/models")
    MANIFEST_FILE = MODEL_DIR / "manifest.json"

    def __init__(self):
        self.manifest = self._load_manifest()

    def _load_manifest(self):
        if self.MANIFEST_FILE.exists():
            with open(self.MANIFEST_FILE) as f:
                return json.load(f)
        return {"models": {}}

    def verify_model(self, name: str) -> bool:
        """校验模型文件完整性"""
        if name not in self.manifest["models"]:
            return False

        info = self.manifest["models"][name]
        model_path = self.MODEL_DIR / info["filename"]

        if not model_path.exists():
            return False

        # SHA256校验
        sha256 = hashlib.sha256()
        with open(model_path, "rb") as f:
            for chunk in iter(lambda: f.read(8192), b''):
                sha256.update(chunk)

        return sha256.hexdigest() == info["sha256"]

    def update_model(self, name: str, url: str, expected_hash: str):
        """从云端更新模型"""
        import requests

        LOG.info(f"Downloading model {name} from {url}")
        response = requests.get(url, stream=True, timeout=300)

```

```

tmp_path = self.MODEL_DIR / f"{name}.tmp"
sha256 = hashlib.sha256()

with open(tmp_path, "wb") as f:
    for chunk in response.iter_content(chunk_size=8192):
        f.write(chunk)
        sha256.update(chunk)

actual_hash = sha256.hexdigest()
if actual_hash != expected_hash:
    tmp_path.unlink()
    raise ValueError(f"Hash mismatch: {actual_hash} != {expected_hash}")

# 原子替换
final_path = self.MODEL_DIR / f"{name}.engine"
tmp_path.rename(final_path)

self.manifest["models"][name] = {
    "filename": f"{name}.engine",
    "sha256": expected_hash,
    "updated_at": datetime.utcnow().isoformat()
}
self._save_manifest()
LOG.info(f"Model {name} updated successfully")

```

G.4 健康监控与告警

G.4.1 系统指标采集

```

# health_monitor.py
import psutil
import GPUUtil
import time
import threading

class HealthMonitor:
    ALERT_CPU_THRESHOLD = 90.0      # CPU使用率告警阈值
    ALERT_MEM_THRESHOLD = 85.0      # 内存使用率告警阈值
    ALERT_GPU_TEMP_THRESHOLD = 80    # GPU温度告警阈值 (°C)
    ALERT_DISK_THRESHOLD = 90.0     # 磁盘使用率告警阈值

    def __init__(self, alert_callback):
        self.alert_cb = alert_callback
        self.running = False

    def start(self):
        self.running = True
        self.thread = threading.Thread(target=self._monitor_loop, daemon=True)
        self.thread.start()

    def _monitor_loop(self):
        while self.running:
            metrics = self._collect_metrics()

```



```

        self._check_alerts(metrics)
        self._export_prometheus(metrics)
        time.sleep(10)

def _collect_metrics(self) -> dict:
    metrics = {
        "cpu_percent": psutil.cpu_percent(interval=1),
        "memory_percent": psutil.virtual_memory().percent,
        "disk_percent": psutil.disk_usage("/").percent,
        "net_bytes_sent": psutil.net_io_counters().bytes_sent,
        "net_bytes_recv": psutil.net_io_counters().bytes_recv,
        "timestamp": time.time()
    }

    # GPU指标 (Jetson Nano)
    try:
        gpus = GPUUtil.getGPUs()
        if gpus:
            gpu = gpus[0]
            metrics["gpu_load"] = gpu.load * 100
            metrics["gpu_temp"] = gpu.temperature
            metrics["gpu_mem_percent"] = gpu.memoryUtil * 100
    except Exception:
        pass

    return metrics

def _check_alerts(self, metrics: dict):
    if metrics["cpu_percent"] > self.ALERT_CPU_THRESHOLD:
        self.alert_cb("HIGH_CPU", f"CPU使用率 {metrics['cpu_percent']:.1f}%")

    if metrics["memory_percent"] > self.ALERT_MEM_THRESHOLD:
        self.alert_cb("HIGH_MEM", f"内存使用率 {metrics['memory_percent']:.1f}%")

    if metrics.get("gpu_temp", 0) > self.ALERT_GPU_TEMP_THRESHOLD:
        self.alert_cb("HIGH_GPU_TEMP", f"GPU温度 {metrics['gpu_temp']}°C")

def _export_prometheus(self, metrics: dict):
    """写入Prometheus指标文件"""
    lines = [
        f'edge_cpu_percent {metrics["cpu_percent"]}',
        f'edge_memory_percent {metrics["memory_percent"]}',
        f'edge_disk_percent {metrics["disk_percent"]}',
    ]
    if "gpu_load" in metrics:
        lines.append(f'edge_gpu_load_percent {metrics["gpu_load"]}')
        lines.append(f'edge_gpu_temp_celsius {metrics["gpu_temp"]}')

    with open("/opt/wrotech/metrics/node.prom", "w") as f:
        f.write("\n".join(lines) + "\n")

```

G.5 数据安全性与加密传输

G.5.1 TLS双向认证配置

```

# tls_config.py
import ssl

def create_mutual_tls_context(cert_path: str, key_path: str,
                             ca_path: str) -> ssl.SSLContext:
    """创建双向TLS认证上下文"""
    ctx = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
    ctx.verify_mode = ssl.CERT_REQUIRED
    ctx.check_hostname = True

    # 加载客户端证书和私钥
    ctx.load_cert_chain(certfile=cert_path, keyfile=key_path)

    # 加载CA证书，用于验证服务端
    ctx.load_verify_locations(ca_path)

    # 强制TLS 1.2+
    ctx.minimum_version = ssl.TLSVersion.TLSv1_2

    # 配置加密套件（仅允许强加密）
    ctx.set_ciphers(
        "ECDHE-ECDSA-AES256-GCM-SHA384:"
        "ECDHE-RSA-AES256-GCM-SHA384:"
        "ECDHE-ECDSA-CHACHA20-POLY1305"
    )

    return ctx

```

G.5.2 本地数据加密存储

```

# secure_storage.py
from cryptography.fernet import Fernet
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
import base64
import os

class SecureStorage:
    def __init__(self, passphrase: bytes):
        # 从设备唯一标识派生密钥
        salt = self._get_device_salt()
        kdf = PBKDF2HMAC(
            algorithm=hashes.SHA256(),
            length=32,
            salt=salt,
            iterations=100000
        )
        key = base64.urlsafe_b64encode(kdf.derive(passphrase))
        self.fernet = Fernet(key)

    def _get_device_salt(self) -> bytes:
        """读取设备唯一盐值（基于MAC地址和序列号）"""
        try:
            with open("/proc/net/if_inet6", "r") as f:

```

```
        mac_part = f.readline()[:16].encode()
    except Exception:
        mac_part = os.urandom(16)
    return mac_part

def encrypt_file(self, src: str, dst: str):
    with open(src, "rb") as f:
        data = f.read()
    encrypted = self.fernet.encrypt(data)
    with open(dst, "wb") as f:
        f.write(encrypted)

def decrypt_file(self, src: str) -> bytes:
    with open(src, "rb") as f:
        encrypted = f.read()
    return self.fernet.decrypt(encrypted)
```

本文档版权归深圳自然写科技有限公司所有，技术细节仅用于软件著作权登记鉴别。