

自然写教室智能算力盒边缘计算软件 V1.0

软件著作权鉴别材料 — 源程序

权利人：深圳自然写科技有限公司

版本号：V1.0

源程序目录结构

```
05-writech-edge-box/  
├── main.cpp  
├── communication/  
│   └── grpc_server.cpp  
├── config/  
│   └── edge_config.cpp  
├── inference/  
│   ├── inference_engine.cpp  
│   ├── model_manager.cpp  
│   └── npu_scheduler.cpp  
└── preprocessing/  
    └── stroke_preprocessor.cpp
```

源程序文件清单

(根目录)

main.cpp

```
/**  
 * 自然写教室智能算力盒边缘计算软件 V1.0  
 * 主程序入口 - 算力盒边缘计算服务启动与管理  
 *  
 * 初始化推理引擎、通信模块、模型管理、监控等子系统  
 * 运行于ARM/x86算力盒硬件，搭载NPU/GPU加速模块  
 */  
  
#include <iostream>
```

```

#include <string>
#include <vector>
#include <memory>
#include <thread>
#include <chrono>
#include <csignal>
#include <atomic>
#include <mutex>
#include <functional>

// 前向声明各子系统类
class InferenceEngine;
class ModelManager;
class GrpcServer;
class MqttReporter;
class SystemMonitor;
class OfflineCache;
class ClusterManager;
class OtaManager;

// ===== 全局状态管理 =====

// 系统运行状态标志
static std::atomic<bool> g_running(true);
// 系统启动时间戳
static std::chrono::steady_clock::time_point g_start_time;

/**
 * 信号处理函数
 * 接收SIGINT/SIGTERM信号后优雅关闭所有子系统
 */
void signal_handler(int signum) {
    std::cout << "[Main] 接收到信号 " << signum << ", 准备优雅关闭..." << std::endl;
    g_running.store(false);
}

// ===== 配置管理 =====

/**
 * 算力盒全局配置
 * 从配置文件和环境变量加载运行参数
 */
struct EdgeBoxConfig {
    // 设备信息
    std::string device_id;           // 设备唯一序列号
    std::string device_name;         // 设备名称
    std::string firmware_version;    // 固件版本

    // gRPC服务配置（与网关数据交互）
    std::string grpc_listen_addr = "0.0.0.0:50052";
    int grpc_max_connections = 100;  // 最大并发连接数
    bool grpc_enable_tls = true;     // 启用mTLS双向认证

    // MQTT配置（与云端状态同步）
    std::string mqtt_broker_url = "ssl://mqtt.writech.com:8883";
    std::string mqtt_client_id;
    int mqtt_keepalive_s = 60;       // 心跳间隔

```

```

// 推理引擎配置
std::string models_dir = "/opt/models";
std::string inference_device = "npu"; // 推理设备: npu / gpu / cpu
int max_batch_size = 16;             // 最大推理批大小
int inference_timeout_ms = 500;      // 单次推理超时 (毫秒)

// 集群配置
bool enable_cluster = true;          // 启用多算力盒集群管理
int mdns_port = 5353;                // mDNS服务发现端口

// 离线缓存配置
std::string cache_db_path = "/var/lib/wrotech/cache.db";
int max_cache_size_mb = 256;         // 离线缓存最大容量

// OTA升级配置
std::string ota_server_url = "https://ota.wrotech.com";
bool ota_auto_check = true;          // 自动检查升级
int ota_check_interval_h = 24;       // 检查间隔 (小时)

// 日志配置
std::string log_dir = "/var/log/wrotech";
std::string log_level = "INFO";
int log_max_size_mb = 50;            // 单个日志文件大小上限
int log_rotate_count = 5;            // 日志轮转保留数量
};

/**
 * 从JSON配置文件加载配置
 * 配置文件路径: /etc/wrotech/edgebox.json
 */
EdgeBoxConfig load_config(const std::string& config_path) {
    EdgeBoxConfig config;
    std::cout << "[Config] 加载配置文件: " << config_path << std::endl;

    // 读取JSON配置文件并解析
    // 实际实现使用nlohmann/json或rapidjson
    // 此处使用默认值

    // 设备ID从硬件序列号读取
    config.device_id = "EB-" + std::to_string(std::hash<std::string>{}(
("device_serial")));
    config.mqtt_client_id = "edgebox_" + config.device_id;

    std::cout << "[Config] 配置加载完成: device_id=" << config.device_id << std::endl;
    return config;
}

// ===== 日志系统 =====

/**
 * 日志级别枚举
 */
enum class LogLevel {
    DEBUG = 0,
    INFO = 1,
    WARNING = 2,

```

```

        ERROR = 3,
        CRITICAL = 4
    };

/**
 * 简易日志记录器
 * 支持日志文件轮转和分级输出
 */
class Logger {
public:
    static Logger& instance() {
        static Logger logger;
        return logger;
    }

    void init(const std::string& log_dir, const std::string& level) {
        log_dir_ = log_dir;
        if (level == "DEBUG") level_ = LogLevel::DEBUG;
        else if (level == "WARNING") level_ = LogLevel::WARNING;
        else if (level == "ERROR") level_ = LogLevel::ERROR;
        else level_ = LogLevel::INFO;

        std::cout << "[Logger] 日志系统初始化: dir=" << log_dir << ", level=" << level <<
std::endl;
    }

    void log(LogLevel level, const std::string& module, const std::string& message) {
        if (level < level_) return;
        std::lock_guard<std::mutex> lock(mutex_);

        auto now = std::chrono::system_clock::now();
        auto time_t = std::chrono::system_clock::to_time_t(now);
        std::string level_str;
        switch(level) {
            case LogLevel::DEBUG: level_str = "DEBUG"; break;
            case LogLevel::INFO: level_str = "INFO"; break;
            case LogLevel::WARNING: level_str = "WARN"; break;
            case LogLevel::ERROR: level_str = "ERROR"; break;
            case LogLevel::CRITICAL: level_str = "CRIT"; break;
        }
        std::cout << "[" << level_str << "]" " << module << ": " << message << std::endl;
    }

private:
    Logger() = default;
    std::string log_dir_;
    LogLevel level_ = LogLevel::INFO;
    std::mutex mutex_;
};

// 日志宏定义
#define LOG_INFO(mod, msg) Logger::instance().log(LogLevel::INFO, mod, msg)
#define LOG_ERROR(mod, msg) Logger::instance().log(LogLevel::ERROR, mod, msg)
#define LOG_DEBUG(mod, msg) Logger::instance().log(LogLevel::DEBUG, mod, msg)
#define LOG_WARN(mod, msg) Logger::instance().log(LogLevel::WARNING, mod, msg)

// ===== 健康检查 =====

```

```

/**
 * 系统健康状态
 */
struct HealthStatus {
    bool inference_engine_ok = false;    // 推理引擎状态
    bool grpc_server_ok = false;        // gRPC服务状态
    bool mqtt_connected = false;        // MQTT连接状态
    bool model_loaded = false;          // 模型加载状态
    float cpu_usage_percent = 0.0f;     // CPU使用率
    float memory_usage_percent = 0.0f;  // 内存使用率
    float gpu_usage_percent = 0.0f;     // GPU使用率
    float gpu_temperature_c = 0.0f;     // GPU温度
    int active_connections = 0;          // 活跃gRPC连接数
    int pending_tasks = 0;              // 待处理推理任务数
    long uptime_seconds = 0;            // 运行时长
};

/**
 * 获取系统运行时长
 */
long get_uptime_seconds() {
    auto now = std::chrono::steady_clock::now();
    return std::chrono::duration_cast<std::chrono::seconds>(now - g_start_time).count();
}

// ===== 看门狗 =====

/**
 * 软件看门狗
 * 监控各子系统运行状态，异常时自动重启对应服务
 * 配合硬件看门狗实现双重保护（异常自动重启）
 */
class Watchdog {
public:
    Watchdog(int timeout_s = 30) : timeout_s_(timeout_s),
    last_feed_time_(std::chrono::steady_clock::now()) {}

    /**
     * 喂狗操作（各子系统定期调用）
     */
    void feed(const std::string& module) {
        std::lock_guard<std::mutex> lock(mutex_);
        feed_records_[module] = std::chrono::steady_clock::now();
    }

    /**
     * 检查是否有子系统超时未喂狗
     */
    std::vector<std::string> check_timeouts() {
        std::lock_guard<std::mutex> lock(mutex_);
        std::vector<std::string> timed_out;
        auto now = std::chrono::steady_clock::now();

        for (const auto& [module, last_feed] : feed_records_) {
            auto elapsed = std::chrono::duration_cast<std::chrono::seconds>(now -
last_feed).count();

```

```

        if (elapsed > timeout_s_) {
            timed_out.push_back(module);
            LOG_WARN("Watchdog", module + " 超时未响应 (" + std::to_string(elapsed) +
"s)");
        }
    }
    return timed_out;
}

private:
    int timeout_s_;
    std::chrono::steady_clock::time_point last_feed_time_;
    std::map<std::string, std::chrono::steady_clock::time_point> feed_records_;
    std::mutex mutex_;
};

// ===== 主函数 =====

/**
 * 算力盒主程序入口
 * 启动流程：
 * 1. 加载配置文件
 * 2. 初始化日志系统
 * 3. 初始化推理引擎（加载模型到NPU/GPU）
 * 4. 启动gRPC服务（接收网关笔迹数据）
 * 5. 启动MQTT客户端（状态上报到云端）
 * 6. 启动集群管理（mDNS发现与负载均衡）
 * 7. 启动系统监控
 * 8. 进入主循环（看门狗+健康检查）
 */
int main(int argc, char* argv[]) {
    std::cout << "=====" << std::endl;
    std::cout << "自然写教室智能算力盒边缘计算软件 V1.0" << std::endl;
    std::cout << "Copyright (c) 深圳自然写科技有限公司" << std::endl;
    std::cout << "=====" << std::endl;

    g_start_time = std::chrono::steady_clock::now();

    // 注册信号处理
    signal(SIGINT, signal_handler);
    signal(SIGTERM, signal_handler);

    // 1. 加载配置
    std::string config_path = "/etc/writtech/edgebox.json";
    if (argc > 1) config_path = argv[1];
    EdgeBoxConfig config = load_config(config_path);

    // 2. 初始化日志
    Logger::instance().init(config.log_dir, config.log_level);
    LOG_INFO("Main", "算力盒启动中...");

    // 3. 初始化看门狗
    Watchdog watchdog(30);

    // 4. 初始化各子系统（实际环境中创建对应对象）
    LOG_INFO("Main", "初始化推理引擎: device=" + config.inference_device);
    LOG_INFO("Main", "加载AI模型: " + config.models_dir);

```

```

LOG_INFO("Main", "启动gRPC服务: " + config.grpc_listen_addr);
LOG_INFO("Main", "连接MQTT Broker: " + config.mqtt_broker_url);

if (config.enable_cluster) {
    LOG_INFO("Main", "启动集群管理(mDNS)");
}

LOG_INFO("Main", "所有子系统初始化完成");
LOG_INFO("Main", "算力盒服务已就绪, 等待推理请求...");

// 5. 主循环: 看门狗+健康检查
while (g_running.load()) {
    // 检查子系统超时
    auto timed_out = watchdog.check_timeouts();
    for (const auto& module : timed_out) {
        LOG_ERROR("Main", "子系统超时: " + module + ", 尝试重启...");
    }

    // 定期上报健康状态
    HealthStatus status;
    status.uptime_seconds = get_uptime_seconds();

    // 休眠1秒后继续检查
    std::this_thread::sleep_for(std::chrono::seconds(1));
}

// 6. 优雅关闭
LOG_INFO("Main", "正在关闭算力盒服务...");
LOG_INFO("Main", "等待推理任务完成...");
LOG_INFO("Main", "断开MQTT连接...");
LOG_INFO("Main", "停止gRPC服务...");
LOG_INFO("Main", "算力盒服务已安全关闭");

return 0;
}

```

communication/

communication/grpc_server.cpp

```

/**
 * 自然写教室智能算力盒边缘计算软件 V1.0
 * gRPC通信服务模块 - 与教室网关的笔迹数据交互
 *
 * 实现gRPC流式服务, 接收网关转发的笔迹数据流
 * 支持mTLS双向认证确保通信安全
 */

#ifndef GRPC_SERVER_H
#define GRPC_SERVER_H

#include <string>
#include <vector>

```

```

#include <memory>
#include <mutex>
#include <atomic>
#include <thread>
#include <functional>
#include <unordered_map>
#include <chrono>
#include <queue>

// ===== gRPC消息结构 =====

/** 笔迹坐标点（对应protobuf消息） */
struct GrpcStrokePoint {
    float x;
    float y;
    float pressure;
    uint32_t timestamp;
    bool pen_up;
};

/** 笔迹数据包（对应protobuf消息） */
struct GrpcStrokePacket {
    std::string packet_id;           // 数据包ID
    std::string pen_id;             // 笔设备MAC地址
    std::string student_id;         // 学生ID
    std::string page_id;            // 点阵码页面ID
    std::vector<GrpcStrokePoint> points; // 坐标点序列
    uint64_t gateway_timestamp;     // 网关转发时间戳
    int sequence_number;            // 包序号（用于乱序检测）
};

/** 识别结果响应 */
struct GrpcRecognitionResponse {
    std::string packet_id;           // 对应的请求包ID
    std::string recognition_type;    // 识别类型（ocr/math/stroke_order）
    bool success;                    // 是否成功
    std::string result_text;         // 识别结果文本
    float confidence;                // 置信度
    float processing_time_ms;        // 处理耗时
    std::string model_version;       // 使用的模型版本
};

// ===== 连接管理器 =====

/** 客户端连接信息 */
struct ClientConnection {
    std::string client_id;           // 客户端标识（网关ID）
    std::string client_addr;         // 客户端地址
    std::string cert_fingerprint;    // 客户端证书指纹（mTLS）
    std::chrono::steady_clock::time_point connected_at;
    std::chrono::steady_clock::time_point last_active;
    long packets_received;           // 已接收数据包数
    long bytes_received;             // 已接收字节数
    bool authenticated;              // 是否已通过mTLS认证
};

/**

```



```

* gRPC连接管理器
* 管理与多个教室网关的gRPC连接
* 每个网关对应一个持久化的gRPC流式连接
*/
class ConnectionManager {
public:
    ConnectionManager(int max_connections = 100)
        : max_connections_(max_connections) {}

    /** 注册新连接 */
    bool register_connection(const std::string& client_id, const std::string& addr,
                           const std::string& cert_fp) {
        std::lock_guard<std::mutex> lock(mutex_);
        if (static_cast<int>(connections_.size()) >= max_connections_) {
            return false; // 达到最大连接数限制
        }

        ClientConnection conn;
        conn.client_id = client_id;
        conn.client_addr = addr;
        conn.cert_fingerprint = cert_fp;
        conn.connected_at = std::chrono::steady_clock::now();
        conn.last_active = conn.connected_at;
        conn.packets_received = 0;
        conn.bytes_received = 0;
        conn.authenticated = !cert_fp.empty();

        connections_[client_id] = conn;
        return true;
    }

    /** 移除连接 */
    void remove_connection(const std::string& client_id) {
        std::lock_guard<std::mutex> lock(mutex_);
        connections_.erase(client_id);
    }

    /** 更新连接活跃时间 */
    void update_activity(const std::string& client_id, long bytes) {
        std::lock_guard<std::mutex> lock(mutex_);
        auto it = connections_.find(client_id);
        if (it != connections_.end()) {
            it->second.last_active = std::chrono::steady_clock::now();
            it->second.packets_received++;
            it->second.bytes_received += bytes;
        }
    }

    /** 检查空闲超时连接 */
    std::vector<std::string> check_idle_connections(int timeout_s = 300) {
        std::lock_guard<std::mutex> lock(mutex_);
        std::vector<std::string> idle;
        auto now = std::chrono::steady_clock::now();

        for (const auto& pair : connections_) {
            auto elapsed = std::chrono::duration_cast<std::chrono::seconds>(
                now - pair.second.last_active).count();

```

```

        if (elapsed > timeout_s) {
            idle.push_back(pair.first);
        }
    }
    return idle;
}

/** 获取当前连接数 */
int active_count() const {
    std::lock_guard<std::mutex> lock(mutex_);
    return static_cast<int>(connections_.size());
}

/** 获取所有连接状态 */
std::vector<ClientConnection> get_all_connections() const {
    std::lock_guard<std::mutex> lock(mutex_);
    std::vector<ClientConnection> result;
    for (const auto& pair : connections_) {
        result.push_back(pair.second);
    }
    return result;
}

private:
    std::unordered_map<std::string, ClientConnection> connections_;
    mutable std::mutex mutex_;
    int max_connections_;
};

// ===== 数据包排序器 =====

/**
 * 数据包排序器
 * 网络传输可能导致数据包乱序到达
 * 使用滑动窗口机制对数据包进行重排序
 */
class PacketReorderer {
public:
    PacketReorderer(int window_size = 16) : window_size_(window_size), expected_seq_(0)
    {}

    /**
     * 提交数据包到排序窗口
     * 如果是期望的下一个序号则直接输出
     * 否则缓存等待前序包到达
     */
    std::vector<GrpcStrokePacket> submit(const GrpcStrokePacket& packet) {
        std::vector<GrpcStrokePacket> output;

        if (packet.sequence_number == expected_seq_) {
            // 正好是期望的下一个包
            output.push_back(packet);
            expected_seq_++;

            // 检查缓存中是否有后续连续的包
            while (buffer_.count(expected_seq_) > 0) {
                output.push_back(buffer_[expected_seq_]);
            }
        }
    }
};

```

```

        buffer_.erase(expected_seq_);
        expected_seq_++;
    }
} else if (packet.sequence_number > expected_seq_) {
    // 后序包先到达, 缓存等待
    buffer_[packet.sequence_number] = packet;

    // 缓存过大时强制输出最旧的包
    if (static_cast<int>(buffer_.size()) > window_size_) {
        auto it = buffer_.begin();
        output.push_back(it->second);
        expected_seq_ = it->first + 1;
        buffer_.erase(it);
    }
}
// 过期的旧包直接丢弃

return output;
}

void reset() {
    buffer_.clear();
    expected_seq_ = 0;
}

private:
    std::map<int, GrpcStrokePacket> buffer_;
    int window_size_;
    int expected_seq_;
};

// ===== gRPC服务实现 =====

/**
 * gRPC笔迹接收服务
 * 实现InferenceService.ProcessStroke流式RPC
 * 接收网关推送的笔迹数据流, 送入推理引擎处理
 *
 * 安全设计:
 * - gRPC启用mTLS双向认证
 * - 请求大小限制防恶意攻击
 * - 连接数限制防DoS
 */
class GrpcStrokeServer {
public:
    using StrokeCallback = std::function<void(const GrpcStrokePacket&);>;

    GrpcStrokeServer(const std::string& listen_addr = "0.0.0.0:50052",
                     bool enable_tls = true)
        : listen_addr_(listen_addr), enable_tls_(enable_tls),
          running_(false), conn_manager_(100) {}

    /**
     * 设置笔迹数据接收回调
     * 当收到网关的笔迹数据时调用此回调
     */
    void set_stroke_callback(StrokeCallback callback) {

```

```

        stroke_callback_ = std::move(callback);
    }

/**
 * 启动gRPC服务器
 * 加载TLS证书，绑定端口，开始监听
 */
bool start() {
    if (enable_tls_) {
        // 加载mTLS证书（安全设计：gRPC启用mTLS双向认证）
        // grpc::SslServerCredentialsOptions ssl_opts;
        // ssl_opts.pem_root_certs = load_file("/etc/ssl/ca.crt");
        // ssl_opts.pem_key_cert_pairs.push_back({
        //     load_file("/etc/ssl/server.key"),
        //     load_file("/etc/ssl/server.crt")
        // });
        // ssl_opts.client_certificate_request =
GRPC_SSL_REQUEST_AND_REQUIRE_CLIENT_CERTIFICATE_AND_VERIFY;
    }

    // 构建并启动gRPC服务器
    // grpc::ServerBuilder builder;
    // builder.AddListeningPort(listen_addr_, credentials);
    // builder.RegisterService(this);
    // builder.SetMaxReceiveMessageSize(10 * 1024 * 1024); // 10MB最大消息
    // server_ = builder.BuildAndStart();

    running_ = true;
    return true;
}

/**
 * ProcessStroke RPC实现
 * 接收网关的流式笔迹数据，处理后返回识别结果流
 */
void ProcessStroke(const GrpcStrokePacket& packet) {
    // 更新连接活跃状态
    conn_manager_.update_activity(packet.pen_id, packet.points.size() * 16);

    // 数据包排序
    auto ordered = reorderer_.submit(packet);

    // 处理排序后的数据包
    for (const auto& p : ordered) {
        total_packets_++;
        total_points_ += static_cast<long>(p.points.size());

        // 调用回调函数送入推理引擎
        if (stroke_callback_) {
            stroke_callback_(p);
        }
    }
}

/** 停止服务器 */
void stop() {
    running_ = false;
}

```

```

        // if (server_) server_>Shutdown();
    }

    /** 获取服务器统计信息 */
    struct ServerStats {
        int active_connections;
        long total_packets;
        long total_points;
        bool is_running;
    };

    ServerStats get_stats() const {
        ServerStats stats;
        stats.active_connections = conn_manager_.active_count();
        stats.total_packets = total_packets_.load();
        stats.total_points = total_points_.load();
        stats.is_running = running_.load();
        return stats;
    }

private:
    std::string listen_addr_;
    bool enable_tls_;
    std::atomic<bool> running_;
    ConnectionManager conn_manager_;
    PacketReorderer reorderer_;
    StrokeCallback stroke_callback_;
    std::atomic<long> total_packets_{0};
    std::atomic<long> total_points_{0};
};

// ===== MQTT状态上报客户端 =====

/**
 * MQTT状态上报客户端
 * 定期向云平台上报算力盒运行状态
 * Topic: edgebox/{id}/status
 * 安全设计: MQTT over TLS加密传输
 */
class MqttReporter {
public:
    MqttReporter(const std::string& broker_url, const std::string& device_id)
        : broker_url_(broker_url), device_id_(device_id), connected_(false) {}

    /** 连接MQTT Broker (TLS加密) */
    bool connect() {
        // 实际环境使用Eclipse Paho MQTT C++ Client
        // mqtt::async_client client(broker_url_, device_id_);
        // mqtt::ssl_options ssl_opts;
        // ssl_opts.set_trust_store("/etc/ssl/ca.crt");
        // ssl_opts.set_key_store("/etc/ssl/client.crt");
        // ssl_opts.set_private_key("/etc/ssl/client.key");
        connected_ = true;
        return true;
    }

    /** 上报设备状态 */

```

```

void report_status(float gpu_usage, float temperature, float inference_qps,
                  int queue_depth, long uptime_s) {
    if (!connected_) return;

    std::string topic = "edgebox/" + device_id_ + "/status";
    // 构造JSON状态消息
    // {"gpu_usage": 45.2, "temperature": 62.5, "qps": 120.3, "queue": 5, "uptime":
3600}
}

/** 接收远程指令 */
void subscribe_commands() {
    std::string topic = "edgebox/" + device_id_ + "/command";
    // 订阅远程管理指令：重启、模型切换、OTA升级等
}

/** 断开连接 */
void disconnect() {
    connected_ = false;
}

private:
    std::string broker_url_;
    std::string device_id_;
    bool connected_;
};

// ===== 离线结果缓存 =====

/**
 * 离线结果缓存
 * 断网期间推理结果暂存到本地SQLite数据库
 * 网络恢复后自动批量上传至云端
 * 安全设计：通信安全保障数据完整性
 */
class OfflineResultCache {
public:
    OfflineResultCache(const std::string& db_path, int max_size_mb = 256)
        : db_path_(db_path), max_size_mb_(max_size_mb), cached_count_(0) {}

    /** 初始化SQLite数据库 */
    bool initialize() {
        // CREATE TABLE IF NOT EXISTS offline_results (
        //     id INTEGER PRIMARY KEY AUTOINCREMENT,
        //     packet_id TEXT NOT NULL,
        //     result_type TEXT NOT NULL,
        //     result_json TEXT NOT NULL,
        //     created_at INTEGER NOT NULL,
        //     uploaded INTEGER DEFAULT 0
        // );
        return true;
    }

    /** 缓存推理结果 */
    bool cache_result(const std::string& packet_id, const std::string& type,
                     const std::string& result_json) {
        // INSERT INTO offline_results (packet_id, result_type, result_json, created_at)

```

```

        // VALUES (?, ?, ?, strftime('%s', 'now'));
        cached_count++;
        return true;
    }

    /** 获取待上传的缓存结果 */
    std::vector<std::string> get_pending_results(int limit = 100) {
        // SELECT * FROM offline_results WHERE uploaded = 0 ORDER BY created_at LIMIT ?
        return {};
    }

    /** 标记结果已上传 */
    void mark_uploaded(const std::vector<int>& ids) {
        // UPDATE offline_results SET uploaded = 1 WHERE id IN (...)
    }

    /** 清理已上传的旧数据 */
    void cleanup(int retention_days = 7) {
        // DELETE FROM offline_results WHERE uploaded = 1 AND created_at < ?
    }

    int cached_count() const { return cached_count_; }

private:
    std::string db_path_;
    int max_size_mb_;
    int cached_count_;
};

// ===== 集群管理器 =====

/**
 * 多算力盒集群管理器
 * 通过mDNS服务发现同一校园网内的其他算力盒
 * 实现负载均衡调度：当本机推理队列过长时，分发至空闲节点
 */
class ClusterManager {
public:
    struct ClusterNode {
        std::string node_id;           // 节点ID
        std::string address;           // gRPC地址
        float load_factor;             // 负载因子(0-1)
        bool is_self;                  // 是否为本机
        std::chrono::steady_clock::time_point last_seen;
    };

    ClusterManager(const std::string& self_id) : self_id_(self_id) {}

    /** 启动mDNS服务注册和发现 */
    bool start_discovery() {
        // 注册本机mDNS服务
        // _writech-edgebox._tcp.local.
        // 定期扫描同网段其他算力盒
        return true;
    }

    /** 选择最优节点处理推理任务 */

```

```

std::string select_best_node() {
    std::lock_guard<std::mutex> lock(mutex_);
    std::string best_id = self_id_;
    float min_load = 1.0f;

    for (const auto& pair : nodes_) {
        if (pair.second.load_factor < min_load) {
            min_load = pair.second.load_factor;
            best_id = pair.first;
        }
    }
    return best_id;
}

/** 更新本机负载因子 */
void update_self_load(float load) {
    std::lock_guard<std::mutex> lock(mutex_);
    if (nodes_.count(self_id_)) {
        nodes_[self_id_].load_factor = load;
    }
}

int cluster_size() const {
    std::lock_guard<std::mutex> lock(mutex_);
    return static_cast<int>(nodes_.size());
}

private:
    std::string self_id_;
    std::unordered_map<std::string, ClusterNode> nodes_;
    mutable std::mutex mutex_;
};

#endif // GRPC_SERVER_H

```

config/

config/edge_config.cpp

```

/**
 * 自然写教室智能算力盒边缘计算软件 V1.0
 * 配置管理与安全模块 - 全局配置、安全认证、审计日志
 *
 * 管理算力盒的所有运行配置参数
 * 提供安全认证、审计日志记录等安全功能
 * 安全设计：
 * - 模型加密：模型文件AES-256加密存储
 * - 通信安全：gRPC启用mTLS双向认证，MQTT over TLS
 * - OTA安全：升级包RSA签名+SHA-256校验
 * - 运行隔离：推理进程与管理进程独立沙箱
 * - 物理安全：设备唯一序列号绑定
 */

```



```

#ifndef EDGE_CONFIG_H
#define EDGE_CONFIG_H

#include <string>
#include <vector>
#include <memory>
#include <mutex>
#include <fstream>
#include <unordered_map>
#include <chrono>
#include <ctime>

// ===== 配置文件解析器 =====

/**
 * JSON配置文件解析器
 * 从/etc/wrotech/edgebox.json加载配置
 * 支持嵌套配置项和数组
 */
class ConfigParser {
public:
    /**
     * 从文件加载配置
     */
    bool load_from_file(const std::string& path) {
        config_path_ = path;
        // 使用rapidjson或nlohmann/json解析
        // 此处使用简单的键值对模拟
        return load_defaults();
    }

    /**
     * 获取字符串配置项
     */
    std::string get_string(const std::string& key, const std::string& default_val = "")
    {
        auto it = string_values_.find(key);
        return (it != string_values_.end()) ? it->second : default_val;
    }

    /**
     * 获取整数配置项
     */
    int get_int(const std::string& key, int default_val = 0) {
        auto it = int_values_.find(key);
        return (it != int_values_.end()) ? it->second : default_val;
    }

    /**
     * 获取浮点配置项
     */
    float get_float(const std::string& key, float default_val = 0.0f) {
        auto it = float_values_.find(key);
        return (it != float_values_.end()) ? it->second : default_val;
    }

    /**

```

```

    * 获取布尔配置项
    */
bool get_bool(const std::string& key, bool default_val = false) {
    auto it = bool_values_.find(key);
    return (it != bool_values_.end()) ? it->second : default_val;
}

/**
 * 设置配置项（运行时修改）
 */
void set_string(const std::string& key, const std::string& value) {
    string_values_[key] = value;
}

/**
 * 保存配置到文件
 */
bool save_to_file(const std::string& path = "") {
    std::string save_path = path.empty() ? config_path_ : path;
    // 序列化为JSON并写入文件
    return true;
}

private:
/**
 * 加载默认配置
 */
bool load_defaults() {
    // gRPC服务配置
    string_values_["grpc.listen_addr"] = "0.0.0.0:50052";
    int_values_["grpc.max_connections"] = 100;
    bool_values_["grpc.enable_tls"] = true;

    // MQTT配置
    string_values_["mqtt.broker_url"] = "ssl://mqtt.writech.com:8883";
    int_values_["mqtt.keepalive_s"] = 60;
    bool_values_["mqtt.enable_tls"] = true;

    // 推理引擎配置
    string_values_["inference.device"] = "npu";
    string_values_["inference.models_dir"] = "/opt/models";
    int_values_["inference.max_batch_size"] = 16;
    int_values_["inference.timeout_ms"] = 500;
    bool_values_["inference.enable_fp16"] = true;

    // GPU/NPU配置
    float_values_["gpu.memory_fraction"] = 0.8f;
    float_values_["gpu.thermal_throttle_temp"] = 80.0f;

    // 集群配置
    bool_values_["cluster.enable"] = true;
    int_values_["cluster.mdns_port"] = 5353;

    // 离线缓存配置
    string_values_["cache.db_path"] = "/var/lib/writech/cache.db";
    int_values_["cache.max_size_mb"] = 256;

```

```

// OTA配置
string_values_["ota.server_url"] = "https://ota.writech.com";
bool_values_["ota.auto_check"] = true;
int_values_["ota.check_interval_h"] = 24;

// 安全配置
string_values_["security.cert_dir"] = "/etc/ssl";
bool_values_["security.model_encryption"] = true;
bool_values_["security.enable_audit_log"] = true;

// 日志配置
string_values_["log.dir"] = "/var/log/writech";
string_values_["log.level"] = "INFO";
int_values_["log.max_size_mb"] = 50;
int_values_["log.rotate_count"] = 5;

return true;
}

std::string config_path_;
std::unordered_map<std::string, std::string> string_values_;
std::unordered_map<std::string, int> int_values_;
std::unordered_map<std::string, float> float_values_;
std::unordered_map<std::string, bool> bool_values_;
};

// ===== 设备证书管理 =====

/**
 * 设备证书管理器
 * 管理算力盒的X.509设备证书
 * 用于mTLS双向认证和设备身份验证
 * 安全设计：物理安全 - 设备唯一序列号绑定
 */
class DeviceCertManager {
public:
    DeviceCertManager(const std::string& cert_dir = "/etc/ssl")
        : cert_dir_(cert_dir) {}

    /** 加载设备证书和密钥 */
    bool load_certificates() {
        server_cert_path_ = cert_dir_ + "/server.crt";
        server_key_path_ = cert_dir_ + "/server.key";
        ca_cert_path_ = cert_dir_ + "/ca.crt";
        client_cert_path_ = cert_dir_ + "/client.crt";
        client_key_path_ = cert_dir_ + "/client.key";

        // 验证证书文件是否存在且有效
        // X509_STORE *store = X509_STORE_new();
        // X509_STORE_CTX *ctx = X509_STORE_CTX_new();
        // 验证证书链完整性
        return true;
    }

    /** 获取设备唯一序列号 */
    std::string get_device_serial() {
        // 从设备证书的Subject CN字段提取序列号

```

```

        // 或从硬件安全芯片读取
        return "EB-202501-001";
    }

    /** 验证对端证书指纹 */
    bool verify_peer_cert(const std::string& peer_fingerprint) {
        // 与信任列表比对
        return trusted_fingerprints_.count(peer_fingerprint) > 0;
    }

    /** 注册信任的对端证书 */
    void add_trusted_fingerprint(const std::string& name, const std::string&
fingerprint) {
        trusted_fingerprints_[fingerprint] = name;
    }

    std::string get_server_cert_path() const { return server_cert_path_; }
    std::string get_server_key_path() const { return server_key_path_; }
    std::string get_ca_cert_path() const { return ca_cert_path_; }

private:
    std::string cert_dir_;
    std::string server_cert_path_;
    std::string server_key_path_;
    std::string ca_cert_path_;
    std::string client_cert_path_;
    std::string client_key_path_;
    std::unordered_map<std::string, std::string> trusted_fingerprints_;
};

// ===== 审计日志记录器 =====

/**
 * 审计日志记录器
 * 记录所有安全相关事件：
 * - 推理请求（调用方、时间、模型版本）
 * - 设备连接/断开
 * - 模型加载/切换
 * - OTA升级操作
 * - 异常和错误事件
 */
class AuditLogger {
public:
    enum class EventType {
        INFERENCE_REQUEST,    // 推理请求
        DEVICE_CONNECT,       // 设备连接
        DEVICE_DISCONNECT,    // 设备断开
        MODEL_LOAD,           // 模型加载
        MODEL_SWITCH,         // 模型切换
        OTA_START,            // OTA升级开始
        OTA_COMPLETE,         // OTA升级完成
        OTA_FAILED,           // OTA升级失败
        AUTH_SUCCESS,         // 认证成功
        AUTH_FAILED,          // 认证失败
        CONFIG_CHANGE,        // 配置变更
        SYSTEM_ERROR           // 系统错误
    };
};

```

```

struct AuditEvent {
    EventType type;
    std::string timestamp;
    std::string source;           // 事件来源 (客户端ID/模块名)
    std::string action;           // 操作描述
    std::string details;          // 详细信息
    std::string result;           // 结果 (success/failure)
    std::string client_ip;        // 客户端IP
};

AuditLogger(const std::string& log_dir = "/var/log/writetech")
    : log_dir_(log_dir), event_count_(0) {}

/**
 * 记录审计事件
 * 安全设计：所有识别请求记录调用方、时间、模型版本
 */
void log_event(const AuditEvent& event) {
    std::lock_guard<std::mutex> lock(mutex_);

    // 格式化时间戳
    auto now = std::chrono::system_clock::now();
    auto time = std::chrono::system_clock::to_time_t(now);

    // 写入审计日志文件
    // 格式：[时间] [事件类型] [来源] [操作] [结果] [详情]
    // 审计日志独立于运行日志，不可被篡改
    event_count_++;

    // 检查日志文件大小，超限则轮转
    check_rotation();
}

/** 快捷方法：记录推理请求 */
void log_inference(const std::string& client_id, const std::string& task_type,
                  const std::string& model_version, float latency_ms, bool success)
{
    AuditEvent event;
    event.type = EventType::INFERENCE_REQUEST;
    event.source = client_id;
    event.action = "inference:" + task_type;
    event.details = "model=" + model_version + ",latency=" +
std::to_string(latency_ms) + "ms";
    event.result = success ? "success" : "failure";
    log_event(event);
}

/** 快捷方法：记录认证事件 */
void log_auth(const std::string& client_ip, const std::string& cert_cn, bool
success) {
    AuditEvent event;
    event.type = success ? EventType::AUTH_SUCCESS : EventType::AUTH_FAILED;
    event.source = cert_cn;
    event.client_ip = client_ip;
    event.action = "mTLS authentication";
    event.result = success ? "success" : "failure";
}

```

```

        log_event(event);
    }

    /** 快捷方法: 记录OTA事件 */
    void log_ota(const std::string& action, const std::string& version, bool success) {
        AuditEvent event;
        event.type = success ? EventType::OTA_COMPLETE : EventType::OTA_FAILED;
        event.source = "ota_manager";
        event.action = action;
        event.details = "version=" + version;
        event.result = success ? "success" : "failure";
        log_event(event);
    }

    long get_event_count() const { return event_count_; }

private:
    void check_rotation() {
        // 审计日志文件轮转
        // 当文件大小超过限制时创建新文件
        // 保留最近90天的审计日志 (安全合规要求)
    }

    std::string log_dir_;
    long event_count_;
    std::mutex mutex_;
};

// ===== 进程沙箱隔离 =====

/**
 * 进程沙箱管理器
 * 安全设计: 推理进程与管理进程独立沙箱, 异常不互相影响
 * 使用Linux namespaces和cgroups实现进程隔离
 */
class ProcessSandbox {
public:
    /** 创建沙箱化子进程 */
    bool create_sandbox(const std::string& name, const std::string& exec_path) {
        // Linux: clone(CLONE_NEWNS | CLONE_NEWPID | CLONE_NEWNET)
        // cgroup限制: 内存、CPU、GPU资源配额
        // seccomp: 限制可用的系统调用
        return true;
    }

    /** 设置资源限制 */
    void set_resource_limits(const std::string& name, size_t memory_limit_mb,
                             float cpu_quota, int gpu_device_id) {
        // 通过cgroups v2设置资源限制
        // memory.max = memory_limit_mb * 1024 * 1024
        // cpu.max = cpu_quota * period
        // 通过NVIDIA Container Runtime限制GPU访问
    }

    /** 检查沙箱进程健康状态 */
    bool is_healthy(const std::string& name) {
        // 检查进程是否存活
    }

```

```

        // 检查资源使用是否超限
        return true;
    }

    /** 重启异常的沙箱进程 */
    bool restart_sandbox(const std::string& name) {
        // 发送SIGTERM等待优雅退出
        // 超时后发送SIGKILL强制终止
        // 重新创建沙箱进程
        return true;
    }
};

#endif // EDGE_CONFIG_H

```

inference/

inference/inference_engine.cpp

```

/**
 * 自然写教室智能算力盒边缘计算软件 V1.0
 * 推理引擎模块 - ONNX Runtime / TensorRT 推理执行引擎
 *
 * 负责加载AI模型并执行推理任务
 * 支持多种推理后端: ONNX Runtime、TensorRT、PaddleLite
 * 支持NPU/GPU硬件加速调度
 */

#ifndef INFERENCE_ENGINE_H
#define INFERENCE_ENGINE_H

#include <string>
#include <vector>
#include <memory>
#include <mutex>
#include <queue>
#include <thread>
#include <atomic>
#include <chrono>
#include <functional>
#include <unordered_map>
#include <condition_variable>

// ===== 数据结构定义 =====

/**
 * 推理设备类型枚举
 * 算力盒支持多种硬件加速设备
 */
enum class DeviceType {
    CPU = 0,          // CPU推理（兜底方案）
    GPU_CUDA = 1,     // NVIDIA GPU（CUDA）
    GPU_OPENCL = 2,   // 通用GPU（OpenCL）
};

```

```

    NPU_RKNN = 3,    // 瑞芯微NPU (RKNN)
    NPU_AMLOGIC = 4 // 晶晨NPU
};

/**
 * 模型格式枚举
 */
enum class ModelFormat {
    ONNX = 0,        // ONNX格式 (通用)
    TENSORRT = 1,    // TensorRT引擎 (NVIDIA优化)
    PADDLE_LITE = 2, // PaddleLite (ARM优化)
    RKNN = 3         // RKNN格式 (瑞芯微NPU专用)
};

/**
 * 推理任务类型
 */
enum class TaskType {
    OCR = 0,          // 文字OCR识别
    MATH_RECOGNITION = 1, // 数学列式识别
    STROKE_ORDER = 2,  // 笔顺分析
    WRITING_QUALITY = 3 // 书写质量评测
};

/**
 * 张量数据 (推理输入/输出)
 * 封装多维数组数据和形状信息
 */
struct Tensor {
    std::vector<float> data;          // 浮点数据
    std::vector<int64_t> shape;      // 维度形状 (如 [1, 3, 64, 64])
    std::string name;               // 张量名称

    /** 获取数据元素总数 */
    size_t size() const {
        size_t s = 1;
        for (auto d : shape) s *= d;
        return s;
    }
};

/**
 * 推理请求
 */
struct InferenceRequest {
    std::string request_id;          // 请求唯一ID
    TaskType task_type;              // 任务类型
    std::vector<Tensor> inputs;      // 输入张量列表
    int priority = 2;                // 优先级 (0=最高)
    int timeout_ms = 500;            // 超时时间
    std::string pen_id;              // 来源笔设备ID
    std::string student_id;          // 学生ID
    std::chrono::steady_clock::time_point submit_time; // 提交时间
};

/**
 * 推理结果

```



```

*/
struct InferenceResult {
    std::string request_id;
    bool success = false;
    std::string error_message;
    std::vector<Tensor> outputs;           // 输出张量列表
    float inference_time_ms = 0.0f;      // 推理耗时
    std::string model_version;           // 使用的模型版本
};

// ===== 推理后端抽象 =====

/**
 * 推理后端抽象基类
 * 所有推理引擎（ONNX Runtime、TensorRT等）的统一接口
 */
class InferenceBackend {
public:
    virtual ~InferenceBackend() = default;

    /** 加载模型文件 */
    virtual bool load_model(const std::string& model_path) = 0;

    /** 执行推理 */
    virtual InferenceResult infer(const InferenceRequest& request) = 0;

    /** 卸载模型释放资源 */
    virtual void unload() = 0;

    /** 获取后端名称 */
    virtual std::string name() const = 0;
};

/**
 * ONNX Runtime推理后端
 * 支持CPU/GPU/NPU多种执行提供者
 */
class OnnxRuntimeBackend : public InferenceBackend {
public:
    OnnxRuntimeBackend(DeviceType device) : device_(device), loaded_(false) {}

    bool load_model(const std::string& model_path) override {
        model_path_ = model_path;
        // 实际环境中:
        // Ort::SessionOptions options;
        // if (device_ == DeviceType::GPU_CUDA) {
        //     OrtCUDAProviderOptions cuda_opts;
        //     cuda_opts.device_id = 0;
        //     options.AppendExecutionProvider_CUDA(cuda_opts);
        // }
        // session_ = std::make_unique<Ort::Session>(env, model_path.c_str(), options);
        loaded_ = true;
        return true;
    }

    InferenceResult infer(const InferenceRequest& request) override {
        InferenceResult result;
    }

```

```

        result.request_id = request.request_id;

        if (!loaded_) {
            result.success = false;
            result.error_message = "模型未加载";
            return result;
        }

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

        // 执行ONNX Runtime推理
        // std::vector<Ort::Value> input_tensors;
        // for (const auto& input : request.inputs) {
        //     auto tensor = Ort::Value::CreateTensor<float>(
        //         memory_info, input.data.data(), input.size(),
        //         input.shape.data(), input.shape.size());
        //     input_tensors.push_back(std::move(tensor));
        // }
        // auto output_tensors = session_->Run(run_options, input_names, input_tensors,
output_names);

        // 模拟推理输出
        Tensor output;
        output.name = "output";
        output.shape = {1, 10};
        output.data.resize(10, 0.1f);
        result.outputs.push_back(output);
        result.success = true;

        auto end = std::chrono::steady_clock::now();
        result.inference_time_ms = std::chrono::duration<float, std::milli>(end -
start).count();
        return result;
    }

    void unload() override {
        loaded_ = false;
    }

    std::string name() const override { return "ONNXRuntime"; }

private:
    DeviceType device_;
    std::string model_path_;
    bool loaded_;
};

/**
 * TensorRT推理后端
 * NVIDIA GPU专用高性能推理引擎
 * 支持FP16/INT8量化推理, 显著降低推理延迟
 */
class TensorRTBackend : public InferenceBackend {
public:
    TensorRTBackend() : loaded_(false) {}

    bool load_model(const std::string& engine_path) override {

```

```

        engine_path_ = engine_path;
        // 实际环境中:
        // std::ifstream file(engine_path, std::ios::binary);
        // file.seekg(0, std::ios::end);
        // size_t size = file.tellg();
        // file.seekg(0, std::ios::beg);
        // std::vector<char> engine_data(size);
        // file.read(engine_data.data(), size);
        //
        // auto runtime = nvinfer1::createInferRuntime(logger);
        // engine_ = runtime->deserializeCudaEngine(engine_data.data(), size);
        // context_ = engine_->createExecutionContext();
        loaded_ = true;
        return true;
    }

    InferenceResult infer(const InferenceRequest& request) override {
        InferenceResult result;
        result.request_id = request.request_id;

        if (!loaded_) {
            result.success = false;
            result.error_message = "TensorRT引擎未加载";
            return result;
        }

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

        // 执行TensorRT推理
        // cudaMemcpyAsync(gpu_input, request.inputs[0].data.data(), ...);
        // context_->enqueueV2(buffers, stream, nullptr);
        // cudaMemcpyAsync(cpu_output, gpu_output, ...);
        // cudaStreamSynchronize(stream);

        Tensor output;
        output.name = "output";
        output.shape = {1, 10};
        output.data.resize(10, 0.1f);
        result.outputs.push_back(output);
        result.success = true;

        auto end = std::chrono::steady_clock::now();
        result.inference_time_ms = std::chrono::duration<float, std::milli>(end -
start).count();
        return result;
    }

    void unload() override {
        loaded_ = false;
    }

    std::string name() const override { return "TensorRT"; }

private:
    std::string engine_path_;
    bool loaded_;
};

```

```
// ===== 推理任务队列 =====

/**
 * 优先级推理任务队列
 * 按优先级和提交时间排序，高优先级任务优先处理
 * 课堂实时场景的推理请求拥有最高优先级
 */
class InferenceTaskQueue {
public:
    InferenceTaskQueue(size_t max_size = 1024) : max_size_(max_size) {}

    /**
     * 提交推理请求到队列
     * 如果队列已满，丢弃最低优先级的任务
     */
    bool enqueue(InferenceRequest request) {
        std::lock_guard<std::mutex> lock(mutex_);
        if (queue_.size() >= max_size_) {
            // 队列已满，检查是否可以替换低优先级任务
            if (!queue_.empty() && queue_.top().priority > request.priority) {
                queue_.pop(); // 移除最低优先级任务
            } else {
                return false; // 无法入队
            }
        }
        request.submit_time = std::chrono::steady_clock::now();
        queue_.push(std::move(request));
        cv_.notify_one();
        return true;
    }

    /**
     * 从队列获取最高优先级的任务
     * 如果队列为空则阻塞等待
     */
    bool dequeue(InferenceRequest& request, int timeout_ms = 100) {
        std::unique_lock<std::mutex> lock(mutex_);
        if (cv_.wait_for(lock, std::chrono::milliseconds(timeout_ms),
                        [this] { return !queue_.empty(); })) {
            request = queue_.top();
            queue_.pop();
            return true;
        }
        return false;
    }

    size_t size() const {
        std::lock_guard<std::mutex> lock(mutex_);
        return queue_.size();
    }

private:
    // 自定义比较器：优先级小的排前面，相同优先级按提交时间排序
    struct RequestCompare {
        bool operator()(const InferenceRequest& a, const InferenceRequest& b) {
            if (a.priority != b.priority) return a.priority > b.priority;

```

```

        return a.submit_time > b.submit_time;
    }
};

std::priority_queue<InferenceRequest, std::vector<InferenceRequest>, RequestCompare>
queue_;
mutable std::mutex mutex_;
std::condition_variable cv_;
size_t max_size_;
};

// ===== 推理引擎（核心类） =====

/**
 * 推理引擎
 * 管理多个推理后端，根据模型类型和硬件条件选择最优推理路径
 * 支持:
 * - 多模型并发推理（OCR、数学、笔顺各独立模型）
 * - 动态批处理（攒批提升GPU利用率）
 * - 推理结果缓存（相同输入直接返回缓存结果）
 * - 超时控制和优雅降级
 */
class InferenceEngine {
public:
    InferenceEngine(DeviceType device, const std::string& models_dir)
        : device_(device), models_dir_(models_dir), running_(false) {}

    /**
     * 初始化推理引擎
     * 检测硬件设备、创建推理后端、加载模型
     */
    bool initialize() {
        // 检测硬件加速设备
        detect_hardware();

        // 为每种任务类型创建专用推理后端
        backends_[TaskType::OCR] = create_backend("ocr");
        backends_[TaskType::MATH_RECOGNITION] = create_backend("math");
        backends_[TaskType::STROKE_ORDER] = create_backend("stroke_order");
        backends_[TaskType::WRITING_QUALITY] = create_backend("writing_quality");

        // 加载各模型
        for (auto& [type, backend] : backends_) {
            std::string model_file = get_model_path(type);
            if (!backend->load_model(model_file)) {
                return false;
            }
        }

        // 启动推理工作线程
        running_ = true;
        worker_thread_ = std::thread(&InferenceEngine::worker_loop, this);

        return true;
    }

    /**

```

```

    * 提交推理请求 (异步)
    */
std::string submit(InferenceRequest request) {
    task_queue_.enqueue(std::move(request));
    return request.request_id;
}

/**
 * 同步推理 (直接执行并返回结果)
 */
InferenceResult infer_sync(const InferenceRequest& request) {
    auto it = backends_.find(request.task_type);
    if (it == backends_.end()) {
        InferenceResult result;
        result.request_id = request.request_id;
        result.success = false;
        result.error_message = "不支持的任务类型";
        return result;
    }
    return it->second->infer(request);
}

/**
 * 关闭推理引擎
 */
void shutdown() {
    running_ = false;
    if (worker_thread_.joinable()) {
        worker_thread_.join();
    }
    for (auto& [type, backend] : backends_) {
        backend->unload();
    }
}

/**
 * 获取推理统计信息
 */
struct Stats {
    long total_requests = 0;
    long total_success = 0;
    long total_failures = 0;
    float avg_latency_ms = 0.0f;
    float p99_latency_ms = 0.0f;
    size_t queue_size = 0;
};

Stats get_stats() const {
    Stats stats;
    stats.total_requests = total_requests_.load();
    stats.total_success = total_success_.load();
    stats.total_failures = total_failures_.load();
    stats.queue_size = task_queue_.size();
    if (stats.total_success > 0) {
        stats.avg_latency_ms = total_latency_ms_.load() / stats.total_success;
    }
    return stats;
}

```

```

    }

private:
    void detect_hardware() {
        // 检测可用的硬件加速设备
        // 瑞芯微NPU: 检查/dev/mali0或/dev/rknpu
        // NVIDIA GPU: 检查CUDA Runtime
    }

    std::unique_ptr<InferenceBackend> create_backend(const std::string& model_name) {
        // 根据设备类型创建对应的推理后端
        if (device_ == DeviceType::GPU_CUDA) {
            return std::make_unique<TensorRTBackend>();
        }
        return std::make_unique<OnnxRuntimeBackend>(device_);
    }

    std::string get_model_path(TaskType type) {
        switch (type) {
            case TaskType::OCR: return models_dir_ + "/ocr/model.onnx";
            case TaskType::MATH_RECOGNITION: return models_dir_ + "/math/model.onnx";
            case TaskType::STROKE_ORDER: return models_dir_ + "/stroke/model.onnx";
            case TaskType::WRITING_QUALITY: return models_dir_ + "/quality/model.onnx";
        }
        return "";
    }

    /**
     * 推理工作线程主循环
     * 从任务队列取出请求, 执行推理, 存储结果
     */
    void worker_loop() {
        while (running_) {
            InferenceRequest request;
            if (task_queue_.dequeue(request, 100)) {
                total_requests++;

                auto result = infer_sync(request);

                if (result.success) {
                    total_success++;
                    total_latency_ms_ += result.inference_time_ms;
                } else {
                    total_failures++;
                }

                // 存储结果供查询
                std::lock_guard<std::mutex> lock(results_mutex_);
                results_[request.request_id] = result;
            }
        }
    }

    DeviceType device_;
    std::string models_dir_;
    std::atomic<bool> running_;
    std::thread worker_thread_;

```

```

    InferenceTaskQueue task_queue_;
    std::unordered_map<TaskType, std::unique_ptr<InferenceBackend>> backends_;
    std::unordered_map<std::string, InferenceResult> results_;
    std::mutex results_mutex_;

    // 统计计数器
    std::atomic<long> total_requests_{0};
    std::atomic<long> total_success_{0};
    std::atomic<long> total_failures_{0};
    std::atomic<float> total_latency_ms_{0.0f};
};

#endif // INFERENCE_ENGINE_H

```

inference/model_manager.cpp

```

/**
 * 自然写教室智能算力盒边缘计算软件 V1.0
 * 模型管理模块 - 模型加载、版本管理、量化压缩、云端同步
 *
 * 管理算力盒上部署的所有AI推理模型的生命周期
 * 支持模型热更新、A/B切换、云端版本同步
 * 模型文件AES-256加密存储，推理时内存解密加载
 */

#ifndef MODEL_MANAGER_H
#define MODEL_MANAGER_H

#include <string>
#include <vector>
#include <memory>
#include <mutex>
#include <atomic>
#include <unordered_map>
#include <chrono>
#include <functional>

// ===== 模型元信息 =====

/** 模型状态枚举 */
enum class ModelState {
    NOT_FOUND = 0,    // 未发现
    DOWNLOADING = 1,  // 下载中
    DECRYPTING = 2,    // 解密中
    LOADING = 3,       // 加载到设备中
    READY = 4,         // 就绪可用
    ACTIVE = 5,        // 当前使用中
    DEPRECATED = 6,    // 已弃用
    ERROR = 7         // 错误状态
};

/** 模型量化精度 */
enum class QuantizationType {
    FP32 = 0,          // 全精度浮点

```



```

    FP16 = 1,          // 半精度浮点
    INT8 = 2,          // 8位整型量化
    INT4 = 3           // 4位整型量化（极致压缩）
};

/** 模型元信息 */
struct ModelInfo {
    std::string name;           // 模型名称
    std::string version;        // 版本号（语义化版本）
    std::string format;         // 格式（onnx/trt/rknn）
    std::string file_path;      // 本地文件路径
    size_t file_size_bytes;     // 文件大小
    std::string sha256;         // 文件SHA-256校验和
    QuantizationType quantization; // 量化类型
    float accuracy;             // 测试集准确率
    float latency_ms;           // 平均推理延迟
    ModelState state;           // 当前状态
    std::string deployed_at;     // 部署时间
    std::string description;     // 模型描述
};

// ===== 模型加密管理 =====

/**
 * 模型文件加密/解密管理器
 * 安全设计：模型文件AES-256加密存储，推理时内存解密加载
 * 加密密钥通过安全芯片（TPM）或环境变量注入
 */
class ModelCryptoManager {
public:
    ModelCryptoManager() : key_loaded_(false) {}

    /**
     * 加载加密密钥
     * 优先从安全芯片读取，其次从环境变量
     */
    bool load_encryption_key() {
        // 尝试从TPM安全芯片读取密钥
        // if (tpm_available()) { key_ = tpm_read_key("model_key"); }

        // 后备方案：从环境变量读取
        const char* env_key = std::getenv("WRITECH_MODEL_KEY");
        if (env_key) {
            key_ = std::string(env_key);
            key_loaded_ = true;
            return true;
        }
        return false;
    }

    /**
     * 解密模型文件到内存
     * 不在磁盘上生成明文文件，仅在内存中解密
     */
    std::vector<uint8_t> decrypt_model(const std::string& encrypted_path) {
        std::vector<uint8_t> decrypted_data;
        if (!key_loaded_) return decrypted_data;
    }
};

```

```

        // 读取加密文件
        // AES-256-CBC解密
        // openssl EVP_DecryptInit_ex(ctx, EVP_aes_256_cbc(), NULL, key, iv);
        // EVP_DecryptUpdate(ctx, output, &out_len, input, in_len);
        // EVP_DecryptFinal_ex(ctx, output + out_len, &final_len);

        return decrypted_data;
    }

    /**
     * 加密模型文件
     * 新下载的模型文件加密后存储到本地Flash
     */
    bool encrypt_model(const std::vector<uint8_t>& data, const std::string& output_path)
    {
        if (!key_loaded_) return false;
        // AES-256-CBC加密并写入文件
        return true;
    }

    /**
     * 验证模型文件完整性
     * 计算SHA-256校验和并与元数据中的值比对
     */
    bool verify_integrity(const std::string& file_path, const std::string&
expected_sha256) {
        // 计算文件SHA-256
        // SHA256_CTX sha256;
        // SHA256_Init(&sha256);
        // while (read chunk) SHA256_Update(&sha256, chunk, len);
        // SHA256_Final(hash, &sha256);
        return true;
    }

private:
    std::string key_;
    bool key_loaded_;
};

// ===== 模型版本管理器 =====

/**
 * 模型版本管理器
 * 管理算力盒上所有AI模型的版本、加载、切换
 * 支持A/B分区切换实现热更新
 */
class ModelVersionManager {
public:
    ModelVersionManager(const std::string& models_dir)
        : models_dir_(models_dir) {}

    /**
     * 注册模型
     * 扫描模型目录，加载所有可用模型的元信息
     */
    bool register_model(const ModelInfo& info) {

```

```

        std::lock_guard<std::mutex> lock(mutex_);
        std::string key = info.name + "@" + info.version;
        models_[key] = info;
        return true;
    }

/**
 * 激活指定版本的模型
 * 将旧版本标记为deprecated, 新版本标记为active
 */
bool activate_version(const std::string& name, const std::string& version) {
    std::lock_guard<std::mutex> lock(mutex_);

    // 将当前活跃版本设为deprecated
    for (auto& pair : models_) {
        if (pair.second.name == name && pair.second.state == ModelState::ACTIVE) {
            pair.second.state = ModelState::DEPRECATED;
        }
    }

    // 激活新版本
    std::string key = name + "@" + version;
    auto it = models_.find(key);
    if (it != models_.end()) {
        it->second.state = ModelState::ACTIVE;
        return true;
    }
    return false;
}

/**
 * 获取当前活跃版本的模型信息
 */
ModelInfo get_active_model(const std::string& name) {
    std::lock_guard<std::mutex> lock(mutex_);
    for (const auto& pair : models_) {
        if (pair.second.name == name && pair.second.state == ModelState::ACTIVE) {
            return pair.second;
        }
    }
    return ModelInfo{};
}

/**
 * 获取所有模型状态列表
 */
std::vector<ModelInfo> get_all_models() {
    std::lock_guard<std::mutex> lock(mutex_);
    std::vector<ModelInfo> result;
    for (const auto& pair : models_) {
        result.push_back(pair.second);
    }
    return result;
}

/**
 * 清理已废弃的旧版本模型文件

```

```

    * 保留最近2个版本，删除更早的版本释放存储空间
    */
void cleanup_old_versions(const std::string& name, int keep_count = 2) {
    std::lock_guard<std::mutex> lock(mutex_);
    std::vector<std::string> deprecated_keys;

    for (const auto& pair : models_) {
        if (pair.second.name == name && pair.second.state == ModelState::DEPRECATED)
        {
            deprecated_keys.push_back(pair.first);
        }
    }

    // 按版本排序，保留最新的keep_count个
    if (static_cast<int>(deprecated_keys.size()) > keep_count) {
        for (int i = 0; i < static_cast<int>(deprecated_keys.size()) - keep_count;
i++) {
            // 删除模型文件并从注册表移除
            models_.erase(deprecated_keys[i]);
        }
    }
}

private:
    std::string models_dir_;
    std::unordered_map<std::string, ModelInfo> models_;
    std::mutex mutex_;
};

// ===== 云端模型同步器 =====

/**
 * 云端模型同步器
 * 定期检查云端是否有新版本模型，自动下载并部署
 * 通过HTTPS加密通道下载，下载后RSA签名校验
 */
class CloudModelSyncer {
public:
    CloudModelSyncer(const std::string& server_url, const std::string& device_id)
        : server_url_(server_url), device_id_(device_id) {}

    /**
     * 检查云端是否有模型更新
     * GET /api/v1/model/check-update?device_id=xxx&models=ocr@1.0,math@1.0
     */
    struct UpdateInfo {
        std::string model_name;
        std::string new_version;
        std::string download_url;
        size_t file_size;
        std::string sha256;
    };

    std::vector<UpdateInfo> check_updates(const std::vector<ModelInfo>& current_models)
    {
        std::vector<UpdateInfo> updates;
        // 向云端API发送当前模型版本列表，获取可更新版本

```

```

        // HTTPS请求: GET server_url_/api/v1/model/check-update
        return updates;
    }

    /**
     * 下载模型文件
     * HTTPS下载, 支持断点续传
     * 下载完成后进行SHA-256校验和RSA签名验证
     */
    bool download_model(const UpdateInfo& info, const std::string& save_path) {
        // HTTPS下载
        // 进度回调上报
        // SHA-256校验
        // RSA签名验证 (OTA安全: 升级包RSA签名+SHA-256校验, 防篡改)
        return true;
    }

    /**
     * 上报模型部署状态
     * POST /api/v1/model/deploy-status
     */
    void report_deploy_status(const std::string& model_name, const std::string& version,
                             bool success, const std::string& error = "") {
        // 向云端上报模型部署结果
    }

private:
    std::string server_url_;
    std::string device_id_;
};

// ===== OTA固件升级管理器 =====

/**
 * OTA固件升级管理器
 * 管理算力盒固件的远程升级
 * 采用A/B双分区方案, 升级失败自动回滚
 * 安全设计: 升级包RSA签名+SHA-256校验, 防篡改
 */
class OtaUpgradeManager {
public:
    enum class OtaState {
        IDLE,           // 空闲
        CHECKING,       // 检查更新中
        DOWNLOADING,    // 下载中
        VERIFYING,      // 校验中
        INSTALLING,     // 安装中
        REBOOTING,      // 重启中
        FAILED          // 失败
    };

    OtaUpgradeManager(const std::string& ota_url, const std::string& device_id)
        : ota_url_(ota_url), device_id_(device_id), state_(OtaState::IDLE),
          current_partition_("A"), download_progress_(0) {}

    /** 检查固件更新 */
    bool check_update() {

```

```

        state_ = OtaState::CHECKING;
        // GET ota_url_/api/v1/ota/check?device_id=xxx&version=xxx
        return false; // 返回是否有新版本
    }

    /** 下载固件升级包 */
    bool download_firmware(const std::string& download_url) {
        state_ = OtaState::DOWNLOADING;
        // HTTPS分块下载到非活跃分区
        // 支持断点续传
        return true;
    }

    /** 验证固件包完整性和签名 */
    bool verify_firmware(const std::string& firmware_path) {
        state_ = OtaState::VERIFYING;
        // SHA-256校验
        // RSA-2048签名验证
        return true;
    }

    /** 安装固件（写入非活跃分区） */
    bool install_firmware() {
        state_ = OtaState::INSTALLING;
        // 写入B分区（如当前运行A分区）
        // 设置下次启动从B分区引导
        return true;
    }

    /** 回滚到上一版本 */
    bool rollback() {
        // 切换回上一个分区
        std::string target = (current_partition_ == "A") ? "B" : "A";
        // 设置引导分区为target
        return true;
    }

    /** 获取当前OTA状态 */
    OtaState get_state() const { return state_; }
    int get_progress() const { return download_progress_; }
    std::string get_current_partition() const { return current_partition_; }

private:
    std::string ota_url_;
    std::string device_id_;
    OtaState state_;
    std::string current_partition_;
    int download_progress_;
};

// ===== 系统监控模块 =====

/**
 * 系统运行状态监控
 * 采集CPU、内存、GPU/NPU利用率、温度等硬件指标
 * 为云端监控告警和集群调度提供数据支撑
 */

```

```

class SystemMonitor {
public:
    struct SystemMetrics {
        float cpu_usage_percent;           // CPU使用率
        float memory_usage_percent;        // 内存使用率
        long memory_total_mb;              // 总内存
        long memory_used_mb;               // 已用内存
        float gpu_usage_percent;            // GPU/NPU利用率
        float gpu_memory_usage_mb;          // GPU显存使用
        float gpu_temperature_c;            // GPU温度
        float disk_usage_percent;           // 磁盘使用率
        float network_rx_mbps;              // 网络接收速率
        float network_tx_mbps;              // 网络发送速率
        long uptime_seconds;                // 系统运行时长
    };

    SystemMonitor() : running_(false) {}

    /** 启动监控采集线程 */
    void start(int interval_ms = 5000) {
        running_ = true;
        // 定时采集系统指标
    }

    /** 获取最新系统指标 */
    SystemMetrics get_metrics() {
        SystemMetrics metrics;
        metrics.cpu_usage_percent = read_cpu_usage();
        metrics.memory_usage_percent = read_memory_usage();
        metrics.gpu_usage_percent = read_gpu_usage();
        metrics.gpu_temperature_c = read_gpu_temperature();
        metrics.disk_usage_percent = read_disk_usage();
        return metrics;
    }

    void stop() { running_ = false; }

private:
    float read_cpu_usage() {
        // 读取 /proc/stat 计算CPU使用率
        return 0.0f;
    }

    float read_memory_usage() {
        // 读取 /proc/meminfo
        return 0.0f;
    }

    float read_gpu_usage() {
        // NVIDIA: nvidia-smi / NVML
        // 瑞芯微: /sys/class/devfreq/xxx
        return 0.0f;
    }

    float read_gpu_temperature() {
        // 读取GPU温度传感器
        return 0.0f;
    }
}

```

```

    }

    float read_disk_usage() {
        // statfs("/")
        return 0.0f;
    }

    std::atomic<bool> running_;
};

#endif // MODEL_MANAGER_H

```

inference/npu_scheduler.cpp

```

/**
 * 自然写教室智能算力盒边缘计算软件 V1.0
 * NPU/GPU硬件调度模块 - 硬件加速资源管理与任务分配
 *
 * 管理算力盒上的NPU/GPU计算资源
 * 支持多种硬件平台: NVIDIA GPU(CUDA)、瑞芯微NPU(RKNN)、通用GPU(OpenCL)
 * 根据任务类型和硬件负载动态选择最优推理路径
 */

#ifndef NPU_SCHEDULER_H
#define NPU_SCHEDULER_H

#include <string>
#include <vector>
#include <memory>
#include <mutex>
#include <atomic>
#include <chrono>
#include <queue>
#include <functional>
#include <unordered_map>
#include <thread>
#include <condition_variable>
#include <cstring>

// ===== 硬件设备抽象 =====

/** 硬件加速器类型 */
enum class AcceleratorType {
    CPU_ONLY = 0,          // 仅CPU (无加速器可用时的兜底方案)
    NVIDIA_GPU = 1,        // NVIDIA GPU (CUDA/TensorRT)
    ROCKCHIP_NPU = 2,      // 瑞芯微NPU (RKNN)
    AMLOGIC_NPU = 3,       // 晶晨NPU
    GENERIC_OPENCL = 4     // 通用OpenCL GPU
};

/** 硬件设备信息 */
struct AcceleratorDevice {
    AcceleratorType type;          // 加速器类型
    int device_id;                // 设备编号

```



```

    std::string name;           // 设备名称
    std::string driver_version; // 驱动版本
    size_t total_memory_mb;     // 总显存/内存(MB)
    size_t free_memory_mb;     // 可用显存/内存(MB)
    float compute_capability;   // 算力指标
    float current_utilization;  // 当前利用率(0-1)
    float temperature_celsius;  // 当前温度
    float max_temperature;      // 最高安全温度
    bool is_available;          // 是否可用
};

/** 推理任务资源需求 */
struct TaskResourceRequirement {
    size_t memory_mb;           // 需要的显存(MB)
    float estimated_time_ms;    // 预估推理时间
    bool requires_fp16;         // 是否需要FP16支持
    bool requires_int8;         // 是否需要INT8支持
    int preferred_device;       // 偏好设备ID (-1表示无偏好)
};

// ===== 硬件检测器 =====

/**
 * 硬件加速器检测器
 * 启动时扫描系统中可用的NPU/GPU设备
 * 自动匹配设备驱动和推理后端
 */
class HardwareDetector {
public:
    /**
     * 扫描系统中所有可用的加速器设备
     * 检测顺序: NVIDIA GPU → 瑞芯微NPU → 通用OpenCL → CPU
     */
    std::vector<AcceleratorDevice> detect_devices() {
        std::vector<AcceleratorDevice> devices;

        // 检测NVIDIA GPU
        if (detect_nvidia_gpu(devices)) {
            // 通过NVML库获取GPU信息
        }

        // 检测瑞芯微NPU
        if (detect_rockchip_npu(devices)) {
            // 通过sysfs获取NPU信息
        }

        // 如果没有加速器, 添加CPU作为兜底
        if (devices.empty()) {
            AcceleratorDevice cpu_dev;
            cpu_dev.type = AcceleratorType::CPU_ONLY;
            cpu_dev.device_id = 0;
            cpu_dev.name = "CPU";
            cpu_dev.total_memory_mb = get_system_memory_mb();
            cpu_dev.free_memory_mb = get_free_memory_mb();
            cpu_dev.is_available = true;
            devices.push_back(cpu_dev);
        }
    }
};

```

```

        return devices;
    }

private:
    bool detect_nvidia_gpu(std::vector<AcceleratorDevice>& devices) {
        // 检查 /dev/nvidia0 是否存在
        // 使用NVML API获取设备信息
        // nvmlInit();
        // nvmlDeviceGetCount(&count);
        // for (int i = 0; i < count; i++) {
        //     nvmlDeviceGetHandleByIndex(i, &device);
        //     nvmlDeviceGetName(device, name, sizeof(name));
        //     nvmlDeviceGetMemoryInfo(device, &mem);
        //     nvmlDeviceGetUtilizationRates(device, &util);
        //     nvmlDeviceGetTemperature(device, NVML_TEMPERATURE_GPU, &temp);
        // }
        return false;
    }

    bool detect_rockchip_npu(std::vector<AcceleratorDevice>& devices) {
        // 检查 /dev/rknpu 或 /sys/class/misc/rknpu 是否存在
        // 读取NPU硬件信息
        // cat /sys/kernel/debug/rknpu/load // NPU负载
        return false;
    }

    size_t get_system_memory_mb() {
        // 读取 /proc/meminfo
        return 4096; // 默认4GB
    }

    size_t get_free_memory_mb() {
        return 2048;
    }
};

// ===== 设备负载监控 =====

/**
 * 硬件设备负载实时监控
 * 定期采集GPU/NPU利用率、温度、显存使用等指标
 * 为调度策略提供实时数据支撑
 */
class DeviceLoadMonitor {
public:
    struct DeviceMetrics {
        int device_id;
        float utilization; // 利用率 (0-1)
        float memory_usage; // 显存使用率 (0-1)
        float temperature; // 温度(摄氏度)
        float power_watts; // 功耗(瓦)
        int inference_qps; // 当前推理QPS
        std::chrono::steady_clock::time_point timestamp;
    };

    DeviceLoadMonitor() : running_(false) {}
};

```

```

/** 启动监控（后台线程定期采集） */
void start(int interval_ms = 1000) {
    running_ = true;
    monitor_thread_ = std::thread([this, interval_ms]() {
        while (running_) {
            collect_metrics();
            std::this_thread::sleep_for(std::chrono::milliseconds(interval_ms));
        }
    });
}

/** 获取指定设备的最新指标 */
DeviceMetrics get_metrics(int device_id) {
    std::lock_guard<std::mutex> lock(mutex_);
    auto it = latest_metrics_.find(device_id);
    if (it != latest_metrics_.end()) {
        return it->second;
    }
    return DeviceMetrics{};
}

/** 获取所有设备指标 */
std::vector<DeviceMetrics> get_all_metrics() {
    std::lock_guard<std::mutex> lock(mutex_);
    std::vector<DeviceMetrics> result;
    for (const auto& pair : latest_metrics_) {
        result.push_back(pair.second);
    }
    return result;
}

void stop() {
    running_ = false;
    if (monitor_thread_.joinable()) {
        monitor_thread_.join();
    }
}

private:
    void collect_metrics() {
        std::lock_guard<std::mutex> lock(mutex_);
        // NVIDIA GPU: nvmlDeviceGetUtilizationRates + nvmlDeviceGetTemperature
        // 瑞芯微NPU: 读取 /sys/kernel/debug/rknpu/load
        // CPU: 读取 /proc/stat
    }

    std::unordered_map<int, DeviceMetrics> latest_metrics_;
    std::mutex mutex_;
    std::atomic<bool> running_;
    std::thread monitor_thread_;
};

// ===== 调度策略 =====

/**
 * 推理任务调度策略

```

```

    * 根据任务特征和设备负载选择最优的推理设备
    */
class SchedulingPolicy {
public:
    virtual ~SchedulingPolicy() = default;

    /** 选择最优设备执行推理任务 */
    virtual int select_device(const TaskResourceRequirement& requirement,
                             const std::vector<AcceleratorDevice>& devices,
                             const std::vector<DeviceLoadMonitor::DeviceMetrics>&
metrics) = 0;
};

/**
 * 最小负载调度策略
 * 优先选择当前利用率最低的设备
 */
class MinLoadPolicy : public SchedulingPolicy {
public:
    int select_device(const TaskResourceRequirement& requirement,
                     const std::vector<AcceleratorDevice>& devices,
                     const std::vector<DeviceLoadMonitor::DeviceMetrics>& metrics)
override {
    int best_device = 0;
    float min_load = 1.0f;

    for (size_t i = 0; i < devices.size(); i++) {
        if (!devices[i].is_available) continue;
        if (devices[i].free_memory_mb < requirement.memory_mb) continue;

        float load = (i < metrics.size()) ? metrics[i].utilization : 0.0f;
        if (load < min_load) {
            min_load = load;
            best_device = static_cast<int>(i);
        }
    }
    return best_device;
}
};

/**
 * 温度感知调度策略
 * 除了负载外还考虑设备温度，防止过热降频
 */
class ThermalAwarePolicy : public SchedulingPolicy {
public:
    ThermalAwarePolicy(float temp_threshold = 80.0f) : temp_threshold_(temp_threshold)
{}

    int select_device(const TaskResourceRequirement& requirement,
                     const std::vector<AcceleratorDevice>& devices,
                     const std::vector<DeviceLoadMonitor::DeviceMetrics>& metrics)
override {
    int best_device = 0;
    float best_score = -1.0f;

    for (size_t i = 0; i < devices.size(); i++) {

```

```

        if (!devices[i].is_available) continue;
        if (devices[i].free_memory_mb < requirement.memory_mb) continue;

        float load = (i < metrics.size()) ? metrics[i].utilization : 0.0f;
        float temp = (i < metrics.size()) ? metrics[i].temperature : 0.0f;

        // 综合评分: 负载权重0.6 + 温度权重0.4
        float load_score = 1.0f - load;
        float temp_score = (temp < temp_threshold_) ? 1.0f : (1.0f - (temp -
temp_threshold_) / 20.0f);
        float score = load_score * 0.6f + temp_score * 0.4f;

        if (score > best_score) {
            best_score = score;
            best_device = static_cast<int>(i);
        }
    }
    return best_device;
}

private:
    float temp_threshold_;
};

// ===== NPU调度器 (核心) =====

/**
 * NPU/GPU硬件调度器
 * 管理推理任务到硬件设备的分配调度
 * 核心功能:
 * 1. 硬件资源池化管理
 * 2. 基于负载和温度的智能调度
 * 3. 设备故障自动切换
 * 4. 推理性能指标采集
 */
class NpuScheduler {
public:
    NpuScheduler() : initialized_(false) {}

    /**
     * 初始化调度器
     * 检测硬件设备, 启动负载监控, 设置调度策略
     */
    bool initialize() {
        // 检测可用硬件加速器
        HardwareDetector detector;
        devices_ = detector.detect_devices();

        if (devices_.empty()) {
            return false;
        }

        // 启动设备负载监控
        load_monitor_.start(1000);

        // 设置调度策略 (默认温度感知策略)
        policy_ = std::make_unique<ThermalAwarePolicy>(80.0f);

```

```

        initialized_ = true;
        return true;
    }

    /**
     * 为推理任务分配最优设备
     */
    int schedule_task(const TaskResourceRequirement& requirement) {
        if (!initialized_) return 0;

        auto metrics = load_monitor_.get_all_metrics();
        return policy_>select_device(requirement, devices_, metrics);
    }

    /**
     * 获取所有设备状态
     */
    std::vector<AcceleratorDevice> get_device_status() {
        // 更新设备实时状态
        auto metrics = load_monitor_.get_all_metrics();
        for (auto& dev : devices_) {
            for (const auto& m : metrics) {
                if (m.device_id == dev.device_id) {
                    dev.current_utilization = m.utilization;
                    dev.temperature_celsius = m.temperature;
                }
            }
        }
        return devices_;
    }

    /** 获取调度统计信息 */
    struct SchedulerStats {
        long total_tasks_scheduled;
        long total_tasks_completed;
        long total_tasks_failed;
        float avg_inference_ms;
        float gpu_avg_utilization;
        float gpu_temperature;
        int active_devices;
    };

    SchedulerStats get_stats() {
        SchedulerStats stats;
        stats.total_tasks_scheduled = tasks_scheduled_.load();
        stats.total_tasks_completed = tasks_completed_.load();
        stats.total_tasks_failed = tasks_failed_.load();
        stats.active_devices = static_cast<int>(devices_.size());

        auto metrics = load_monitor_.get_all_metrics();
        if (!metrics.empty()) {
            float total_util = 0;
            for (const auto& m : metrics) total_util += m.utilization;
            stats.gpu_avg_utilization = total_util / metrics.size();
            stats.gpu_temperature = metrics[0].temperature;
        }
    }

```

```

        return stats;
    }

    void shutdown() {
        load_monitor_.stop();
        initialized_ = false;
    }

private:
    std::vector<AcceleratorDevice> devices_;
    DeviceLoadMonitor load_monitor_;
    std::unique_ptr<SchedulingPolicy> policy_;
    bool initialized_;

    std::atomic<long> tasks_scheduled_{0};
    std::atomic<long> tasks_completed_{0};
    std::atomic<long> tasks_failed_{0};
};

// ===== 配置管理 =====

/**
 * 算力盒配置管理（边缘设备专用）
 * 从JSON配置文件和环境变量加载配置
 * 支持运行时配置热更新（通过MQTT远程指令）
 */
struct EdgeBoxConfiguration {
    // 推理配置
    int max_concurrent_inferences = 4;    // 最大并发推理数
    int inference_queue_size = 256;       // 推理队列大小
    int default_timeout_ms = 500;         // 默认推理超时

    // NPU/GPU配置
    float gpu_memory_fraction = 0.8f;     // GPU显存使用比例上限
    float thermal_throttle_temp = 80.0f;  // 温度降频阈值
    bool enable_fp16 = true;               // 启用FP16推理
    bool enable_int8 = false;              // 启用INT8量化

    // 网络配置
    std::string grpc_listen = "0.0.0.0:50052";
    std::string mqtt_broker = "ssl://mqtt.writech.com:8883";
    bool enable_mtls = true;

    // 存储配置
    std::string models_dir = "/opt/models";
    std::string cache_dir = "/var/lib/writech/cache";
    int offline_cache_max_mb = 256;

    // 集群配置
    bool enable_cluster = true;
    std::string cluster_discovery = "mdns";
};

#endif // NPU_SCHEDULER_H

```

preprocessing/

preprocessing/stroke_preprocessor.cpp

```
/**
 * 自然写教室智能算力盒边缘计算软件 V1.0
 * 笔迹预处理模块 - 笔迹坐标数据预处理管道
 *
 * 对网关转发的原始笔迹坐标进行预处理：
 * 去噪滤波、坐标归一化、笔画分割、特征提取
 * 预处理结果作为NPU/GPU推理的标准化输入
 */

#ifndef STROKE_PREPROCESSOR_H
#define STROKE_PREPROCESSOR_H

#include <vector>
#include <cmath>
#include <algorithm>
#include <numeric>
#include <cstring>

// ===== 基础数据结构 =====

/** 原始笔迹坐标点（来自网关gRPC数据流） */
struct RawPoint {
    float x;           // X坐标（点阵单位，约300DPI）
    float y;           // Y坐标
    float pressure;     // 压力值（0.0-1.0）
    uint32_t timestamp; // 采集时间戳（毫秒）
    bool pen_up;        // 抬笔标记
};

/** 归一化后的坐标点 */
struct NormalizedPoint {
    float x;           // 归一化X（0.0-1.0）
    float y;           // 归一化Y（0.0-1.0）
    float pressure;     // 压力值（0.0-1.0）
};

/** 笔画数据 */
struct Stroke {
    std::vector<NormalizedPoint> points; // 归一化坐标点序列
    int stroke_index;                    // 笔画序号
    float length;                        // 笔画路径长度
    int duration_ms;                     // 书写耗时（毫秒）
};

/** 预处理输出（用于NPU推理输入） */
struct PreprocessedData {
    std::vector<float> image;           // 渲染后的灰度图像（H*W）
    int image_width;                    // 图像宽度
    int image_height;                   // 图像高度
    std::vector<Stroke> strokes;         // 分割后的笔画列表
};
```



```

    int total_points;           // 总坐标点数
    int stroke_count;          // 笔画数量
};

// ===== 去噪滤波器 =====

/**
 * 笔迹去噪滤波器
 * 消除点阵笔采集过程中的抖动噪声和异常跳跃点
 * 多级滤波策略: 异常点剔除 → 中值滤波 → 移动平均平滑
 */
class StrokeNoiseFilter {
public:
    /**
     * 构造函数
     * max_jump: 最大允许跳跃距离 (超过则视为异常点)
     * window_size: 滤波窗口大小 (奇数)
     */
    StrokeNoiseFilter(float max_jump = 50.0f, int window_size = 3)
        : max_jump_(max_jump), window_size_(window_size) {}

    /**
     * 剔除异常跳跃点
     * 点阵笔摄像头短暂遮挡会导致坐标突变, 需要过滤
     */
    std::vector<RawPoint> remove_outliers(const std::vector<RawPoint>& points) {
        if (points.size() < 3) return points;

        std::vector<RawPoint> result;
        result.push_back(points[0]);

        for (size_t i = 1; i < points.size(); i++) {
            float dx = points[i].x - points[i-1].x;
            float dy = points[i].y - points[i-1].y;
            float dist = std::sqrt(dx * dx + dy * dy);

            // 跳跃距离在合理范围内才保留该点
            if (dist <= max_jump_) {
                result.push_back(points[i]);
            }
        }
        return result;
    }

    /**
     * 中值滤波去噪
     * 对X和Y坐标分别进行一维中值滤波
     * 有效消除脉冲噪声同时保留笔画转折特征
     */
    std::vector<RawPoint> median_filter(const std::vector<RawPoint>& points) {
        int n = static_cast<int>(points.size());
        if (n < window_size_) return points;

        int half = window_size_ / 2;
        std::vector<RawPoint> result(n);

        for (int i = 0; i < n; i++) {

```

```

        // 收集窗口内的X和Y值
        std::vector<float> wx, wy;
        for (int j = std::max(0, i - half); j <= std::min(n - 1, i + half); j++) {
            wx.push_back(points[j].x);
            wy.push_back(points[j].y);
        }
        // 排序取中值
        std::sort(wx.begin(), wx.end());
        std::sort(wy.begin(), wy.end());

        result[i] = points[i];
        result[i].x = wx[wx.size() / 2];
        result[i].y = wy[wy.size() / 2];
    }
    return result;
}

```

```

/**
 * 移动平均平滑
 * 进一步减少微小抖动，使笔画更流畅
 */
std::vector<RawPoint> moving_average(const std::vector<RawPoint>& points) {
    int n = static_cast<int>(points.size());
    if (n < 3) return points;

    std::vector<RawPoint> result(n);
    int half = window_size_ / 2;

    for (int i = 0; i < n; i++) {
        float sum_x = 0, sum_y = 0;
        int count = 0;
        for (int j = std::max(0, i - half); j <= std::min(n - 1, i + half); j++) {
            sum_x += points[j].x;
            sum_y += points[j].y;
            count++;
        }
        result[i] = points[i];
        result[i].x = sum_x / count;
        result[i].y = sum_y / count;
    }
    return result;
}

```

```

/** 执行完整去噪流程 */
std::vector<RawPoint> apply(const std::vector<RawPoint>& points) {
    auto step1 = remove_outliers(points);
    auto step2 = median_filter(step1);
    auto step3 = moving_average(step2);
    return step3;
}

```

```

private:
    float max_jump_;
    int window_size_;
};

```

```

// ===== 坐标归一化器 =====

```

```

/**
 * 坐标归一化器
 * 将不同纸张尺寸和分辨率的原始坐标统一归一化到[0,1]范围
 * 保持宽高比以避免笔迹变形
 */
class CoordinateNormalizer {
public:
    CoordinateNormalizer(bool preserve_aspect = true) :
        preserve_aspect_(preserve_aspect) {}

    /**
     * Min-Max归一化, 映射到[0,1]范围
     */
    std::vector<NormalizedPoint> normalize(const std::vector<RawPoint>& points) {
        if (points.empty()) return {};

        // 计算坐标范围
        float min_x = points[0].x, max_x = points[0].x;
        float min_y = points[0].y, max_y = points[0].y;
        for (const auto& p : points) {
            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 range_x = max_x - min_x;
        float range_y = max_y - min_y;

        // 保持宽高比时使用统一的缩放因子
        float scale = 1.0f;
        if (preserve_aspect_) {
            scale = std::max(range_x, range_y);
            if (scale < 1e-6f) scale = 1.0f;
        }

        std::vector<NormalizedPoint> result;
        result.reserve(points.size());

        for (const auto& p : points) {
            NormalizedPoint np;
            if (preserve_aspect_) {
                np.x = (p.x - min_x) / scale;
                np.y = (p.y - min_y) / scale;
            } else {
                np.x = (range_x > 1e-6f) ? (p.x - min_x) / range_x : 0.5f;
                np.y = (range_y > 1e-6f) ? (p.y - min_y) / range_y : 0.5f;
            }
            np.pressure = p.pressure;
            result.push_back(np);
        }
        return result;
    }

private:
    bool preserve_aspect_;

```

```

};

// ===== 笔画分割器 =====

/**
 * 笔画分割器
 * 根据抬笔事件和时间间隔将连续坐标流分割为独立笔画
 */
class StrokeSegmenter {
public:
    StrokeSegmenter(int time_threshold_ms = 200, int min_points = 3)
        : time_threshold_(time_threshold_ms), min_points_(min_points) {}

    /**
     * 将原始点序列分割为笔画列表
     */
    std::vector<std::vector<RawPoint>> segment(const std::vector<RawPoint>& points) {
        if (points.empty()) return {};

        std::vector<std::vector<RawPoint>> strokes;
        std::vector<RawPoint> current;
        current.push_back(points[0]);

        for (size_t i = 1; i < points.size(); i++) {
            bool is_break = points[i].pen_up;
            int time_gap = static_cast<int>(points[i].timestamp - points[i-1].timestamp);

            if ((is_break || time_gap > time_threshold_) &&
                static_cast<int>(current.size()) >= min_points_) {
                strokes.push_back(current);
                current.clear();
            }
            if (!points[i].pen_up) {
                current.push_back(points[i]);
            }
        }
        if (static_cast<int>(current.size()) >= min_points_) {
            strokes.push_back(current);
        }
        return strokes;
    }

private:
    int time_threshold_;
    int min_points_;
};

// ===== 图像渲染器 =====

/**
 * 笔迹图像渲染器
 * 将归一化坐标渲染为灰度图像作为CNN模型输入
 * 使用Bresenham直线算法连接相邻坐标点
 */
class StrokeImageRenderer {
public:

```

```

StrokeImageRenderer(int width = 64, int height = 64)
    : width_(width), height_(height) {}

/**
 * 将坐标序列渲染为灰度图像
 * 输出一维浮点数组，值域[0,1]，1表示笔迹
 */
std::vector<float> render(const std::vector<NormalizedPoint>& points) {
    std::vector<float> image(width_ * height_, 0.0f);

    for (size_t i = 1; i < points.size(); i++) {
        int x0 = static_cast<int>(points[i-1].x * (width_ - 1));
        int y0 = static_cast<int>(points[i-1].y * (height_ - 1));
        int x1 = static_cast<int>(points[i].x * (width_ - 1));
        int y1 = static_cast<int>(points[i].y * (height_ - 1));

        // 裁剪到图像范围
        x0 = std::clamp(x0, 0, width_ - 1);
        y0 = std::clamp(y0, 0, height_ - 1);
        x1 = std::clamp(x1, 0, width_ - 1);
        y1 = std::clamp(y1, 0, height_ - 1);

        float pressure = (points[i-1].pressure + points[i].pressure) * 0.5f;

        // Bresenham直线算法
        draw_line(image, x0, y0, x1, y1, pressure);
    }
    return image;
}

private:
    void draw_line(std::vector<float>& image, int x0, int y0, int x1, int y1, float
value) {
        int dx = std::abs(x1 - x0);
        int dy = std::abs(y1 - y0);
        int sx = (x0 < x1) ? 1 : -1;
        int sy = (y0 < y1) ? 1 : -1;
        int err = dx - dy;

        while (true) {
            int idx = y0 * width_ + x0;
            if (idx >= 0 && idx < width_ * height_) {
                image[idx] = std::max(image[idx], value);
            }
            if (x0 == x1 && y0 == y1) break;
            int e2 = 2 * err;
            if (e2 > -dy) { err -= dy; x0 += sx; }
            if (e2 < dx) { err += dx; y0 += sy; }
        }
    }

    int width_;
    int height_;
};

// ===== 预处理管道（整合） =====

```

```

/**
 * 笔迹预处理管道
 * 整合去噪、归一化、分割、渲染的完整处理流程
 * 输入原始坐标点序列，输出标准化的推理输入数据
 */
class StrokePreprocessor {
public:
    StrokePreprocessor(int image_size = 64)
        : noise_filter_(50.0f, 3),
          normalizer_(true),
          segmenter_(200, 3),
          renderer_(image_size, image_size),
          image_size_(image_size) {}

    /**
     * 执行完整预处理管道
     * 流程：原始坐标 → 去噪 → 归一化 → 笔画分割 → 图像渲染
     */
    PreprocessedData process(const std::vector<RawPoint>& raw_points) {
        PreprocessedData result;

        // 步骤1：去噪滤波
        auto denoised = noise_filter_.apply(raw_points);

        // 步骤2：坐标归一化
        auto normalized = normalizer_.normalize(denoised);

        // 步骤3：笔画分割
        auto stroke_groups = segmenter_.segment(denoised);

        // 构建笔画数据
        for (int i = 0; i < static_cast<int>(stroke_groups.size()); i++) {
            Stroke stroke;
            stroke.stroke_index = i;
            auto norm_group = normalizer_.normalize(stroke_groups[i]);
            stroke.points = norm_group;
            stroke.length = calc_path_length(norm_group);
            if (stroke_groups[i].size() >= 2) {
                stroke.duration_ms = static_cast<int>(
                    stroke_groups[i].back().timestamp -
                    stroke_groups[i].front().timestamp);
            }
            result.strokes.push_back(stroke);
        }

        // 步骤4：渲染为灰度图像
        result.image = renderer_.render(normalized);
        result.image_width = image_size_;
        result.image_height = image_size_;
        result.total_points = static_cast<int>(denoised.size());
        result.stroke_count = static_cast<int>(result.strokes.size());

        return result;
    }

private:
    float calc_path_length(const std::vector<NormalizedPoint>& points) {

```

```
float total = 0.0f;
for (size_t i = 1; i < points.size(); i++) {
    float dx = points[i].x - points[i-1].x;
    float dy = points[i].y - points[i-1].y;
    total += std::sqrt(dx * dx + dy * dy);
}
return total;
}

StrokeNoiseFilter noise_filter_;
CoordinateNormalizer normalizer_;
StrokeSegmenter segmenter_;
StrokeImageRenderer renderer_;
int image_size_;
};

#endif // STROKE_PREPROCESSOR_H
```