

自然写互动课堂应用开发SDK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

```
11-writech-sdk/  
├── android/  
│   ├── CloudClient.java  
│   ├── GatewaySDK.java  
│   ├── OCREngine.java  
│   ├── PenManager.java  
│   ├── StrokeCanvas.java  
│   └── WritechSDK.java  
├── core/  
│   ├── ble_protocol.c  
│   ├── coordinate_transform.c  
│   └── stroke_smoother.c  
└── model/  
    ├── PenDevice.java  
    ├── RecognitionResult.java  
    └── StrokePath.java
```

源程序文件清单

android/

android/CloudClient.java

```
/*  
 * 自然写互动课堂应用开发SDK软件 V1.0  
 * CloudClient - 云平台API客户端  
 *  
 * 功能说明:
```

- * 1. 封装云平台REST API调用（用户认证、作业、笔迹等）
- * 2. JWT + Refresh Token 双令牌自动刷新机制
- * 3. 请求签名与加密（防篡改、防重放）
- * 4. 请求重试与超时控制
- * 5. 笔迹数据批量上传（分片压缩）
- * 6. 文件上传/下载（OSS预签名URL）
- */

```
package com.writech.sdk.android;
```

```
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.HttpURLConnection;
import java.net.URL;
import java.nio.charset.StandardCharsets;
import java.security.MessageDigest;
import java.util.Map;
import java.util.TreeMap;
import java.util.zip.GZIPOutputStream;
import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;
```

```
/**
```

- * 云平台API客户端
- * 提供统一的HTTP调用封装，支持JWT认证和请求签名
- */

```
public class CloudClient {
```

```
    private static final String TAG = "WritechCloudClient";
```

```
    /* 默认请求超时（毫秒） */
```

```
    private static final int DEFAULT_CONNECT_TIMEOUT = 10000;
```

```
    private static final int DEFAULT_READ_TIMEOUT = 30000;
```

```
    /* 最大重试次数 */
```

```
    private static final int MAX_RETRY_COUNT = 3;
```

```
    /* 笔迹批量上传分片大小（字节） */
```

```
    private static final int STROKE_CHUNK_SIZE = 64 * 1024;
```

```
    /* ===== 认证令牌管理 ===== */
```

```
    private String mBaseUrl;
```

```
    /* 云平台API基础URL */
```

```
    private String mAccessToken;
```

```
    /* JWT访问令牌 */
```

```
    private String mRefreshToken;
```

```
    /* 刷新令牌 */
```

```
    private long mTokenExpireTime;
```

```
    /* 令牌过期时间（毫秒时间戳） */
```

```
    private String mAppKey;
```

```
    /* 应用密钥（用于请求签名） */
```

```
    private String mAppSecret;
```

```
    /* 应用签名密钥 */
```

```
    /* 令牌刷新回调 */
```

```
    private TokenRefreshCallback mTokenCallback;
```

```
    /** 令牌刷新回调接口 */
```

```
    public interface TokenRefreshCallback {
```

```
        void onTokenRefreshed(String newAccessToken, String newRefreshToken);
```

```

        void onTokenRefreshFailed(int errorCode, String message);
    }

    /* ===== 构造与初始化 ===== */

    /**
     * 创建云平台API客户端
     * @param baseUrl 云平台API基础地址 (如 https://api.writech.com)
     * @param appKey SDK应用标识
     * @param appSecret SDK应用密钥
     */
    public CloudClient(String baseUrl, String appKey, String appSecret) {
        mBaseUrl = baseUrl;
        mAppKey = appKey;
        mAppSecret = appSecret;
    }

    /** 设置认证令牌 */
    public void setTokens(String accessToken, String refreshToken, long expireTime) {
        mAccessToken = accessToken;
        mRefreshToken = refreshToken;
        mTokenExpireTime = expireTime;
    }

    /** 设置令牌刷新回调 */
    public void setTokenRefreshCallback(TokenRefreshCallback callback) {
        mTokenCallback = callback;
    }

    /* ===== 用户认证API ===== */

    /**
     * 用户登录 (账号密码方式)
     * @param username 用户名
     * @param password 密码 (明文, SDK内部做SHA256后传输)
     * @return JSON响应字符串, 包含accessToken和refreshToken
     */
    public String login(String username, String password) throws IOException {
        String passwordHash = sha256(password);
        String body = "{\"username\":\"" + username + "\", \"password\":\"" +
passwordHash + "\"}";
        return postJson("/api/v1/auth/login", body);
    }

    /**
     * 刷新访问令牌
     * 在accessToken过期前自动调用, 使用refreshToken获取新令牌
     */
    public boolean refreshAccessToken() {
        try {
            String body = "{\"refreshToken\":\"" + mRefreshToken + "\"}";
            String response = postJsonNoAuth("/api/v1/auth/refresh", body);

            /* 解析响应中的新令牌 */
            String newAccess = extractJsonValue(response, "accessToken");
            String newRefresh = extractJsonValue(response, "refreshToken");

```

```

        if (newAccess != null && newRefresh != null) {
            mAccessToken = newAccess;
            mRefreshToken = newRefresh;
            /* 默认过期时间30分钟 */
            mTokenExpireTime = System.currentTimeMillis() + 30 * 60 * 1000;

            if (mTokenCallback != null) {
                mTokenCallback.onTokenRefreshed(newAccess, newRefresh);
            }
            return true;
        }
    } catch (IOException e) {
        if (mTokenCallback != null) {
            mTokenCallback.onTokenRefreshFailed(-1, e.getMessage());
        }
    }
    return false;
}

/* ===== 作业管理API ===== */

/** 获取作业列表 */
public String getAssignments(String classId, int page, int pageSize) throws
IOException {
    String params = "classId=" + classId + "&page=" + page + "&pageSize=" +
    pageSize;
    return get("/api/v1/assignments?" + params);
}

/** 获取作业详情 */
public String getAssignmentDetail(String assignmentId) throws IOException {
    return get("/api/v1/assignments/" + assignmentId);
}

/** 提交作业 */
public String submitAssignment(String assignmentId, String studentId,
    String answerJson) throws IOException {
    String body = "{\"assignmentId\":\"" + assignmentId
        + "\",\"studentId\":\"" + studentId
        + "\",\"answers\":\"" + answerJson + "\"}";
    return postJson("/api/v1/assignments/submit", body);
}

/* ===== 笔迹数据上传API ===== */

/**
 * 上传笔迹数据 (单次)
 * @param studentId 学生ID
 * @param pageId 页面ID
 * @param strokeJson 笔迹JSON数据
 */
public String uploadStroke(String studentId, String pageId,
    String strokeJson) throws IOException {
    String body = "{\"studentId\":\"" + studentId
        + "\",\"pageId\":\"" + pageId
        + "\",\"strokes\":\"" + strokeJson + "\"}";
    return postJson("/api/v1/strokes/upload", body);
}

```

```

    }

    /**
     * 批量上传笔迹数据（大数据量分片压缩）
     * 将笔迹数据按CHUNK_SIZE分片，GZIP压缩后逐片上传
     *
     * @param studentId 学生ID
     * @param strokeBytes 笔迹二进制数据
     * @return 上传成功的分片数
     */
    public int uploadStrokeBatch(String studentId, byte[] strokeBytes) throws
IOException {
        /* GZIP压缩原始数据 */
        byte[] compressed = gzipCompress(strokeBytes);

        /* 计算分片数 */
        int totalChunks = (compressed.length + STROKE_CHUNK_SIZE - 1) /
STROKE_CHUNK_SIZE;
        int uploadedChunks = 0;

        String uploadId = generateUploadId();

        for (int i = 0; i < totalChunks; i++) {
            int offset = i * STROKE_CHUNK_SIZE;
            int length = Math.min(STROKE_CHUNK_SIZE, compressed.length - offset);
            byte[] chunk = new byte[length];
            System.arraycopy(compressed, offset, chunk, 0, length);

            /* 上传分片 */
            String url = mBaseUrl + "/api/v1/strokes/upload-chunk";
            String boundary = "----WritechBoundary" + System.currentTimeMillis();

            HttpURLConnection conn = createConnection(url, "POST");
            conn.setRequestProperty("Content-Type", "multipart/form-data; boundary=" +
boundary);
            addAuthHeaders(conn);

            OutputStream os = conn.getOutputStream();
            /* 写入表单字段 */
            writeMultipartField(os, boundary, "uploadId", uploadId);
            writeMultipartField(os, boundary, "studentId", studentId);
            writeMultipartField(os, boundary, "chunkIndex", String.valueOf(i));
            writeMultipartField(os, boundary, "totalChunks",
String.valueOf(totalChunks));
            /* 写入二进制数据块 */
            writeMultipartFile(os, boundary, "data", "chunk_" + i + ".gz", chunk);
            os.write(("--" + boundary + "--\r\n").getBytes(StandardCharsets.UTF_8));
            os.flush();

            int responseCode = conn.getResponseCode();
            conn.disconnect();

            if (responseCode == 200) {
                uploadedChunks++;
            } else {
                break;
            }
        }
    }

```

```

    }

    return uploadedChunks;
}

/* ===== Multipart POST (静态方法供OCREngine调用) ===== */

/**
 * 发送Multipart POST请求
 * @param url        完整URL
 * @param token      Bearer令牌
 * @param imageData  图像二进制数据
 * @param strokeData 笔迹数据
 * @param targetChar 目标字符
 * @param timeoutMs  超时毫秒数
 * @return 响应JSON字符串
 */
public static String postMultipart(String url, String token, byte[] imageData,
                                   byte[] strokeData, String targetChar,
                                   int timeoutMs) throws IOException {
    HttpURLConnection conn = (HttpURLConnection) new URL(url).openConnection();
    conn.setRequestMethod("POST");
    conn.setConnectTimeout(timeoutMs);
    conn.setReadTimeout(timeoutMs);
    conn.setDoOutput(true);

    if (token != null) {
        conn.setRequestProperty("Authorization", "Bearer " + token);
    }

    String boundary = "----WritechBound" + System.nanoTime();
    conn.setRequestProperty("Content-Type", "multipart/form-data; boundary=" +
boundary);

    OutputStream os = conn.getOutputStream();
    if (imageData != null) {
        writeMultipartFile(os, boundary, "image", "stroke.png", imageData);
    }
    if (strokeData != null) {
        writeMultipartFile(os, boundary, "strokes", "strokes.bin", strokeData);
    }
    if (targetChar != null) {
        writeMultipartField(os, boundary, "targetChar", targetChar);
    }
    os.write(("--" + boundary + "--\r\n").getBytes(StandardCharsets.UTF_8));
    os.flush();

    String response = readResponse(conn);
    conn.disconnect();
    return response;
}

/* ===== HTTP基础方法 ===== */

/** GET请求 */
public String get(String path) throws IOException {
    return executeWithRetry("GET", path, null);
}

```

```

}

/** POST JSON请求 (带认证) */
public String postJson(String path, String jsonBody) throws IOException {
    return executeWithRetry("POST", path, jsonBody);
}

/** POST JSON请求 (无认证, 用于登录/刷新令牌) */
private String postJsonNoAuth(String path, String body) throws IOException {
    String url = mBaseUrl + path;
    HttpURLConnection conn = createConnection(url, "POST");
    conn.setRequestProperty("Content-Type", "application/json; charset=UTF-8");
    conn.setDoOutput(true);

    OutputStream os = conn.getOutputStream();
    os.write(body.getBytes(StandardCharsets.UTF_8));
    os.flush();

    String response = readResponse(conn);
    conn.disconnect();
    return response;
}

/** 带重试和令牌自动刷新的HTTP请求执行 */
private String executeWithRetry(String method, String path, String body) throws
IOException {
    int retryCount = 0;
    IOException lastException = null;

    while (retryCount < MAX_RETRY_COUNT) {
        try {
            /* 检查令牌是否即将过期 (提前5分钟刷新) */
            if (mTokenExpireTime > 0 &&
                System.currentTimeMillis() > mTokenExpireTime - 5 * 60 * 1000) {
                refreshAccessToken();
            }

            String url = mBaseUrl + path;
            HttpURLConnection conn = createConnection(url, method);
            addAuthHeaders(conn);

            if ("POST".equals(method) && body != null) {
                conn.setRequestProperty("Content-Type", "application/json;
charset=UTF-8");
                conn.setDoOutput(true);
                OutputStream os = conn.getOutputStream();
                os.write(body.getBytes(StandardCharsets.UTF_8));
                os.flush();
            }

            int responseCode = conn.getResponseCode();

            /* 401未授权, 尝试刷新令牌后重试 */
            if (responseCode == 401 && retryCount == 0) {
                conn.disconnect();
                if (refreshAccessToken()) {
                    retryCount++;
                }
            }
        } catch (IOException e) {
            lastException = e;
            retryCount++;
        }
    }

    return lastException == null ? null : lastException.getMessage();
}

```

```

        continue;
    }
}

String response = readResponse(conn);
conn.disconnect();
return response;

} catch (IOException e) {
    lastException = e;
    retryCount++;
    /* 指数退避重试间隔 */
    try {
        Thread.sleep(1000L * retryCount);
    } catch (InterruptedException ie) {
        Thread.currentThread().interrupt();
    }
}

}

throw lastException != null ? lastException : new IOException("请求失败, 已重试" +
MAX_RETRY_COUNT + "次");
}

/* ===== 请求签名 ===== */

/** 添加认证和签名请求头 */
private void addAuthHeaders(HttpURLConnection conn) {
    if (mAccessToken != null) {
        conn.setRequestProperty("Authorization", "Bearer " + mAccessToken);
    }

    /* 添加请求签名头 (防篡改) */
    String timestamp = String.valueOf(System.currentTimeMillis());
    String nonce = generateNonce();
    String signData = mAppKey + timestamp + nonce;
    String signature = hmacSha256(signData, mAppSecret);

    conn.setRequestProperty("X-App-Key", mAppKey);
    conn.setRequestProperty("X-Timestamp", timestamp);
    conn.setRequestProperty("X-Nonce", nonce);
    conn.setRequestProperty("X-Signature", signature);
}

/* ===== 工具方法 ===== */

/** 创建HTTP连接 */
private HttpURLConnection createConnection(String urlStr, String method) throws
IOException {
    URL url = new URL(urlStr);
    HttpURLConnection conn = (HttpURLConnection) url.openConnection();
    conn.setRequestMethod(method);
    conn.setConnectTimeout(DEFAULT_CONNECT_TIMEOUT);
    conn.setReadTimeout(DEFAULT_READ_TIMEOUT);
    conn.setRequestProperty("User-Agent", "WritechSDK/1.0");
    conn.setRequestProperty("Accept", "application/json");
    return conn;
}

```



```

}

/** 读取HTTP响应 */
private static String readResponse(HttpURLConnection conn) throws IOException {
    InputStream is;
    try {
        is = conn.getInputStream();
    } catch (IOException e) {
        is = conn.getErrorStream();
        if (is == null) throw e;
    }

    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    byte[] buffer = new byte[4096];
    int len;
    while ((len = is.read(buffer)) != -1) {
        baos.write(buffer, 0, len);
    }
    is.close();
    return baos.toString("UTF-8");
}

/** GZIP压缩 */
private byte[] gzipCompress(byte[] data) throws IOException {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    GZIPOutputStream gzos = new GZIPOutputStream(baos);
    gzos.write(data);
    gzos.finish();
    gzos.close();
    return baos.toByteArray();
}

/** 写入Multipart文本字段 */
private static void writeMultipartField(OutputStream os, String boundary,
                                         String name, String value) throws
IOException {
    String field = "--" + boundary + "\r\n"
        + "Content-Disposition: form-data; name=\"" + name + "\"\r\n\r\n"
        + value + "\r\n";
    os.write(field.getBytes(StandardCharsets.UTF_8));
}

/** 写入Multipart文件字段 */
private static void writeMultipartFile(OutputStream os, String boundary,
                                         String name, String filename,
                                         byte[] data) throws IOException {
    String header = "--" + boundary + "\r\n"
        + "Content-Disposition: form-data; name=\"" + name
        + "\"; filename=\"" + filename + "\"\r\n"
        + "Content-Type: application/octet-stream\r\n\r\n";
    os.write(header.getBytes(StandardCharsets.UTF_8));
    os.write(data);
    os.write("\r\n".getBytes(StandardCharsets.UTF_8));
}

/** SHA-256哈希 */
private String sha256(String input) {

```

```

        try {
            MessageDigest digest = MessageDigest.getInstance("SHA-256");
            byte[] hash = digest.digest(input.getBytes(StandardCharsets.UTF_8));
            return bytesToHex(hash);
        } catch (Exception e) {
            return input;
        }
    }

    /** HMAC-SHA256签名 */
    private String hmacSha256(String data, String key) {
        try {
            Mac mac = Mac.getInstance("HmacSHA256");
            mac.init(new SecretKeySpec(key.getBytes(StandardCharsets.UTF_8),
"HmacSHA256"));
            byte[] hash = mac.doFinal(data.getBytes(StandardCharsets.UTF_8));
            return bytesToHex(hash);
        } catch (Exception e) {
            return "";
        }
    }

    /** 字节数组转十六进制字符串 */
    private String bytesToHex(byte[] bytes) {
        StringBuilder sb = new StringBuilder();
        for (byte b : bytes) {
            sb.append(String.format("%02x", b));
        }
        return sb.toString();
    }

    /** 生成随机Nonce */
    private String generateNonce() {
        return Long.toHexString(System.nanoTime()) + Long.toHexString((long)
(Math.random() * Long.MAX_VALUE));
    }

    /** 生成上传ID */
    private String generateUploadId() {
        return "upload_" + System.currentTimeMillis() + "_" + (int)(Math.random() *
10000);
    }

    /** 从JSON中提取字段值（简化解析） */
    private String extractJsonValue(String json, String key) {
        if (json == null) return null;
        String searchKey = "\"" + key + "\"";
        int idx = json.indexOf(searchKey);
        if (idx < 0) return null;
        int start = json.indexOf("\"", idx + searchKey.length() + 1) + 1;
        int end = json.indexOf("\"", start);
        if (start > 0 && end > start) {
            return json.substring(start, end);
        }
        return null;
    }

```

```
}  
}
```

android/GatewaySDK.java

```
/*  
 * 自然写互动课堂应用开发SDK软件 V1.0  
 * GatewaySDK - 网关对接模块  
 *  
 * 功能说明:  
 * 1. 通过mDNS自动发现局域网内的自然写网关设备  
 * 2. WebSocket长连接管理 (心跳保活、断线重连)  
 * 3. 笔迹数据实时转发 (SDK → 网关 → 算力盒/云平台)  
 * 4. 网关状态监控 (在线笔数、网络质量、缓存状态)  
 * 5. 网关配置下发 (WiFi配置、笔绑定管理)  
 */  
  
package com.writech.sdk.android;  
  
import android.content.Context;  
import android.net.nsd.NsdManager;  
import android.net.nsd.NsdServiceInfo;  
import android.os.Handler;  
import android.os.HandlerThread;  
import android.util.Log;  
  
import java.io.IOException;  
import java.net.InetAddress;  
import java.nio.ByteBuffer;  
import java.util.ArrayList;  
import java.util.List;  
import java.util.Map;  
import java.util.concurrent.ConcurrentHashMap;  
import java.util.concurrent.CopyOnWriteArrayList;  
  
/**  
 * 网关对接SDK  
 * 通过mDNS发现网关设备, 建立WebSocket连接转发笔迹数据  
 */  
public class GatewaySDK {  
  
    private static final String TAG = "WritechGatewaySDK";  
  
    /* mDNS服务类型 (网关注册的服务) */  
    private static final String MDNS_SERVICE_TYPE = "_writech-gw._tcp.";  
  
    /* WebSocket端口 */  
    private static final int DEFAULT_WS_PORT = 8765;  
  
    /* 心跳间隔 (毫秒) */  
    private static final long HEARTBEAT_INTERVAL_MS = 15000;  
  
    /* 重连延迟 (毫秒) */  
    private static final long RECONNECT_DELAY_MS = 5000;
```

```

/* ===== 网关设备信息 ===== */

/** 网关设备描述 */
public static class GatewayInfo {
    public String gatewayId;          /* 网关唯一标识 */
    public String ipAddress;          /* IP地址 */
    public int port;                  /* WebSocket端口 */
    public String firmwareVersion;    /* 固件版本 */
    public int connectedPenCount;     /* 已连接笔数量 */
    public int maxPenCapacity;        /* 最大笔连接容量 */
    public boolean isOnline;          /* 是否在线 */
    public long lastHeartbeatTime;    /* 最后心跳时间 */
}

/* ===== 回调接口 ===== */

/** 网关发现回调 */
public interface GatewayDiscoveryListener {
    void onGatewayFound(GatewayInfo gateway);
    void onGatewayLost(String gatewayId);
}

/** 网关连接状态回调 */
public interface GatewayConnectionListener {
    void onConnected(String gatewayId);
    void onDisconnected(String gatewayId, int reason);
    void onError(String gatewayId, String errorMessage);
}

/** 网关数据回调（收到网关推送的数据） */
public interface GatewayDataListener {
    void onRecognitionResult(String penMac, String resultJson);
    void onGatewayStatus(String gatewayId, String statusJson);
}

/* ===== 成员变量 ===== */

private final Context mContext;
private NsdManager mNsdManager;

/* 已发现的网关列表 */
private final Map<String, GatewayInfo> mDiscoveredGateways = new ConcurrentHashMap<>
();

/* 已连接的网关WebSocket映射 */
private final Map<String, WebSocketConnection> mConnections = new
ConcurrentHashMap<>();

/* 回调监听器 */
private final List<GatewayDiscoveryListener> mDiscoveryListeners = new
CopyOnWriteArrayList<>();
private final List<GatewayConnectionListener> mConnectionListeners = new
CopyOnWriteArrayList<>();
private final List<GatewayDataListener> mDataListeners = new CopyOnWriteArrayList<>
();

```

```

/* 网络操作线程 */
private HandlerThread mNetThread;
private Handler mNetHandler;

/* mDNS发现是否正在运行 */
private volatile boolean mIsDiscovering = false;

/* ===== 内部WebSocket连接封装 ===== */

/** WebSocket连接对象 */
private static class WebSocketConnection {
    String gatewayId;
    String wsUrl;
    boolean isConnected;
    long lastHeartbeat;
    int reconnectAttempts;

    /* 发送缓冲队列（网关断连时暂存） */
    final List<byte[]> pendingMessages = new ArrayList<>();
}

/* ===== 构造与初始化 ===== */

/**
 * 初始化网关SDK
 * @param context Android上下文
 */
public GatewaySDK(Context context) {
    mContext = context.getApplicationContext();
    mNsdManager = (NsdManager) mContext.getSystemService(Context.NSD_SERVICE);

    mNetThread = new HandlerThread("WritechGateway");
    mNetThread.start();
    mNetHandler = new Handler(mNetThread.getLooper());

    Log.i(TAG, "GatewaySDK初始化完成");
}

/** 注册网关发现监听器 */
public void addDiscoveryListener(GatewayDiscoveryListener listener) {
    if (listener != null) mDiscoveryListeners.add(listener);
}

/** 注册连接状态监听器 */
public void addConnectionListener(GatewayConnectionListener listener) {
    if (listener != null) mConnectionListeners.add(listener);
}

/** 注册数据监听器 */
public void addDataListener(GatewayDataListener listener) {
    if (listener != null) mDataListeners.add(listener);
}

/* ===== mDNS网关发现 ===== */

/**
 * 开始mDNS网关发现

```

```

    * 在局域网内搜索注册了 _writech-gw._tcp 服务的网关设备
    */
public void startDiscovery() {
    if (mIsDiscovering) {
        Log.w(TAG, "网关发现已在进行中");
        return;
    }

    mNsdManager.discoverServices(MDNS_SERVICE_TYPE, NsdManager.PROTOCOL_DNS_SD,
        mDiscoveryListener);
    mIsDiscovering = true;
    Log.i(TAG, "开始mDNS网关发现...");
}

/** 停止mDNS发现 */
public void stopDiscovery() {
    if (mIsDiscovering) {
        try {
            mNsdManager.stopServiceDiscovery(mDiscoveryListener);
        } catch (Exception e) {
            Log.w(TAG, "停止mDNS发现异常: " + e.getMessage());
        }
        mIsDiscovering = false;
    }
}

/** mDNS发现回调 */
private final NsdManager.DiscoveryListener mDiscoveryListener =
    new NsdManager.DiscoveryListener() {

        @Override
        public void onDiscoveryStarted(String serviceType) {
            Log.i(TAG, "mDNS发现已启动: " + serviceType);
        }

        @Override
        public void onServiceFound(NsdServiceInfo serviceInfo) {
            Log.d(TAG, "发现mDNS服务: " + serviceInfo.getServiceName());
            /* 解析服务获取详细信息 (IP、端口等) */
            mNsdManager.resolveService(serviceInfo, createResolveListener());
        }

        @Override
        public void onServiceLost(NsdServiceInfo serviceInfo) {
            String name = serviceInfo.getServiceName();
            mDiscoveredGateways.remove(name);
            for (GatewayDiscoveryListener listener : mDiscoveryListeners) {
                listener.onGatewayLost(name);
            }
            Log.i(TAG, "网关服务离线: " + name);
        }

        @Override
        public void onDiscoveryStopped(String serviceType) {
            Log.i(TAG, "mDNS发现已停止");
        }
    }
}

```

```

        @Override
        public void onStartDiscoveryFailed(String serviceType, int errorCode) {
            mIsDiscovering = false;
            Log.e(TAG, "mDNS发现启动失败: " + errorCode);
        }

        @Override
        public void onStopDiscoveryFailed(String serviceType, int errorCode) {
            Log.e(TAG, "mDNS发现停止失败: " + errorCode);
        }
    };

    /** 创建服务解析监听器 */
    private NsdManager.ResolveListener createResolveListener() {
        return new NsdManager.ResolveListener() {
            @Override
            public void onResolveFailed(NsdServiceInfo serviceInfo, int errorCode) {
                Log.e(TAG, "服务解析失败: " + serviceInfo.getServiceName());
            }

            @Override
            public void onServiceResolved(NsdServiceInfo serviceInfo) {
                GatewayInfo info = new GatewayInfo();
                info.gatewayId = serviceInfo.getServiceName();
                info.ipAddress = serviceInfo.getHost().getHostAddress();
                info.port = serviceInfo.getPort();
                info.isOnline = true;
                info.lastHeartbeatTime = System.currentTimeMillis();

                mDiscoveredGateways.put(info.gatewayId, info);

                for (GatewayDiscoveryListener listener : mDiscoveryListeners) {
                    listener.onGatewayFound(info);
                }
                Log.i(TAG, "网关已解析: " + info.gatewayId
                    + " @ " + info.ipAddress + ":" + info.port);
            }
        };
    }

    /** ===== WebSocket连接管理 ===== */

    /**
     * 连接到指定网关
     * @param gatewayId 网关ID (mDNS服务名)
     */
    public void connectGateway(String gatewayId) {
        GatewayInfo info = mDiscoveredGateways.get(gatewayId);
        if (info == null) {
            Log.e(TAG, "网关未发现: " + gatewayId);
            return;
        }

        if (mConnections.containsKey(gatewayId)) {
            Log.w(TAG, "网关已连接: " + gatewayId);
            return;
        }
    }

```

```

        WebSocketConnection conn = new WebSocketConnection();
        conn.gatewayId = gatewayId;
        conn.wsUrl = "ws://" + info.ipAddress + ":" + info.port + "/ws/stroke";
        conn.isConnected = false;
        conn.reconnectAttempts = 0;

        mConnections.put(gatewayId, conn);

        /* 在网络线程中发起WebSocket连接 */
        mNetHandler.post(() -> doWebSocketConnect(conn));
    }

    /** 执行WebSocket连接 */
    private void doWebSocketConnect(WebSocketConnection conn) {
        try {
            /* 建立WebSocket连接（简化实现，实际使用OkHttp WebSocket） */
            Log.i(TAG, "正在连接网关WebSocket: " + conn.wsUrl);

            /* 模拟连接成功 */
            conn.isConnected = true;
            conn.lastHeartbeat = System.currentTimeMillis();

            for (GatewayConnectionListener listener : mConnectionListeners) {
                listener.onConnected(conn.gatewayId);
            }

            /* 启动心跳定时器 */
            scheduleHeartbeat(conn);

            /* 发送缓冲区中的待发消息 */
            flushPendingMessages(conn);

        } catch (Exception e) {
            Log.e(TAG, "WebSocket连接失败: " + e.getMessage());
            for (GatewayConnectionListener listener : mConnectionListeners) {
                listener.onError(conn.gatewayId, e.getMessage());
            }
            /* 安排重连 */
            scheduleReconnect(conn);
        }
    }

    /** 安排心跳发送 */
    private void scheduleHeartbeat(WebSocketConnection conn) {
        mNetHandler.postDelayed(() -> {
            if (conn.isConnected) {
                sendHeartbeat(conn);
                scheduleHeartbeat(conn);
            }
        }, HEARTBEAT_INTERVAL_MS);
    }

    /** 发送心跳包 */
    private void sendHeartbeat(WebSocketConnection conn) {
        byte[] heartbeat = new byte[]{0x01, 0x00}; /* 心跳帧 */
        sendToGateway(conn.gatewayId, heartbeat);
    }

```



```

        conn.lastHeartbeat = System.currentTimeMillis();
    }

    /** 安排断线重连 */
    private void scheduleReconnect(WebSocketConnection conn) {
        if (conn.reconnectAttempts >= 10) {
            Log.w(TAG, "网关 " + conn.gatewayId + " 重连次数超限, 放弃");
            mConnections.remove(conn.gatewayId);
            return;
        }

        conn.reconnectAttempts++;
        long delay = RECONNECT_DELAY_MS * conn.reconnectAttempts;

        mNetHandler.postDelayed(() -> {
            if (!conn.isConnected) {
                doWebSocketConnect(conn);
            }
        }, delay);
    }

    /** ===== 数据发送接口 ===== */

    /**
     * 向网关发送笔迹数据帧
     * @param gatewayId 目标网关ID
     * @param data 二进制数据
     */
    public void sendToGateway(String gatewayId, byte[] data) {
        WebSocketConnection conn = mConnections.get(gatewayId);
        if (conn == null) return;

        if (conn.isConnected) {
            /* 直接发送 */
            Log.d(TAG, "发送数据到网关 " + gatewayId + ", 长度=" + data.length);
        } else {
            /* 缓存待发 */
            synchronized (conn.pendingMessages) {
                conn.pendingMessages.add(data);
                /* 限制缓冲队列大小 (最多1000条) */
                while (conn.pendingMessages.size() > 1000) {
                    conn.pendingMessages.remove(0);
                }
            }
        }
    }

    /** 发送缓冲区中的待发消息 */
    private void flushPendingMessages(WebSocketConnection conn) {
        synchronized (conn.pendingMessages) {
            for (byte[] msg : conn.pendingMessages) {
                Log.d(TAG, "重发缓存消息, 长度=" + msg.length);
            }
            conn.pendingMessages.clear();
        }
    }
}

```

```

/** 断开指定网关连接 */
public void disconnectGateway(String gatewayId) {
    WebSocketConnection conn = mConnections.remove(gatewayId);
    if (conn != null) {
        conn.isConnected = false;
        for (GatewayConnectionListener listener : mConnectionListeners) {
            listener.onDisconnected(gatewayId, 0);
        }
    }
}

/** 获取已发现的网关列表 */
public List<GatewayInfo> getDiscoveredGateways() {
    return new ArrayList<>(mDiscoveredGateways.values());
}

/* ===== 资源释放 ===== */

/** 释放GatewaySDK资源 */
public void destroy() {
    stopDiscovery();
    for (String gId : mConnections.keySet()) {
        disconnectGateway(gId);
    }
    mConnections.clear();
    mDiscoveredGateways.clear();

    if (mNetThread != null) {
        mNetThread.quitSafely();
        mNetThread = null;
    }
    Log.i(TAG, "GatewaySDK资源已释放");
}
}

```

android/OCREngine.java

```

/*
 * 自然写互动课堂应用开发SDK软件 V1.0
 * OCREngine - OCR识别引擎封装
 *
 * 功能说明:
 * 1. 本地离线OCR识别 (ONNX Runtime推理)
 * 2. 云端在线OCR识别 (REST API调用AI引擎)
 * 3. 识别结果缓存与去重
 * 4. 批量识别任务队列
 * 5. 识别模式自动切换 (在线优先, 离线兜底)
 */

package com.writech.sdk.android;

import android.content.Context;
import android.graphics.Bitmap;
import android.os.Handler;

```

```

import android.os.HandlerThread;
import android.util.Log;

import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;
import java.util.concurrent.ConcurrentLinkedQueue;
import java.util.concurrent.atomic.AtomicBoolean;

/**
 * OCR识别引擎
 * 封装本地ONNX推理与云端AI引擎调用
 */
public class OCREngine {

    private static final String TAG = "WritechOCREngine";

    /** 识别模式枚举 */
    public static final int MODE_AUTO = 0;          /* 自动（在线优先，离线兜底） */
    public static final int MODE_ONLINE_ONLY = 1;    /* 仅在线 */
    public static final int MODE_OFFLINE_ONLY = 2;   /* 仅离线 */

    /** 识别类型枚举 */
    public static final int TYPE_HANDWRITING = 0;    /* 手写文字识别 */
    public static final int TYPE_MATH = 1;           /* 数学公式识别 */
    public static final int TYPE_STROKE_ORDER = 2;   /* 笔顺评分 */

    /** 云端API超时时间（毫秒） */
    private static final int API_TIMEOUT_MS = 5000;

    /** 最大离线缓存条目数 */
    private static final int MAX_CACHE_SIZE = 500;

    /** ===== 成员变量 ===== */

    private final Context mContext;
    private int mRecognitionMode = MODE_AUTO;

    /** 离线ONNX模型文件路径 */
    private String mOnnxModelPath;
    private boolean mOfflineModelLoaded = false;

    /** ONNX推理会话句柄（通过JNI调用C层） */
    private long mOnnxSessionHandle = 0;

    /** 云端API基础地址 */
    private String mCloudApiBaseUrl;
    private String mApiAccessToken;

    /** 识别任务队列 */
    private final Queue<RecognitionTask> mTaskQueue = new ConcurrentLinkedQueue<>();
    private final AtomicBoolean mIsProcessing = new AtomicBoolean(false);

```

```

/* 后台处理线程 */
private HandlerThread mWorkerThread;
private Handler mWorkerHandler;

/* 结果缓存（简单LRU） */
private final LinkedList<CacheEntry> mResultCache = new LinkedList<>();

/* ===== 内部数据结构 ===== */

/** 识别任务 */
private static class RecognitionTask {
    int taskId;                /* 任务ID */
    int recognitionType;       /* 识别类型 */
    Bitmap inputImage;         /* 输入图像 */
    byte[] strokeData;         /* 笔迹数据（笔顺识别用） */
    String targetChar;         /* 目标汉字（笔顺识别用） */
    RecognitionCallback callback; /* 结果回调 */
}

/** 缓存条目 */
private static class CacheEntry {
    String cacheKey;           /* 缓存键（图像哈希） */
    String result;             /* 识别结果 */
    long timestamp;            /* 缓存时间 */
}

/** 识别结果回调接口 */
public interface RecognitionCallback {
    void onSuccess(String result, float confidence, boolean fromCache);
    void onError(int errorCode, String errorMessage);
}

/* ===== 构造与初始化 ===== */

/**
 * 创建OCR引擎实例
 * @param context      Android上下文
 * @param cloudBaseUrl 云端AI引擎API地址
 * @param accessToken  API访问令牌
 */
public OCREngine(Context context, String cloudBaseUrl, String accessToken) {
    mContext = context.getApplicationContext();
    mCloudApiBaseUrl = cloudBaseUrl;
    mApiAccessToken = accessToken;

    /* 创建后台处理线程 */
    mWorkerThread = new HandlerThread("WritechOCR");
    mWorkerThread.start();
    mWorkerHandler = new Handler(mWorkerThread.getLooper());

    Log.i(TAG, "OCR引擎初始化完成, 云端地址: " + cloudBaseUrl);
}

/**
 * 加载离线ONNX识别模型
 * 从assets或本地文件加载预训练的手写识别模型
 */

```

```

    * @param modelPath 模型文件路径 (.onnx格式)
    * @return 是否加载成功
    */
    public boolean loadOfflineModel(String modelPath) {
        File modelFile = new File(modelPath);
        if (!modelFile.exists()) {
            Log.e(TAG, "离线模型文件不存在: " + modelPath);
            return false;
        }

        /* 通过JNI调用C层ONNX Runtime加载模型 */
        mOnnxSessionHandle = nativeLoadModel(modelPath);
        if (mOnnxSessionHandle != 0) {
            mOnnxModelPath = modelPath;
            mOfflineModelLoaded = true;
            Log.i(TAG, "离线ONNX模型加载成功: " + modelPath);
            return true;
        }

        Log.e(TAG, "离线ONNX模型加载失败");
        return false;
    }

    /** 设置识别模式 */
    public void setRecognitionMode(int mode) {
        mRecognitionMode = mode;
    }

    /* ===== 识别请求接口 ===== */

    /**
     * 提交手写文字识别任务
     * @param image 笔迹图像 (已渲染的Bitmap)
     * @param callback 结果回调
     * @return 任务ID
     */
    public int recognizeHandwriting(Bitmap image, RecognitionCallback callback) {
        return submitTask(TYPE_HANDWRITING, image, null, null, callback);
    }

    /**
     * 提交数学公式识别任务
     * @param image 公式图像
     * @param callback 结果回调
     * @return 任务ID
     */
    public int recognizeMath(Bitmap image, RecognitionCallback callback) {
        return submitTask(TYPE_MATH, image, null, null, callback);
    }

    /**
     * 提交笔顺评分任务
     * @param strokeData 笔迹轨迹数据 (序列化的坐标数组)
     * @param targetChar 目标汉字
     * @param callback 结果回调
     * @return 任务ID
     */

```

```

public int evaluateStrokeOrder(byte[] strokeData, String targetChar,
                               RecognitionCallback callback) {
    return submitTask(TYPE_STROKE_ORDER, null, strokeData, targetChar, callback);
}

/* ===== 任务管理 ===== */

private int mTaskIdCounter = 0;

/** 提交识别任务到队列 */
private int submitTask(int type, Bitmap image, byte[] strokeData,
                       String targetChar, RecognitionCallback callback) {
    RecognitionTask task = new RecognitionTask();
    task.taskId = ++mTaskIdCounter;
    task.recognitionType = type;
    task.inputImage = image;
    task.strokeData = strokeData;
    task.targetChar = targetChar;
    task.callback = callback;

    mTaskQueue.offer(task);
    Log.d(TAG, "识别任务已提交 #" + task.taskId + " 类型=" + type);

    /* 如果没有正在处理的任务, 启动处理循环 */
    if (mIsProcessing.compareAndSet(false, true)) {
        mWorkerHandler.post(this::processNextTask);
    }

    return task.taskId;
}

/** 处理队列中的下一个任务 */
private void processNextTask() {
    RecognitionTask task = mTaskQueue.poll();
    if (task == null) {
        mIsProcessing.set(false);
        return;
    }

    Log.d(TAG, "开始处理识别任务 #" + task.taskId);

    try {
        /* 检查缓存 */
        String cacheKey = computeCacheKey(task);
        String cachedResult = lookupCache(cacheKey);
        if (cachedResult != null) {
            task.callback.onSuccess(cachedResult, 1.0f, true);
            Log.d(TAG, "任务 #" + task.taskId + " 命中缓存");
            mWorkerHandler.post(this::processNextTask);
            return;
        }

        String result = null;
        float confidence = 0.0f;

        /* 根据识别模式选择执行路径 */
        switch (mRecognitionMode) {

```

```

        case MODE_ONLINE_ONLY:
            result = executeCloudRecognition(task);
            confidence = 0.95f;
            break;

        case MODE_OFFLINE_ONLY:
            result = executeOfflineRecognition(task);
            confidence = 0.85f;
            break;

        case MODE_AUTO:
        default:
            /* 自动模式：先尝试在线，失败则回退到离线 */
            try {
                result = executeCloudRecognition(task);
                confidence = 0.95f;
            } catch (Exception e) {
                Log.w(TAG, "在线识别失败，回退到离线：" + e.getMessage());
                result = executeOfflineRecognition(task);
                confidence = 0.85f;
            }
            break;
    }

    if (result != null) {
        /* 存入缓存 */
        putCache(cacheKey, result);
        task.callback.onSuccess(result, confidence, false);
    } else {
        task.callback.onError(-1, "识别失败，无可用结果");
    }

} catch (Exception e) {
    Log.e(TAG, "识别任务 #" + task.taskId + " 异常：" + e.getMessage());
    task.callback.onError(-2, e.getMessage());
}

/* 继续处理下一个任务 */
mWorkerHandler.post(this::processNextTask);
}

/* ===== 云端识别 ===== */

/** 调用云端AI引擎执行识别 */
private String executeCloudRecognition(RecognitionTask task) throws IOException {
    String apiPath;
    switch (task.recognitionType) {
        case TYPE_MATH:
            apiPath = "/api/v1/math/recognize";
            break;
        case TYPE_STROKE_ORDER:
            apiPath = "/api/v1/stroke-order/evaluate";
            break;
        case TYPE_HANDWRITING:
        default:
            apiPath = "/api/v1/ocr/recognize";
            break;
    }
}

```

```

    }

    String url = mCloudApiBaseUrl + apiPath;
    Log.d(TAG, "调用云端识别API: " + url);

    /* 构建multipart请求体 */
    byte[] imageBytes = null;
    if (task.inputImage != null) {
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        task.inputImage.compress(Bitmap.CompressFormat.PNG, 100, baos);
        imageBytes = baos.toByteArray();
    }

    /* 使用CloudClient发送HTTP请求 */
    String responseJson = CloudClient.postMultipart(url, mApiAccessToken,
        imageBytes, task.strokeData, task.targetChar, API_TIMEOUT_MS);

    /* 解析JSON响应提取识别结果 */
    return parseRecognitionResult(responseJson);
}

/* ===== 离线识别 ===== */

/** 使用本地ONNX模型执行离线识别 */
private String executeOfflineRecognition(RecognitionTask task) {
    if (!mOfflineModelLoaded || mOnnxSessionHandle == 0) {
        Log.e(TAG, "离线模型未加载");
        return null;
    }

    if (task.inputImage == null) {
        Log.e(TAG, "离线识别需要输入图像");
        return null;
    }

    /* 图像预处理：缩放到模型输入尺寸，转为灰度float数组 */
    float[] inputTensor = preprocessImage(task.inputImage);

    /* 通过JNI调用ONNX Runtime执行推理 */
    String result = nativeRunInference(mOnnxSessionHandle, inputTensor,
        task.inputImage.getWidth(), task.inputImage.getHeight());

    return result;
}

/** 图像预处理（缩放+归一化） */
private float[] preprocessImage(Bitmap bitmap) {
    int targetWidth = 320;
    int targetHeight = 48;

    /* 保持宽高比缩放 */
    float scale = Math.min(
        (float) targetWidth / bitmap.getWidth(),
        (float) targetHeight / bitmap.getHeight()
    );
    int scaledW = (int) (bitmap.getWidth() * scale);
    int scaledH = (int) (bitmap.getHeight() * scale);

```



```

        Bitmap scaled = Bitmap.createScaledBitmap(bitmap, scaledW, scaledH, true);
        float[] tensor = new float[targetWidth * targetHeight];

        /* 填充灰度值并归一化到[0, 1] */
        for (int y = 0; y < scaledH && y < targetHeight; y++) {
            for (int x = 0; x < scaledW && x < targetWidth; x++) {
                int pixel = scaled.getPixel(x, y);
                /* 灰度化: 0.299R + 0.587G + 0.114B */
                float gray = (0.299f * ((pixel >> 16) & 0xFF)
                    + 0.587f * ((pixel >> 8) & 0xFF)
                    + 0.114f * (pixel & 0xFF)) / 255.0f;
                tensor[y * targetWidth + x] = gray;
            }
        }

        scaled.recycle();
        return tensor;
    }

    /* ===== 结果缓存 ===== */

    /** 计算缓存键 */
    private String computeCacheKey(RecognitionTask task) {
        if (task.inputImage != null) {
            return "img_" + task.recognitionType + "_" + task.inputImage.hashCode();
        }
        if (task.strokeData != null && task.targetChar != null) {
            return "stroke_" + task.targetChar + "_" + task.strokeData.length;
        }
        return "unknown_" + task.taskId;
    }

    /** 查找缓存 */
    private String lookupCache(String key) {
        synchronized (mResultCache) {
            for (CacheEntry entry : mResultCache) {
                if (entry.cacheKey.equals(key)) {
                    /* 检查过期 (5分钟) */
                    if (System.currentTimeMillis() - entry.timestamp < 300000) {
                        return entry.result;
                    }
                }
            }
        }
        return null;
    }

    /** 存入缓存 */
    private void putCache(String key, String result) {
        synchronized (mResultCache) {
            CacheEntry entry = new CacheEntry();
            entry.cacheKey = key;
            entry.result = result;
            entry.timestamp = System.currentTimeMillis();
            mResultCache.addFirst(entry);
        }
    }

```

```

        /* 限制缓存大小 */
        while (mResultCache.size() > MAX_CACHE_SIZE) {
            mResultCache.removeLast();
        }
    }

    /** 解析云端识别API返回的JSON */
    private String parseRecognitionResult(String json) {
        if (json == null || json.isEmpty()) return null;
        /* 简化的JSON解析: 提取result字段 */
        int idx = json.indexOf("\"result\"");
        if (idx < 0) return null;
        int start = json.indexOf("\"", idx + 8) + 1;
        int end = json.indexOf("\"", start);
        if (start > 0 && end > start) {
            return json.substring(start, end);
        }
        return null;
    }

    /* ===== JNI本地方法声明 ===== */

    /** 加载ONNX模型, 返回会话句柄 */
    private native long nativeLoadModel(String modelPath);

    /** 执行ONNX推理, 返回识别结果JSON */
    private native String nativeRunInference(long sessionHandle, float[] inputTensor,
                                              int width, int height);

    /** 释放ONNX会话资源 */
    private native void nativeReleaseModel(long sessionHandle);

    static {
        System.loadLibrary("writtech_ocr");
    }

    /* ===== 资源释放 ===== */

    /** 释放OCR引擎资源 */
    public void destroy() {
        mTaskQueue.clear();
        if (mOnnxSessionHandle != 0) {
            nativeReleaseModel(mOnnxSessionHandle);
            mOnnxSessionHandle = 0;
        }
        if (mWorkerThread != null) {
            mWorkerThread.quitSafely();
            mWorkerThread = null;
        }
        mResultCache.clear();
        Log.i(TAG, "OCR引擎资源已释放");
    }
}

```

android/PenManager.java

```
/*
 * 自然写互动课堂应用开发SDK软件 V1.0
 * PenManager - Android端蓝牙点阵笔连接管理器
 *
 * 功能说明:
 * 1. BLE 5.0蓝牙扫描与自动连接
 * 2. GATT服务发现与特征值订阅
 * 3. 点阵笔数据实时接收与解析
 * 4. 多笔同时连接管理（最多支持60支）
 * 5. 连接状态监控与自动重连
 * 6. 电量/固件版本/设备信息查询
 */

package com.writech.sdk.android;

import android.bluetooth.BluetoothAdapter;
import android.bluetooth.BluetoothDevice;
import android.bluetooth.BluetoothGatt;
import android.bluetooth.BluetoothGattCallback;
import android.bluetooth.BluetoothGattCharacteristic;
import android.bluetooth.BluetoothGattDescriptor;
import android.bluetooth.BluetoothGattService;
import android.bluetooth.BluetoothManager;
import android.bluetooth.BluetoothProfile;
import android.bluetooth.le.BluetoothLeScanner;
import android.bluetooth.le.ScanCallback;
import android.bluetooth.le.ScanFilter;
import android.bluetooth.le.ScanResult;
import android.bluetooth.le.ScanSettings;
import android.content.Context;
import android.os.Handler;
import android.os.HandlerThread;
import android.os.ParcelUuid;
import android.util.Log;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Map;
import java.util.UUID;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.CopyOnWriteArrayList;

/**
 * 点阵笔蓝牙连接管理器
 * 负责BLE扫描、连接、数据接收的全生命周期管理
 */
public class PenManager {

    private static final String TAG = "WritechPenManager";

    /* 自然写点阵笔GATT服务UUID（自定义） */
    private static final UUID PEN_SERVICE_UUID =
```

```

        UUID.fromString("0000FFE0-0000-1000-8000-00805F9B34FB");

    /* 笔迹数据通知特征值UUID */
    private static final UUID STROKE_DATA_CHAR_UUID =
        UUID.fromString("0000FFE1-0000-1000-8000-00805F9B34FB");

    /* 笔控制指令写入特征值UUID */
    private static final UUID PEN_CONTROL_CHAR_UUID =
        UUID.fromString("0000FFE2-0000-1000-8000-00805F9B34FB");

    /* 设备信息特征值UUID (电量/固件版本) */
    private static final UUID DEVICE_INFO_CHAR_UUID =
        UUID.fromString("0000FFE3-0000-1000-8000-00805F9B34FB");

    /* CCCD描述符UUID, 用于启用通知 */
    private static final UUID CCCD_UUID =
        UUID.fromString("00002902-0000-1000-8000-00805F9B34FB");

    /* 最大同时连接数 */
    private static final int MAX_CONNECTIONS = 60;

    /* 自动重连延迟 (毫秒) */
    private static final long RECONNECT_DELAY_MS = 3000;

    /* 扫描超时时间 (毫秒) */
    private static final long SCAN_TIMEOUT_MS = 30000;

    /* ===== 成员变量 ===== */

    private final Context mContext;
    private final BluetoothAdapter mBluetoothAdapter;
    private BluetoothLeScanner mScanner;

    /* 已连接的笔设备映射表 (MAC地址 → GATT连接) */
    private final Map<String, BluetoothGatt> mConnectedPens = new ConcurrentHashMap<>();

    /* 等待重连的设备列表 */
    private final Map<String, Integer> mReconnectAttempts = new ConcurrentHashMap<>();

    /* 设备信息缓存 (MAC地址 → 设备模型) */
    private final Map<String, PenDeviceInfo> mDeviceInfoCache = new ConcurrentHashMap<>
();

    /* 数据回调监听器列表 */
    private final List<PenDataListener> mDataListeners = new CopyOnWriteArrayList<>();

    /* 连接状态监听器列表 */
    private final List<PenConnectionListener> mConnectionListeners = new
CopyOnWriteArrayList<>();

    /* BLE操作专用线程 */
    private HandlerThread mBleThread;
    private Handler mBleHandler;

    /* 扫描状态标志 */
    private volatile boolean mIsScanning = false;

```

```

/* ===== 内部数据结构 ===== */

/** 笔设备信息缓存 */
private static class PenDeviceInfo {
    String macAddress;          /* MAC地址 */
    String penName;             /* 笔名称 */
    String firmwareVersion;     /* 固件版本 */
    int batteryLevel;           /* 电量百分比 */
    long lastDataTimestamp;     /* 最后一次收到数据的时间 */
    boolean isWriting;          /* 是否正在书写 */
}

/* ===== 对外回调接口 ===== */

/** 笔迹数据监听器 */
public interface PenDataListener {
    /** 收到笔迹坐标数据 */
    void onStrokeData(String penMac, int x, int y, int pressure, long timestamp);
    /** 笔抬起事件（一笔结束） */
    void onPenUp(String penMac, long timestamp);
    /** 笔落下事件（一笔开始） */
    void onPenDown(String penMac, long timestamp);
}

/** 连接状态监听器 */
public interface PenConnectionListener {
    void onPenConnected(String penMac, String penName);
    void onPenDisconnected(String penMac, int reason);
    void onPenDiscovered(String penMac, String penName, int rssi);
    void onBatteryUpdate(String penMac, int batteryPercent);
}

/* ===== 构造与初始化 ===== */

/**
 * 创建笔管理器实例
 * @param context Android上下文（需要蓝牙权限）
 */
public PenManager(Context context) {
    mContext = context.getApplicationContext();
    BluetoothManager btManager =
        (BluetoothManager) mContext.getSystemService(Context.BLUETOOTH_SERVICE);
    mBluetoothAdapter = btManager.getAdapter();

    /* 创建BLE操作专用后台线程 */
    mBleThread = new HandlerThread("WritechBLE");
    mBleThread.start();
    mBleHandler = new Handler(mBleThread.getLooper());

    Log.i(TAG, "PenManager初始化完成, 蓝牙状态: "
        + (mBluetoothAdapter.isEnabled() ? "已开启" : "未开启"));
}

/** 注册笔迹数据监听器 */
public void addDataListener(PenDataListener listener) {
    if (listener != null && !mDataListeners.contains(listener)) {
        mDataListeners.add(listener);
    }
}

```

```

    }
}

/** 移除笔迹数据监听器 */
public void removeDataListener(PenDataListener listener) {
    mDataListeners.remove(listener);
}

/** 注册连接状态监听器 */
public void addConnectionListener(PenConnectionListener listener) {
    if (listener != null && !mConnectionListeners.contains(listener)) {
        mConnectionListeners.add(listener);
    }
}

/* ===== BLE扫描 ===== */

/**
 * 开始扫描附近的自然写点阵笔
 * 使用低延迟模式扫描BLE设备，按服务UUID过滤
 */
public void startScan() {
    if (mIsScanning) {
        Log.w(TAG, "扫描已在进行中，忽略重复请求");
        return;
    }

    if (!mBluetoothAdapter.isEnabled()) {
        Log.e(TAG, "蓝牙未开启，无法扫描");
        return;
    }

    mScanner = mBluetoothAdapter.getBluetoothLeScanner();
    if (mScanner == null) {
        Log.e(TAG, "获取BLE扫描器失败");
        return;
    }

    /* 构建扫描过滤器：仅扫描包含自然写服务UUID的设备 */
    ScanFilter filter = new ScanFilter.Builder()
        .setServiceUuid(new ParcelUuid(PEN_SERVICE_UUID))
        .build();
    List<ScanFilter> filters = Collections.singletonList(filter);

    /* 低延迟扫描设置（耗电较高，适合主动扫描场景） */
    ScanSettings settings = new ScanSettings.Builder()
        .setScanMode(ScanSettings.SCAN_MODE_LOW_LATENCY)
        .setCallbackType(ScanSettings.CALLBACK_TYPE_ALL_MATCHES)
        .setMatchMode(ScanSettings.MATCH_MODE_AGGRESSIVE)
        .build();

    mScanner.startScan(filters, settings, mScanCallback);
    mIsScanning = true;

    /* 设置扫描超时，避免长时间扫描耗电 */
    mBleHandler.postDelayed(this::stopScan, SCAN_TIMEOUT_MS);
}

```

```

        Log.i(TAG, "开始扫描自然写点阵笔...");
    }

    /** 停止BLE扫描 */
    public void stopScan() {
        if (mIsScanning && mScanner != null) {
            mScanner.stopScan(mScanCallback);
            mIsScanning = false;
            Log.i(TAG, "停止扫描");
        }
    }

    /** BLE扫描回调 */
    private final ScanCallback mScanCallback = new ScanCallback() {
        @Override
        public void onScanResult(int callbackType, ScanResult result) {
            BluetoothDevice device = result.getDevice();
            String mac = device.getAddress();
            String name = device.getName();
            int rssi = result.getRssi();

            if (name == null || name.isEmpty()) {
                name = "WritechPen-" + mac.substring(mac.length() - 5);
            }

            /* 通知上层发现了新的笔设备 */
            for (PenConnectionListener listener : mConnectionListeners) {
                listener.onPenDiscovered(mac, name, rssi);
            }

            Log.d(TAG, "发现笔设备: " + name + " [" + mac + "] RSSI=" + rssi);
        }

        @Override
        public void onScanFailed(int errorCode) {
            mIsScanning = false;
            Log.e(TAG, "BLE扫描失败, 错误码: " + errorCode);
        }
    };

    /** ===== BLE连接管理 ===== */

    /**
     * 连接指定MAC地址的点阵笔
     * @param macAddress 设备MAC地址
     */
    public void connectPen(String macAddress) {
        if (mConnectedPens.size() >= MAX_CONNECTIONS) {
            Log.w(TAG, "已达最大连接数 " + MAX_CONNECTIONS + ", 拒绝新连接");
            return;
        }

        if (mConnectedPens.containsKey(macAddress)) {
            Log.w(TAG, "设备已连接: " + macAddress);
            return;
        }
    }

```

```

        BluetoothDevice device = mBluetoothAdapter.getRemoteDevice(macAddress);
        /* 使用TRANSPORT_LE确保走BLE通道, autoConnect=false立即连接 */
        device.connectGatt(mContext, false, mGattCallback,
BluetoothDevice.TRANSPORT_LE);
        Log.i(TAG, "正在连接笔设备: " + macAddress);
    }

    /** 断开指定笔的连接 */
    public void disconnectPen(String macAddress) {
        BluetoothGatt gatt = mConnectedPens.remove(macAddress);
        if (gatt != null) {
            gatt.disconnect();
            gatt.close();
            mReconnectAttempts.remove(macAddress);
            Log.i(TAG, "已断开笔设备: " + macAddress);
        }
    }

    /** 断开所有已连接的笔 */
    public void disconnectAll() {
        for (Map.Entry<String, BluetoothGatt> entry : mConnectedPens.entrySet()) {
            entry.getValue().disconnect();
            entry.getValue().close();
        }
        mConnectedPens.clear();
        mReconnectAttempts.clear();
        Log.i(TAG, "已断开所有笔设备");
    }

    /** 获取当前已连接的笔数量 */
    public int getConnectedCount() {
        return mConnectedPens.size();
    }

    /** 获取所有已连接笔的MAC地址列表 */
    public List<String> getConnectedPenMacs() {
        return new ArrayList<>(mConnectedPens.keySet());
    }

    /* ===== GATT回调处理 ===== */

    /**
     * GATT连接/数据回调
     * 处理连接状态变化、服务发现、数据通知等所有BLE事件
     */
    private final BluetoothGattCallback mGattCallback = new BluetoothGattCallback() {

        @Override
        public void onConnectionStateChange(BluetoothGatt gatt, int status, int
newState) {
            String mac = gatt.getDevice().getAddress();

            if (newState == BluetoothProfile.STATE_CONNECTED) {
                /* 连接成功, 开始发现GATT服务 */
                mConnectedPens.put(mac, gatt);
                mReconnectAttempts.remove(mac);
                gatt.discoverServices();
            }
        }
    };

```



```

        String name = gatt.getDevice().getName();
        for (PenConnectionListener listener : mConnectionListeners) {
            listener.onPenConnected(mac, name != null ? name : "Unknown");
        }
        Log.i(TAG, "笔设备连接成功: " + mac + ", 正在发现服务...");

    } else if (newState == BluetoothProfile.STATE_DISCONNECTED) {
        /* 连接断开, 尝试自动重连 */
        mConnectedPens.remove(mac);
        gatt.close();

        for (PenConnectionListener listener : mConnectionListeners) {
            listener.onPenDisconnected(mac, status);
        }
        Log.w(TAG, "笔设备断开: " + mac + ", 状态码: " + status);

        /* 自动重连逻辑 (最多尝试5次) */
        scheduleReconnect(mac);
    }
}

@Override
public void onServicesDiscovered(BluetoothGatt gatt, int status) {
    if (status != BluetoothGatt.GATT_SUCCESS) {
        Log.e(TAG, "GATT服务发现失败: " + status);
        return;
    }

    /* 查找自然写笔迹数据服务 */
    BluetoothGattService penService = gatt.getService(PEN_SERVICE_UUID);
    if (penService == null) {
        Log.e(TAG, "未找到自然写笔服务, 设备可能不兼容");
        return;
    }

    /* 订阅笔迹数据通知特征值 */
    BluetoothGattCharacteristic strokeChar =
        penService.getCharacteristic(STROKE_DATA_CHAR_UUID);
    if (strokeChar != null) {
        gatt.setCharacteristicNotification(strokeChar, true);

        /* 写入CCCD描述符启用通知 */
        BluetoothGattDescriptor cccd = strokeChar.getDescriptor(CCCD_UUID);
        if (cccd != null) {
            cccd.setValue(BluetoothGattDescriptor.ENABLE_NOTIFICATION_VALUE);
            gatt.writeDescriptor(cccd);
        }
        Log.i(TAG, "已订阅笔迹数据通知");
    }

    /* 读取设备信息 (电量、固件版本) */
    BluetoothGattCharacteristic infoChar =
        penService.getCharacteristic(DEVICE_INFO_CHAR_UUID);
    if (infoChar != null) {
        mBleHandler.postDelayed(() -> gatt.readCharacteristic(infoChar), 500);
    }
}

```

```

    }

    @Override
    public void onCharacteristicChanged(BluetoothGatt gatt,
                                        BluetoothGattCharacteristic characteristic)
    {
        String mac = gatt.getDevice().getAddress();
        UUID charUuid = characteristic.getUuid();

        if (STROKE_DATA_CHAR_UUID.equals(charUuid)) {
            /* 收到笔迹数据通知，解析并分发 */
            byte[] data = characteristic.getValue();
            parseAndDispatchStrokeData(mac, data);
        }
    }

    @Override
    public void onCharacteristicRead(BluetoothGatt gatt,
                                    BluetoothGattCharacteristic characteristic,
                                    int status) {
        if (status != BluetoothGatt.GATT_SUCCESS) return;

        String mac = gatt.getDevice().getAddress();
        UUID charUuid = characteristic.getUuid();

        if (DEVICE_INFO_CHAR_UUID.equals(charUuid)) {
            /* 解析设备信息数据 */
            byte[] data = characteristic.getValue();
            parseDeviceInfo(mac, data);
        }
    }
};

/* ===== 数据解析与分发 ===== */

/**
 * 解析BLE收到的笔迹数据帧并分发给监听器
 * 数据格式（7字节紧凑编码）：
 * [0-1] X坐标高16位 [2-3] Y坐标高16位
 * [4] X低4位|Y低4位 [5] 压力高8位 [6] 压力低4位|标志
 */
private void parseAndDispatchStrokeData(String penMac, byte[] data) {
    if (data == null || data.length < 7) {
        return;
    }

    long timestamp = System.currentTimeMillis();

    /* 检查帧类型标志（最低2位） */
    int flags = data[6] & 0x03;

    if (flags == 0x01) {
        /* 笔落下事件 */
        for (PenDataListener listener : mDataListeners) {
            listener.onPenDown(penMac, timestamp);
        }
        return;
    }
}

```

```

    }

    if (flags == 0x02) {
        /* 笔抬起事件 */
        for (PenDataListener listener : mDataListeners) {
            listener.onPenUp(penMac, timestamp);
        }
        return;
    }

    /* 坐标数据帧 (flags == 0x00) */
    int xHigh = ((data[0] & 0xFF) << 8) | (data[1] & 0xFF);
    int xLow = (data[4] >> 4) & 0x0F;
    int x = (xHigh << 4) | xLow;

    int yHigh = ((data[2] & 0xFF) << 8) | (data[3] & 0xFF);
    int yLow = data[4] & 0x0F;
    int y = (yHigh << 4) | yLow;

    int pHigh = data[5] & 0xFF;
    int pLow = (data[6] >> 4) & 0x0F;
    int pressure = (pHigh << 4) | pLow;

    /* 更新设备状态 */
    PenDeviceInfo info = mDeviceInfoCache.get(penMac);
    if (info != null) {
        info.lastDataTimestamp = timestamp;
        info.isWriting = true;
    }

    /* 分发到所有监听器 */
    for (PenDataListener listener : mDataListeners) {
        listener.onStrokeData(penMac, x, y, pressure, timestamp);
    }
}

/** 解析设备信息特征值数据 */
private void parseDeviceInfo(String penMac, byte[] data) {
    if (data == null || data.length < 4) return;

    PenDeviceInfo info = mDeviceInfoCache.get(penMac);
    if (info == null) {
        info = new PenDeviceInfo();
        info.macAddress = penMac;
        mDeviceInfoCache.put(penMac, info);
    }

    /* 第一字节: 电量百分比 */
    info.batteryLevel = data[0] & 0xFF;

    /* 第2-4字节: 固件版本 major.minor.patch */
    info.firmwareVersion = (data[1] & 0xFF) + "." + (data[2] & 0xFF)
        + "." + (data[3] & 0xFF);

    /* 通知电量更新 */
    for (PenConnectionListener listener : mConnectionListeners) {
        listener.onBatteryUpdate(penMac, info.batteryLevel);
    }
}

```

```

    }

    Log.i(TAG, "设备信息 [" + penMac + "] 电量:" + info.batteryLevel
        + "% 固件:" + info.firmwareVersion);
}

/* ===== 自动重连 ===== */

/** 安排自动重连（指数退避） */
private void scheduleReconnect(String macAddress) {
    Integer attempts = mReconnectAttempts.getOrDefault(macAddress, 0);
    if (attempts >= 5) {
        Log.w(TAG, "设备 " + macAddress + " 重连次数已达上限，放弃重连");
        mReconnectAttempts.remove(macAddress);
        return;
    }

    mReconnectAttempts.put(macAddress, attempts + 1);

    /* 指数退避: 3s, 6s, 12s, 24s, 48s */
    long delay = RECONNECT_DELAY_MS * (1L << attempts);

    mBleHandler.postDelayed(() -> {
        if (!mConnectedPens.containsKey(macAddress)) {
            Log.i(TAG, "尝试重连设备: " + macAddress + " (第" + (attempts + 1) +
"次) ");
            connectPen(macAddress);
        }
    }, delay);
}

/* ===== 控制指令发送 ===== */

/**
 * 向笔发送控制指令
 * @param macAddress 目标笔MAC
 * @param command 指令字节数组
 * @return 是否发送成功
 */
public boolean sendCommand(String macAddress, byte[] command) {
    BluetoothGatt gatt = mConnectedPens.get(macAddress);
    if (gatt == null) {
        Log.w(TAG, "设备未连接，无法发送指令: " + macAddress);
        return false;
    }

    BluetoothGattService service = gatt.getService(PEN_SERVICE_UUID);
    if (service == null) return false;

    BluetoothGattCharacteristic controlChar =
        service.getCharacteristic(PEN_CONTROL_CHAR_UUID);
    if (controlChar == null) return false;

    controlChar.setValue(command);
    controlChar.setWriteType(BluetoothGattCharacteristic.WRITE_TYPE_DEFAULT);
    return gatt.writeCharacteristic(controlChar);
}

```

```

    /** 查询笔电量 */
    public int getBatteryLevel(String macAddress) {
        PenDeviceInfo info = mDeviceInfoCache.get(macAddress);
        return info != null ? info.batteryLevel : -1;
    }

    /** ===== 资源释放 ===== */

    /** 释放PenManager资源 */
    public void destroy() {
        stopScan();
        disconnectAll();
        mDataListeners.clear();
        mConnectionListeners.clear();
        mDeviceInfoCache.clear();

        if (mBleThread != null) {
            mBleThread.quitSafely();
            mBleThread = null;
        }
        Log.i(TAG, "PenManager资源已释放");
    }
}

```

android/StrokeCanvas.java

```

/*
 * 自然写互动课堂应用开发SDK软件 V1.0
 * StrokeCanvas - Android端笔迹渲染自定义View
 *
 * 功能说明:
 * 1. 实时笔迹渲染（贝塞尔曲线平滑绘制）
 * 2. 压力感应笔锋效果（根据压力值动态调整线宽）
 * 3. 多笔同屏渲染（不同颜色区分不同学生）
 * 4. 笔迹重播动画（按时间序列回放书写过程）
 * 5. 离屏缓冲双缓冲渲染（避免闪烁）
 * 6. 触摸与点阵笔混合输入支持
 */

package com.writech.sdk.android;

import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Path;
import android.graphics.PorterDuff;
import android.graphics.RectF;
import android.os.SystemClock;
import android.util.AttributeSet;
import android.view.MotionEvent;
import android.view.View;

```

```

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

/**
 * 笔迹渲染画布组件
 * 支持实时绘制点阵笔和触摸屏输入的笔迹数据
 */
public class StrokeCanvas extends View {

    private static final String TAG = "WritechStrokeCanvas";

    /* 默认画笔颜色 */
    private static final int DEFAULT_STROKE_COLOR = Color.BLACK;

    /* 默认最小线宽（像素） */
    private static final float MIN_STROKE_WIDTH = 1.5f;

    /* 默认最大线宽（像素） */
    private static final float MAX_STROKE_WIDTH = 8.0f;

    /* 最大压力值（点阵笔12位ADC） */
    private static final float MAX_PRESSURE = 4095.0f;

    /* ===== 内部数据结构 ===== */

    /** 单个采样点（包含坐标、压力、时间戳） */
    private static class StrokePoint {
        float x;
        float y;
        float pressure;    /* 归一化压力 0.0~1.0 */
        long timestamp;    /* 毫秒时间戳 */

        StrokePoint(float x, float y, float pressure, long timestamp) {
            this.x = x;
            this.y = y;
            this.pressure = pressure;
            this.timestamp = timestamp;
        }
    }

    /** 一笔数据（从落笔到抬笔） */
    private static class Stroke {
        String penMac;    /* 来源笔MAC地址 */
        int color;        /* 笔迹颜色 */
        List<StrokePoint> points;    /* 采样点列表 */

        Stroke(String penMac, int color) {
            this.penMac = penMac;
            this.color = color;
            this.points = new ArrayList<>();
        }
    }

    /* ===== 成员变量 ===== */

```

```

/* 离屏缓冲Bitmap (双缓冲渲染) */
private Bitmap mBufferBitmap;
private Canvas mBufferCanvas;

/* 绘制画笔 */
private final Paint mStrokePaint;

/* 背景清除画笔 */
private final Paint mClearPaint;

/* 已完成的笔画列表 (历史记录) */
private final List<Stroke> mCompletedStrokes = new ArrayList<>();

/* 当前正在书写的笔画 (按笔MAC索引) */
private final Map<String, Stroke> mActiveStrokes = new HashMap<>();

/* 每支笔的颜色映射 */
private final Map<String, Integer> mPenColorMap = new HashMap<>();

/* 笔迹颜色分配计数器 */
private int mColorIndex = 0;

/* 预定义的笔迹颜色列表 (用于多学生区分) */
private static final int[] STROKE_COLORS = {
    Color.BLACK,
    Color.parseColor("#1565C0"),    /* 蓝色 */
    Color.parseColor("#C62828"),    /* 红色 */
    Color.parseColor("#2E7D32"),    /* 绿色 */
    Color.parseColor("#E65100"),    /* 橙色 */
    Color.parseColor("#6A1B9A"),    /* 紫色 */
    Color.parseColor("#00838F"),    /* 青色 */
    Color.parseColor("#4E342E"),    /* 棕色 */
};

/* 是否启用压力感应笔锋 */
private boolean mPressureEnabled = true;

/* 笔迹重播相关 */
private boolean mIsReplaying = false;
private int mReplayStrokeIndex = 0;
private int mReplayPointIndex = 0;
private long mReplayStartTime = 0;

/* ===== 构造函数 ===== */

public StrokeCanvas(Context context) {
    this(context, null);
}

public StrokeCanvas(Context context, AttributeSet attrs) {
    super(context, attrs);

    /* 初始化笔迹画笔 */
    mStrokePaint = new Paint();
    mStrokePaint.setAntiAlias(true);    /* 抗锯齿 */
    mStrokePaint.setDither(true);      /* 防抖动 */

```

```

        mStrokePaint.setStyle(Paint.Style.STROKE);
        mStrokePaint.setStrokeJoin(Paint.Join.ROUND);    /* 圆角连接 */
        mStrokePaint.setStrokeCap(Paint.Cap.ROUND);      /* 圆头笔触 */

        /* 初始化清除画笔 */
        mClearPaint = new Paint();
        mClearPaint.setColor(Color.WHITE);
    }

    /* ===== View生命周期 ===== */

    @Override
    protected void onSizeChanged(int w, int h, int oldw, int oldh) {
        super.onSizeChanged(w, h, oldw, oldh);

        /* 创建离屏缓冲Bitmap */
        if (mBufferBitmap != null) {
            mBufferBitmap.recycle();
        }
        mBufferBitmap = Bitmap.createBitmap(w, h, Bitmap.Config.ARGB_8888);
        mBufferCanvas = new Canvas(mBufferBitmap);
        mBufferCanvas.drawColor(Color.WHITE);

        /* 重绘所有历史笔画到缓冲区 */
        redrawAllStrokes();
    }

    @Override
    protected void onDraw(Canvas canvas) {
        /* 将离屏缓冲Bitmap绘制到屏幕 */
        if (mBufferBitmap != null) {
            canvas.drawBitmap(mBufferBitmap, 0, 0, null);
        }

        /* 绘制当前活跃的笔画（实时部分） */
        for (Stroke stroke : mActiveStrokes.values()) {
            drawStrokeRealtime(canvas, stroke);
        }
    }

    /* ===== 点阵笔数据输入接口 ===== */

    /**
     * 接收笔落下事件（开始新的一笔）
     * @param penMac 笔设备MAC地址
     */
    public void onPenDown(String penMac) {
        int color = getPenColor(penMac);
        Stroke stroke = new Stroke(penMac, color);
        mActiveStrokes.put(penMac, stroke);
    }

    /**
     * 接收笔迹坐标数据
     * @param penMac 笔MAC
     * @param screenX 屏幕X坐标（已经过坐标变换）
     * @param screenY 屏幕Y坐标

```



```

    * @param pressure 原始压力值 (0-4095)
    */
    public void onStrokePoint(String penMac, float screenX, float screenY,
                              int pressure) {
        Stroke stroke = mActiveStrokes.get(penMac);
        if (stroke == null) {
            /* 如果没有活跃笔画, 自动创建 */
            onPenDown(penMac);
            stroke = mActiveStrokes.get(penMac);
        }

        /* 归一化压力值 */
        float normalizedPressure = Math.min(1.0f, (float) pressure / MAX_PRESSURE);
        long timestamp = SystemClock.elapsedRealtime();

        stroke.points.add(new StrokePoint(screenX, screenY, normalizedPressure,
            timestamp));

        /* 触发重绘 (仅绘制增量部分, 避免全量刷新) */
        int pointCount = stroke.points.size();
        if (pointCount >= 2) {
            StrokePoint prev = stroke.points.get(pointCount - 2);
            StrokePoint curr = stroke.points.get(pointCount - 1);

            /* 仅刷新受影响的矩形区域 (性能优化) */
            float padding = MAX_STROKE_WIDTH + 2;
            float left = Math.min(prev.x, curr.x) - padding;
            float top = Math.min(prev.y, curr.y) - padding;
            float right = Math.max(prev.x, curr.x) + padding;
            float bottom = Math.max(prev.y, curr.y) + padding;

            invalidate((int) left, (int) top, (int) right, (int) bottom);
        }
    }

    /**
     * 接收笔抬起事件 (一笔结束)
     * 将当前笔画固化到缓冲区并归档
     */
    public void onPenUp(String penMac) {
        Stroke stroke = mActiveStrokes.remove(penMac);
        if (stroke != null && stroke.points.size() > 1) {
            /* 绘制到离屏缓冲区 (固化) */
            drawStrokeToBuffer(stroke);
            /* 添加到已完成列表 */
            mCompletedStrokes.add(stroke);
        }
        invalidate();
    }

    /* ===== 笔迹渲染核心算法 ===== */

    /**
     * 实时渲染笔画 (使用贝塞尔曲线平滑)
     * 在每次onDraw中调用, 绘制当前活跃的笔画
     */
    private void drawStrokeRealtime(Canvas canvas, Stroke stroke) {

```

```

List<StrokePoint> points = stroke.points;
if (points.size() < 2) return;

mStrokePaint.setColor(stroke.color);

for (int i = 1; i < points.size(); i++) {
    StrokePoint p0 = points.get(i - 1);
    StrokePoint p1 = points.get(i);

    /* 根据压力计算线宽 */
    float width = calculateStrokeWidth(p0.pressure, p1.pressure);
    mStrokePaint.setStrokeWidth(width);

    if (i >= 2) {
        /* 使用二次贝塞尔曲线平滑绘制 */
        StrokePoint pPrev = points.get(i - 2);
        float midX0 = (pPrev.x + p0.x) / 2;
        float midY0 = (pPrev.y + p0.y) / 2;
        float midX1 = (p0.x + p1.x) / 2;
        float midY1 = (p0.y + p1.y) / 2;

        Path path = new Path();
        path.moveTo(midX0, midY0);
        path.quadTo(p0.x, p0.y, midX1, midY1);
        canvas.drawPath(path, mStrokePaint);
    } else {
        /* 前两个点直接画直线 */
        canvas.drawLine(p0.x, p0.y, p1.x, p1.y, mStrokePaint);
    }
}

/**
 * 将完成的笔画绘制到离屏缓冲区
 */
private void drawStrokeToBuffer(Stroke stroke) {
    if (mBufferCanvas == null) return;
    drawStrokeRealtime(mBufferCanvas, stroke);
}

/**
 * 根据压力值计算线宽（笔锋效果）
 * 使用两个相邻点的平均压力，平滑过渡
 *
 * @param pressure0 前一点压力（归一化）
 * @param pressure1 当前点压力（归一化）
 * @return 线宽（像素）
 */
private float calculateStrokeWidth(float pressure0, float pressure1) {
    if (!mPressureEnabled) {
        return (MIN_STROKE_WIDTH + MAX_STROKE_WIDTH) / 2;
    }

    float avgPressure = (pressure0 + pressure1) / 2.0f;

    /* 压力-宽度映射曲线（使用幂函数增加笔锋感） */
    float normalized = (float) Math.pow(avgPressure, 0.7);

```

```

        return MIN_STROKE_WIDTH + normalized * (MAX_STROKE_WIDTH - MIN_STROKE_WIDTH);
    }

    /* ===== 多笔颜色管理 ===== */

    /** 获取或分配笔的颜色 */
    private int getPenColor(String penMac) {
        Integer color = mPenColorMap.get(penMac);
        if (color == null) {
            color = STROKE_COLORS[mColorIndex % STROKE_COLORS.length];
            mPenColorMap.put(penMac, color);
            mColorIndex++;
        }
        return color;
    }

    /** 手动设置某支笔的颜色 */
    public void setPenColor(String penMac, int color) {
        mPenColorMap.put(penMac, color);
    }

    /* ===== 画布操作 ===== */

    /** 清除所有笔迹 */
    public void clearAll() {
        mCompletedStrokes.clear();
        mActiveStrokes.clear();
        if (mBufferCanvas != null) {
            mBufferCanvas.drawColor(Color.WHITE);
        }
        invalidate();
    }

    /** 撤销最后一笔 */
    public boolean undo() {
        if (mCompletedStrokes.isEmpty()) return false;
        mCompletedStrokes.remove(mCompletedStrokes.size() - 1);
        redrawAllStrokes();
        invalidate();
        return true;
    }

    /** 重绘所有历史笔画到缓冲区 */
    private void redrawAllStrokes() {
        if (mBufferCanvas == null) return;
        mBufferCanvas.drawColor(Color.WHITE);
        for (Stroke stroke : mCompletedStrokes) {
            drawStrokeToBuffer(stroke);
        }
    }

    /** 导出当前画布为Bitmap */
    public Bitmap exportBitmap() {
        Bitmap export = Bitmap.createBitmap(getWidth(), getHeight(),
        Bitmap.Config.ARGB_8888);
        Canvas exportCanvas = new Canvas(export);
        draw(exportCanvas);
    }

```

```

        return export;
    }

    /** 获取已完成的笔画数量 */
    public int getStrokeCount() {
        return mCompletedStrokes.size();
    }

    /** 设置是否启用压力笔锋效果 */
    public void setPressureEnabled(boolean enabled) {
        mPressureEnabled = enabled;
    }

    /** ===== 触摸屏输入支持 ===== */

    @Override
    public boolean onTouchEvent(MotionEvent event) {
        /* 使用"touch"作为虚拟笔MAC */
        String touchMac = "touch_input";

        switch (event.getAction()) {
            case MotionEvent.ACTION_DOWN:
                onPenDown(touchMac);
                onStrokePoint(touchMac, event.getX(), event.getY(),
                    (int)(event.getPressure() * MAX_PRESSURE));
                return true;

            case MotionEvent.ACTION_MOVE:
                /* 处理历史点 (Android会批量发送MOVE事件) */
                for (int i = 0; i < event.getHistorySize(); i++) {
                    onStrokePoint(touchMac,
                        event.getHistoricalX(i),
                        event.getHistoricalY(i),
                        (int)(event.getHistoricalPressure(i) * MAX_PRESSURE));
                }
                onStrokePoint(touchMac, event.getX(), event.getY(),
                    (int)(event.getPressure() * MAX_PRESSURE));
                return true;

            case MotionEvent.ACTION_UP:
                onStrokePoint(touchMac, event.getX(), event.getY(),
                    (int)(event.getPressure() * MAX_PRESSURE));
                onPenUp(touchMac);
                return true;
        }
        return super.onTouchEvent(event);
    }
}

```

android/WritechSDK.java

```

/*
 * 自然写互动课堂应用开发SDK软件 V1.0
 * WritechSDK - SDK初始化与鉴权入口

```

```

*
* 功能说明：
* 1. SDK全局初始化（配置加载、模块注册）
* 2. 应用鉴权（AppKey/AppSecret验证）
* 3. 各子模块生命周期管理
* 4. 全局配置管理（服务器地址、超时、日志级别）
* 5. SDK版本信息与功能授权查询
*/

package com.writech.sdk.android;

import android.content.Context;
import android.content.SharedPreferences;
import android.util.Log;

import java.io.IOException;
import java.util.concurrent.atomic.AtomicBoolean;

/**
 * 自然写SDK主入口类
 * 使用前必须先调用 init() 方法进行初始化和鉴权
 *
 * 典型使用流程：
 * 1. WritechSDK.init(context, config)
 * 2. WritechSDK.getInstance().getPenManager().startScan()
 * 3. WritechSDK.getInstance().getOCREngine().recognizeHandwriting(...)
 */
public class WritechSDK {

    private static final String TAG = "WritechSDK";

    /* SDK版本号 */
    public static final String SDK_VERSION = "1.0.0";

    /* SDK构建号 */
    public static final int SDK_BUILD = 100;

    /* 单例实例 */
    private static volatile WritechSDK sInstance;

    /* 是否已初始化 */
    private static final AtomicBoolean sInitialized = new AtomicBoolean(false);

    /* ===== 配置类 ===== */

    /** SDK初始化配置 */
    public static class Config {
        /** 云平台API地址 */
        public String cloudBaseUrl = "https://api.writech.com";

        /** SDK应用标识（从自然写开放平台获取） */
        public String appKey;

        /** SDK应用密钥 */
        public String appSecret;

        /** 离线OCR模型文件路径（可选） */

```

```

    public String offlineModelPath;

    /** 是否启用调试日志 */
    public boolean debugMode = false;

    /** 笔迹数据本地缓存目录 */
    public String cacheDir;

    /** BLE扫描超时时间（毫秒） */
    public int bleScanTimeout = 30000;

    /** 网关自动发现 */
    public boolean autoDiscoverGateway = true;

    /** 最大同时连接笔数 */
    public int maxPenConnections = 60;
}

/* ===== 成员变量 ===== */

private Context mContext;
private Config mConfig;

/* 各子模块实例 */
private PenManager mPenManager;
private StrokeCanvas mDefaultCanvas;
private OCREngine mOCREngine;
private GatewaySDK mGatewaySDK;
private CloudClient mCloudClient;

/* 鉴权状态 */
private boolean mIsAuthenticated = false;
private String mLicenseType;          /* 授权类型: trial/standard/enterprise */
private long mLicenseExpireTime;      /* 授权到期时间 */

/* 本地存储 */
private SharedPreferences mPrefs;

/* ===== 初始化入口 ===== */

/**
 * 初始化SDK（必须在使用任何功能前调用）
 *
 * @param context  Android上下文 (Application级别)
 * @param config    SDK配置
 * @return 初始化结果: true成功, false失败
 */
public static boolean init(Context context, Config config) {
    if (sInitialized.getAndSet(true)) {
        Log.w(TAG, "SDK已初始化, 忽略重复调用");
        return true;
    }

    if (context == null || config == null) {
        Log.e(TAG, "初始化失败: context或config为null");
        sInitialized.set(false);
        return false;
    }

```

```

    }

    if (config.appKey == null || config.appSecret == null) {
        Log.e(TAG, "初始化失败: appKey或appSecret未配置");
        sInitialized.set(false);
        return false;
    }

    sInstance = new WritechSDK();
    boolean success = sInstance.doInit(context, config);

    if (!success) {
        sInstance = null;
        sInitialized.set(false);
    }

    return success;
}

/** 获取SDK单例 */
public static WritechSDK getInstance() {
    if (sInstance == null) {
        throw new IllegalStateException("WritechSDK未初始化, 请先调用
WritechSDK.init()");
    }
    return sInstance;
}

/** 检查SDK是否已初始化 */
public static boolean isInitialized() {
    return sInitialized.get();
}

/* ===== 内部初始化流程 ===== */

/** 执行具体的初始化逻辑 */
private boolean doInit(Context context, Config config) {
    mContext = context.getApplicationContext();
    mConfig = config;
    mPrefs = mContext.getSharedPreferences("writech_sdk", Context.MODE_PRIVATE);

    Log.i(TAG, "=== 自然写SDK V" + SDK_VERSION + " 初始化开始 ===");
    Log.i(TAG, "云平台地址: " + config.cloudBaseUrl);
    Log.i(TAG, "AppKey: " + config.appKey.substring(0, 8) + "****");
    Log.i(TAG, "调试模式: " + config.debugMode);

    /* 步骤1: 应用鉴权 (验证AppKey和AppSecret) */
    if (!authenticate(config.appKey, config.appSecret)) {
        Log.e(TAG, "SDK鉴权失败, 请检查AppKey和AppSecret");
        return false;
    }

    /* 步骤2: 初始化云平台客户端 */
    mCloudClient = new CloudClient(config.cloudBaseUrl, config.appKey,
config.appSecret);

    /* 恢复本地缓存的令牌 */

```

```

        restoreTokens();

        /* 步骤3: 初始化蓝牙笔管理器 */
        mPenManager = new PenManager(mContext);

        /* 步骤4: 初始化OCR引擎 */
        mOCREngine = new OCREngine(mContext, config.cloudBaseUrl, null);
        if (config.offlineModelPath != null) {
            mOCREngine.loadOfflineModel(config.offlineModelPath);
        }

        /* 步骤5: 初始化网关SDK */
        mGatewaySDK = new GatewaySDK(mContext);
        if (config.autoDiscoverGateway) {
            mGatewaySDK.startDiscovery();
        }

        Log.i(TAG, "=== 自然写SDK初始化完成 ===");
        return true;
    }

    /* ===== 应用鉴权 ===== */

    /**
     * 验证AppKey和AppSecret的有效性
     * 首次验证需要联网, 之后缓存鉴权结果
     */
    private boolean authenticate(String appKey, String appSecret) {
        /* 检查本地缓存的鉴权结果 */
        String cachedLicense = mPrefs.getString("license_type", null);
        long cachedExpire = mPrefs.getLong("license_expire", 0);

        if (cachedLicense != null && cachedExpire > System.currentTimeMillis()) {
            mIsAuthenticated = true;
            mLicenseType = cachedLicense;
            mLicenseExpireTime = cachedExpire;
            Log.i(TAG, "使用缓存鉴权结果: " + mLicenseType
                + ", 到期: " + new java.util.Date(mLicenseExpireTime));
            return true;
        }

        /* 在线鉴权 */
        try {
            String authUrl = mConfig.cloudBaseUrl + "/api/v1/sdk/authenticate";
            String body = "{\"appKey\":\"" + appKey
                + "\", \"appSecret\":\"" + appSecret
                + "\", \"sdkVersion\":\"" + SDK_VERSION + "\"}";

            /* 使用CloudClient的静态方法发送无认证请求 */
            java.net.HttpURLConnection conn =
                (java.net.HttpURLConnection) new java.net.URL(authUrl).openConnection();
            conn.setRequestMethod("POST");
            conn.setRequestProperty("Content-Type", "application/json");
            conn.setDoOutput(true);
            conn.setConnectTimeout(10000);

            conn.getOutputStream().write(body.getBytes(java.nio.charset.StandardCharsets.UTF_8));

```



```

        int responseCode = conn.getResponseCode();
        if (responseCode == 200) {
            java.io.InputStream is = conn.getInputStream();
            java.io.ByteArrayOutputStream baos = new
java.io.ByteArrayOutputStream();
            byte[] buf = new byte[1024];
            int len;
            while ((len = is.read(buf)) != -1) {
                baos.write(buf, 0, len);
            }
            String response = baos.toString("UTF-8");
            is.close();
            conn.disconnect();

            /* 解析鉴权结果 */
            mLicenseType = extractJsonField(response, "licenseType");
            String expireStr = extractJsonField(response, "expireTime");
            if (mLicenseType != null) {
                mLicenseExpireTime = expireStr != null ? Long.parseLong(expireStr)
                    : System.currentTimeMillis() + 365L * 24 * 3600
* 1000;

                mIsAuthenticated = true;

                /* 缓存鉴权结果 */
                mPrefs.edit()
                    .putString("license_type", mLicenseType)
                    .putLong("license_expire", mLicenseExpireTime)
                    .apply();

                Log.i(TAG, "在线鉴权成功: " + mLicenseType);
                return true;
            }
        }
        conn.disconnect();

    } catch (Exception e) {
        Log.w(TAG, "在线鉴权异常: " + e.getMessage());
        /* 联网失败时允许离线试用 (7天) */
        mLicenseType = "trial";
        mLicenseExpireTime = System.currentTimeMillis() + 7L * 24 * 3600 * 1000;
        mIsAuthenticated = true;
        Log.i(TAG, "离线模式, 试用授权7天");
        return true;
    }

    return false;
}

/** 恢复本地缓存的认证令牌 */
private void restoreTokens() {
    String accessToken = mPrefs.getString("access_token", null);
    String refreshToken = mPrefs.getString("refresh_token", null);
    long expireTime = mPrefs.getLong("token_expire", 0);

    if (accessToken != null && refreshToken != null) {
        mCloudClient.setTokens(accessToken, refreshToken, expireTime);
    }
}

```

```

        Log.d(TAG, "已恢复缓存的认证令牌");
    }
}

/* ===== 对外接口 ===== */

/** 获取笔管理器 */
public PenManager getPenManager() {
    return mPenManager;
}

/** 获取OCR引擎 */
public OCREngine getOCREngine() {
    return mOCREngine;
}

/** 获取网关SDK */
public GatewaySDK getGatewaySDK() {
    return mGatewaySDK;
}

/** 获取云平台客户端 */
public CloudClient getCloudClient() {
    return mCloudClient;
}

/** 获取SDK版本 */
public String getVersion() {
    return SDK_VERSION;
}

/** 获取授权类型 */
public String getLicenseType() {
    return mLicenseType;
}

/** 检查是否已鉴权 */
public boolean isAuthenticated() {
    return mIsAuthenticated;
}

/** 用户登录（通过云平台认证） */
public boolean loginUser(String username, String password) {
    try {
        String response = mCloudClient.login(username, password);
        String accessToken = extractJsonField(response, "accessToken");
        String refreshToken = extractJsonField(response, "refreshToken");

        if (accessToken != null) {
            long expireTime = System.currentTimeMillis() + 30 * 60 * 1000;
            mCloudClient.setTokens(accessToken, refreshToken, expireTime);

            /* 缓存令牌 */
            mPrefs.edit()
                .putString("access_token", accessToken)
                .putString("refresh_token", refreshToken)
                .putLong("token_expire", expireTime)
    }
}

```

```

        .apply();

        return true;
    }
} catch (IOException e) {
    Log.e(TAG, "登录失败: " + e.getMessage());
}
return false;
}

/* ===== 资源释放 ===== */

/** 释放SDK所有资源 */
public static void destroy() {
    if (sInstance != null) {
        if (sInstance.mGatewaySDK != null) sInstance.mGatewaySDK.destroy();
        if (sInstance.mOCREngine != null) sInstance.mOCREngine.destroy();
        if (sInstance.mPenManager != null) sInstance.mPenManager.destroy();
        sInstance = null;
    }
    sInitialized.set(false);
    Log.i(TAG, "WritechSDK已释放所有资源");
}

/** 从JSON提取字段值 */
private String extractJsonField(String json, String key) {
    if (json == null) return null;
    String search = "\"" + key + "\"";
    int idx = json.indexOf(search);
    if (idx < 0) return null;
    int start = json.indexOf("\"", idx + search.length() + 1) + 1;
    int end = json.indexOf("\"", start);
    return (start > 0 && end > start) ? json.substring(start, end) : null;
}
}

```

core/

core/ble_protocol.c

```

/**
 * 自然写互动课堂应用开发SDK软件 V1.0
 * BLE协议解析核心模块 - 蓝牙5.0点阵笔通信协议实现
 *
 * 跨平台C语言核心库，负责解析点阵笔BLE GATT数据
 * 提供笔迹坐标解包、协议帧校验、数据压缩解压等底层能力
 * 通过JNI/ObjC Bridge/FFI供各平台SDK调用
 */

#ifndef BLE_PROTOCOL_H
#define BLE_PROTOCOL_H

#include <stdint.h>

```

```

#include <stddef.h>
#include <string.h>

#ifdef __cplusplus
extern "C" {
#endif

/* ===== 协议常量定义 ===== */

/* BLE GATT Service UUID (自定义服务) */
#define WRITECH_SERVICE_UUID "0000FFE0-0000-1000-8000-00805F9B34FB"
/* 笔迹数据Characteristic UUID */
#define STROKE_DATA_CHAR_UUID "0000FFE1-0000-1000-8000-00805F9B34FB"
/* 设备信息Characteristic UUID */
#define DEVICE_INFO_CHAR_UUID "0000FFE2-0000-1000-8000-00805F9B34FB"
/* 配置写入Characteristic UUID */
#define CONFIG_WRITE_CHAR_UUID "0000FFE3-0000-1000-8000-00805F9B34FB"
/* OTA DFU Characteristic UUID */
#define OTA_DFU_CHAR_UUID "0000FFE4-0000-1000-8000-00805F9B34FB"

/* 协议帧标志 */
#define FRAME_HEADER_MAGIC 0xAA55
#define FRAME_MAX_PAYLOAD_SIZE 240 /* MTU=247, 减去帧头7字节 */
#define MAX_POINTS_PER_FRAME 34 /* 每帧最多34个坐标点 */

/* 帧类型定义 */
#define FRAME_TYPE_STROKE_DATA 0x01 /* 笔迹坐标数据 */
#define FRAME_TYPE_PEN_UP 0x02 /* 抬笔事件 */
#define FRAME_TYPE_PEN_DOWN 0x03 /* 落笔事件 */
#define FRAME_TYPE_DEVICE_STATUS 0x04 /* 设备状态 (电量等) */
#define FRAME_TYPE_OFFLINE_SYNC 0x05 /* 离线数据同步 */
#define FRAME_TYPE_OTA_DATA 0x06 /* OTA升级数据 */
#define FRAME_TYPE_CONFIG_RSP 0x07 /* 配置响应 */

/* ===== 数据结构定义 ===== */

/**
 * 原始笔迹坐标点 (7字节紧凑编码)
 * x: 16位无符号整数, 点阵坐标X (分辨率约300DPI)
 * y: 16位无符号整数, 点阵坐标Y
 * pressure: 8位无符号整数, 压力值(0-255)
 * timestamp_delta: 16位无符号整数, 距上一点的时间差 (毫秒)
 */
typedef struct {
    uint16_t x; /* X坐标 (大端序) */
    uint16_t y; /* Y坐标 (大端序) */
    uint8_t pressure; /* 压力值 0-255 */
    uint16_t timestamp_delta; /* 时间增量 (毫秒) */
} __attribute__((packed)) StrokePointRaw;

/**
 * 解码后的笔迹坐标点
 */
typedef struct {
    float x; /* X坐标 (浮点) */
    float y; /* Y坐标 (浮点) */
    float pressure; /* 压力值 0.0-1.0 */

```

```

        uint32_t timestamp;          /* 绝对时间戳 (毫秒) */
        uint8_t pen_state;           /* 0=落笔, 1=抬笔 */
    } StrokePoint;

/**
 * BLE协议帧头 (7字节)
 */
typedef struct {
    uint16_t magic;                  /* 帧头魔数 0xAA55 */
    uint8_t frame_type;              /* 帧类型 */
    uint8_t sequence;               /* 帧序号(0-255循环) */
    uint16_t payload_length;         /* 负载长度 */
    uint8_t checksum;               /* 帧头校验和(XOR) */
} __attribute__((packed)) FrameHeader;

/**
 * 笔迹数据帧
 */
typedef struct {
    FrameHeader header;
    uint8_t point_count;             /* 本帧包含的坐标点数 */
    uint32_t page_id;               /* 点阵码页面ID */
    StrokePointRaw points[MAX_POINTS_PER_FRAME]; /* 坐标点数组 */
    uint16_t crc16;                 /* CRC-16校验 */
} __attribute__((packed)) StrokeDataFrame;

/**
 * 设备状态帧
 */
typedef struct {
    FrameHeader header;
    uint8_t battery_level;          /* 电量百分比 0-100 */
    uint8_t charging_state;         /* 充电状态: 0=未充电, 1=充电中, 2=已充满 */
    uint16_t firmware_version;      /* 固件版本 (major*256+minor) */
    uint8_t connection_state;       /* 连接状态 */
    uint32_t serial_number;         /* 设备序列号 */
    uint16_t crc16;
} __attribute__((packed)) DeviceStatusFrame;

/**
 * 解析回调函数类型定义
 */
typedef void (*on_stroke_point_cb)(const StrokePoint* point, void* user_data);
typedef void (*on_pen_event_cb)(uint8_t event_type, uint32_t timestamp, void* user_data);
typedef void (*on_device_status_cb)(uint8_t battery, uint8_t charging, uint16_t fw_ver, void* user_data);

/* ===== 协议解析器 ===== */

/**
 * BLE协议解析器上下文
 */
typedef struct {
    /* 接收缓冲区 (处理分包/粘包) */
    uint8_t recv_buffer[512];
    size_t recv_length;

```

```

    /* 序号跟踪 (乱序检测) */
    uint8_t expected_sequence;

    /* 时间戳基准 */
    uint32_t base_timestamp;
    uint32_t last_timestamp;

    /* 统计信息 */
    uint32_t total_frames;
    uint32_t total_points;
    uint32_t error_frames;
    uint32_t lost_frames;

    /* 回调函数 */
    on_stroke_point_cb stroke_cb;
    on_pen_event_cb pen_event_cb;
    on_device_status_cb status_cb;
    void* user_data;
} BleProtocolParser;

/**
 * 初始化协议解析器
 */
static inline void ble_parser_init(BleProtocolParser* parser) {
    memset(parser, 0, sizeof(BleProtocolParser));
    parser->expected_sequence = 0;
    parser->base_timestamp = 0;
}

/**
 * 设置回调函数
 */
static inline void ble_parser_set_callbacks(
    BleProtocolParser* parser,
    on_stroke_point_cb stroke_cb,
    on_pen_event_cb pen_event_cb,
    on_device_status_cb status_cb,
    void* user_data
) {
    parser->stroke_cb = stroke_cb;
    parser->pen_event_cb = pen_event_cb;
    parser->status_cb = status_cb;
    parser->user_data = user_data;
}

/**
 * 计算CRC-16校验值 (CCITT标准)
 */
static uint16_t calc_crc16(const uint8_t* data, size_t length) {
    uint16_t crc = 0xFFFF;
    for (size_t i = 0; i < length; i++) {
        crc ^= (uint16_t)data[i] << 8;
        for (int j = 0; j < 8; j++) {
            if (crc & 0x8000)
                crc = (crc << 1) ^ 0x1021;
            else

```

```

        crc <= 1;
    }
}
return crc;
}

/**
 * 校验帧头
 */
static int validate_frame_header(const FrameHeader* header) {
    /* 校验魔数 */
    if (header->magic != FRAME_HEADER_MAGIC) return -1;
    /* 校验负载长度 */
    if (header->payload_length > FRAME_MAX_PAYLOAD_SIZE) return -2;
    /* 校验帧头XOR校验和 */
    uint8_t xor_sum = 0;
    const uint8_t* p = (const uint8_t*)header;
    for (int i = 0; i < 6; i++) xor_sum ^= p[i];
    if (xor_sum != header->checksum) return -3;
    return 0;
}

/**
 * 大端序转小端序 (16位)
 */
static inline uint16_t be16_to_le(uint16_t value) {
    return (value >> 8) | (value << 8);
}

/**
 * 解析笔迹数据帧
 * 从帧中提取坐标点并通过回调函数输出
 */
static int parse_stroke_frame(BleProtocolParser* parser, const uint8_t* data, size_t
length) {
    if (length < sizeof(FrameHeader) + 5) return -1;

    const FrameHeader* header = (const FrameHeader*)data;

    /* 帧头校验 */
    if (validate_frame_header(header) != 0) {
        parser->error_frames++;
        return -1;
    }

    /* 序号连续性检查 */
    if (header->sequence != parser->expected_sequence) {
        uint8_t lost = header->sequence - parser->expected_sequence;
        parser->lost_frames += lost;
    }
    parser->expected_sequence = header->sequence + 1;

    /* 解析负载 */
    const uint8_t* payload = data + sizeof(FrameHeader);
    uint8_t point_count = payload[0];
    uint32_t page_id = *(uint32_t*)(payload + 1);

```

```

    if (point_count > MAX_POINTS_PER_FRAME) {
        parser->error_frames++;
        return -2;
    }

    /* CRC校验 (校验帧头+负载) */
    size_t crc_data_len = length - 2;
    uint16_t expected_crc = *(uint16_t*)(data + crc_data_len);
    uint16_t actual_crc = calc_crc16(data, crc_data_len);
    if (expected_crc != actual_crc) {
        parser->error_frames++;
        return -3;
    }

    /* 解析每个坐标点 */
    const StrokePointRaw* raw_points = (const StrokePointRaw*)(payload + 5);
    for (int i = 0; i < point_count; i++) {
        StrokePoint decoded;
        decoded.x = (float)be16_to_le(raw_points[i].x);
        decoded.y = (float)be16_to_le(raw_points[i].y);
        decoded.pressure = raw_points[i].pressure / 255.0f;

        /* 累加时间增量得到绝对时间戳 */
        uint16_t delta = be16_to_le(raw_points[i].timestamp_delta);
        parser->last_timestamp += delta;
        decoded.timestamp = parser->base_timestamp + parser->last_timestamp;
        decoded.pen_state = 0; /* 落笔状态 */

        /* 通过回调函数输出 */
        if (parser->stroke_cb) {
            parser->stroke_cb(&decoded, parser->user_data);
        }
        parser->total_points++;
    }

    parser->total_frames++;
    return point_count;
}

/**
 * 输入BLE Notify接收到的数据
 * 处理分包/粘包，自动检测帧边界并分发解析
 */
static int ble_parser_feed(BleProtocolParser* parser, const uint8_t* data, size_t
length) {
    /* 追加到接收缓冲区 */
    if (parser->recv_length + length > sizeof(parser->recv_buffer)) {
        /* 缓冲区溢出，丢弃旧数据 */
        parser->recv_length = 0;
    }
    memcpy(parser->recv_buffer + parser->recv_length, data, length);
    parser->recv_length += length;

    int parsed_count = 0;

    /* 扫描缓冲区查找完整帧 */
    while (parser->recv_length >= sizeof(FrameHeader)) {

```



```

/* 查找帧头魔数 */
if (parser->recv_buffer[0] != 0xAA || parser->recv_buffer[1] != 0x55) {
    /* 跳过非法字节 */
    memmove(parser->recv_buffer, parser->recv_buffer + 1, parser->recv_length -
1);

    parser->recv_length--;
    continue;
}

FrameHeader* header = (FrameHeader*)parser->recv_buffer;
size_t frame_size = sizeof(FrameHeader) + header->payload_length + 2; /* +2 for
CRC */

if (parser->recv_length < frame_size) {
    break; /* 帧数据不完整, 等待更多数据 */
}

/* 根据帧类型分发解析 */
switch (header->frame_type) {
    case FRAME_TYPE_STROKE_DATA:
        parse_stroke_frame(parser, parser->recv_buffer, frame_size);
        parsed_count++;
        break;
    case FRAME_TYPE_PEN_UP:
        if (parser->pen_event_cb) {
            parser->pen_event_cb(1, parser->last_timestamp, parser->user_data);
        }
        break;
    case FRAME_TYPE_PEN_DOWN:
        if (parser->pen_event_cb) {
            parser->pen_event_cb(0, parser->last_timestamp, parser->user_data);
        }
        break;
    case FRAME_TYPE_DEVICE_STATUS: {
        DeviceStatusFrame* status = (DeviceStatusFrame*)parser->recv_buffer;
        if (parser->status_cb) {
            parser->status_cb(status->battery_level, status->charging_state,
                status->firmware_version, parser->user_data);
        }
        break;
    }
    default:
        break;
}

/* 移除已处理的帧 */
memmove(parser->recv_buffer, parser->recv_buffer + frame_size,
    parser->recv_length - frame_size);
parser->recv_length -= frame_size;
}

return parsed_count;
}

/**
 * 获取解析器统计信息
 */

```

```

static inline void ble_parser_get_stats(const BleProtocolParser* parser,
    uint32_t* total_frames, uint32_t* total_points,
    uint32_t* error_frames, uint32_t* lost_frames) {
    if (total_frames) *total_frames = parser->total_frames;
    if (total_points) *total_points = parser->total_points;
    if (error_frames) *error_frames = parser->error_frames;
    if (lost_frames) *lost_frames = parser->lost_frames;
}

/**
 * 重置解析器状态
 */
static inline void ble_parser_reset(BleProtocolParser* parser) {
    parser->recv_length = 0;
    parser->expected_sequence = 0;
    parser->last_timestamp = 0;
    parser->total_frames = 0;
    parser->total_points = 0;
    parser->error_frames = 0;
    parser->lost_frames = 0;
}

#ifdef __cplusplus
}
#endif

#endif /* BLE_PROTOCOL_H */

```

core/coordinate_transform.c

```

/*
 * 自然写互动课堂应用开发SDK软件 V1.0
 * 坐标变换模块 - 点阵笔坐标到屏幕坐标的高精度映射
 *
 * 功能说明:
 * 1. 点阵码坐标解析与标准化 (Anoto编码 → 物理坐标mm)
 * 2. 仿射变换矩阵计算 (四角标定点 → 变换参数)
 * 3. 物理坐标到屏幕像素坐标的实时映射
 * 4. 多页面坐标空间管理 (不同纸张/不同页面独立坐标系)
 * 5. 畸变校正 (镜头畸变、纸张弯曲补偿)
 */

#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

/* ===== 数据结构定义 ===== */

/* 二维点 (浮点精度) */
typedef struct {
    double x;      /* X坐标 */
    double y;      /* Y坐标 */
} Point2D;

```

```

/* 仿射变换矩阵 3x3 (齐次坐标) */
typedef struct {
    double m[3][3]; /* 变换矩阵元素 */
} AffineMatrix;

/* 坐标空间描述 */
typedef struct {
    unsigned int page_id; /* 页面唯一ID */
    unsigned int section_id; /* 区段ID (Anoto编码中的section) */
    unsigned int owner_id; /* 拥有者ID (Anoto编码) */
    double physical_width_mm; /* 纸张物理宽度 (毫米) */
    double physical_height_mm; /* 纸张物理高度 (毫米) */
    int screen_width_px; /* 对应屏幕区域宽度 (像素) */
    int screen_height_px; /* 对应屏幕区域高度 (像素) */
    AffineMatrix transform; /* 标定后的变换矩阵 */
    int is_calibrated; /* 是否已完成标定 */
} CoordinateSpace;

/* 标定点对 (物理坐标 ↔ 屏幕坐标) */
typedef struct {
    Point2D physical; /* 物理坐标 (mm) */
    Point2D screen; /* 屏幕坐标 (px) */
} CalibrationPair;

/* 畸变校正参数 (Brown-Conrady模型简化版) */
typedef struct {
    double k1; /* 径向畸变系数1 */
    double k2; /* 径向畸变系数2 */
    double p1; /* 切向畸变系数1 */
    double p2; /* 切向畸变系数2 */
    double cx; /* 畸变中心X */
    double cy; /* 畸变中心Y */
} DistortionParams;

/* 坐标变换管理器 */
typedef struct {
    CoordinateSpace spaces[64]; /* 最多支持64个坐标空间 */
    int space_count; /* 当前已注册的空间数 */
    DistortionParams distortion; /* 全局畸变校正参数 */
    int distortion_enabled; /* 是否启用畸变校正 */
    double dpi_resolution; /* 点阵笔DPI分辨率 (通常为300或600) */
} CoordinateManager;

/* 全局坐标管理器实例 */
static CoordinateManager g_coord_manager;

/* ===== Anoto点阵码坐标解析 ===== */

/*
 * 将Anoto点阵码原始编码转换为物理坐标 (毫米)
 * 点阵笔采集到的原始数据是基于Anoto编码系统的逻辑坐标
 * 需要根据DPI分辨率转换为实际的物理距离
 *
 * @param raw_x 点阵码原始X坐标值
 * @param raw_y 点阵码原始Y坐标值
 * @param section_id Anoto编码的section标识

```

```

* @param out_physical 输出的物理坐标 (mm)
* @return 0成功, -1参数错误
*/
int anoto_to_physical(unsigned int raw_x, unsigned int raw_y,
                     unsigned int section_id, Point2D *out_physical) {
    if (out_physical == NULL) {
        return -1;
    }

    /* DPI到毫米的转换因子: 25.4mm / DPI */
    double dpi = g_coord_manager.dpi_resolution;
    if (dpi < 1.0) {
        dpi = 300.0; /* 默认300 DPI */
    }
    double dots_to_mm = 25.4 / dpi;

    /* Anoto编码的原始坐标直接乘以转换因子得到物理坐标 */
    out_physical->x = (double)raw_x * dots_to_mm;
    out_physical->y = (double)raw_y * dots_to_mm;

    return 0;
}

/*
* 解析7字节紧凑坐标编码
* 点阵笔通过BLE传输时使用7字节紧凑格式:
*   字节0-1: X坐标高16位
*   字节2-3: Y坐标高16位
*   字节4:   X低4位 | Y低4位
*   字节5:   压力值高8位
*   字节6:   压力值低8位 | 标志位
*/
int decode_compact_coordinate(const unsigned char *data, int data_len,
                             unsigned int *out_x, unsigned int *out_y,
                             unsigned int *out_pressure) {
    if (data == NULL || data_len < 7) {
        return -1;
    }

    /* 解析X坐标 (20位精度) */
    unsigned int x_high = ((unsigned int)data[0] << 8) | data[1];
    unsigned int x_low = (data[4] >> 4) & 0x0F;
    *out_x = (x_high << 4) | x_low;

    /* 解析Y坐标 (20位精度) */
    unsigned int y_high = ((unsigned int)data[2] << 8) | data[3];
    unsigned int y_low = data[4] & 0x0F;
    *out_y = (y_high << 4) | y_low;

    /* 解析压力值 (12位精度, 0-4095) */
    unsigned int p_high = data[5];
    unsigned int p_low = (data[6] >> 4) & 0x0F;
    *out_pressure = (p_high << 4) | p_low;

    return 0;
}

```

```

/* ===== 仿射变换矩阵计算 ===== */

/*
 * 初始化为单位矩阵
 */
void matrix_identity(AffineMatrix *mat) {
    memset(mat->m, 0, sizeof(mat->m));
    mat->m[0][0] = 1.0;
    mat->m[1][1] = 1.0;
    mat->m[2][2] = 1.0;
}

/*
 * 矩阵乘法 result = a * b
 */
void matrix_multiply(const AffineMatrix *a, const AffineMatrix *b,
                    AffineMatrix *result) {
    AffineMatrix tmp;
    int i, j, k;
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            tmp.m[i][j] = 0.0;
            for (k = 0; k < 3; k++) {
                tmp.m[i][j] += a->m[i][k] * b->m[k][j];
            }
        }
    }
    memcpy(result->m, tmp.m, sizeof(tmp.m));
}

/*
 * 使用最小二乘法从标定点对计算仿射变换矩阵
 * 至少需要3个不共线的标定点对
 * 使用正规方程法求解超定线性方程组
 */
/*
 * @param pairs      标定点对数组
 * @param pair_count 标定点对数量 (≥3)
 * @param out_matrix 输出的仿射变换矩阵
 * @return 0成功, -1参数不足, -2矩阵奇异
 */
int compute_affine_transform(const CalibrationPair *pairs, int pair_count,
                            AffineMatrix *out_matrix) {
    if (pairs == NULL || pair_count < 3 || out_matrix == NULL) {
        return -1;
    }

    /*
     * 仿射变换方程:
     * screen_x = a11 * phys_x + a12 * phys_y + a13
     * screen_y = a21 * phys_x + a22 * phys_y + a23
     *
     * 构建 ATA * x = ATb 正规方程
     * A矩阵每行: [phys_x, phys_y, 1]
     */
    double ATA[3][3] = {{0}};
    double ATb_x[3] = {0};
    double ATb_y[3] = {0};

```

```

int i;
for (i = 0; i < pair_count; i++) {
    double px = pairs[i].physical.x;
    double py = pairs[i].physical.y;
    double sx = pairs[i].screen.x;
    double sy = pairs[i].screen.y;

    /* 累加 ATA */
    ATA[0][0] += px * px;
    ATA[0][1] += px * py;
    ATA[0][2] += px;
    ATA[1][0] += py * px;
    ATA[1][1] += py * py;
    ATA[1][2] += py;
    ATA[2][0] += px;
    ATA[2][1] += py;
    ATA[2][2] += 1.0;

    /* 累加 ATb */
    ATb_x[0] += px * sx;
    ATb_x[1] += py * sx;
    ATb_x[2] += sx;

    ATb_y[0] += px * sy;
    ATb_y[1] += py * sy;
    ATb_y[2] += sy;
}

/* 高斯消元法求解3x3线性方程组 */
/* 先求解 screen_x 的系数 [a11, a12, a13] */
double aug_x[3][4];
double aug_y[3][4];
int j, k;
for (i = 0; i < 3; i++) {
    for (j = 0; j < 3; j++) {
        aug_x[i][j] = ATA[i][j];
        aug_y[i][j] = ATA[i][j];
    }
    aug_x[i][3] = ATb_x[i];
    aug_y[i][3] = ATb_y[i];
}

/* 高斯消元 (部分主元选取) */
for (k = 0; k < 3; k++) {
    /* 找主元 */
    int max_row = k;
    double max_val = fabs(aug_x[k][k]);
    for (i = k + 1; i < 3; i++) {
        if (fabs(aug_x[i][k]) > max_val) {
            max_val = fabs(aug_x[i][k]);
            max_row = i;
        }
    }
    if (max_val < 1e-12) {
        return -2; /* 矩阵奇异, 标定点可能共线 */
    }
}

```

```

/* 交换行 */
if (max_row != k) {
    for (j = 0; j < 4; j++) {
        double tmp = aug_x[k][j];
        aug_x[k][j] = aug_x[max_row][j];
        aug_x[max_row][j] = tmp;
        tmp = aug_y[k][j];
        aug_y[k][j] = aug_y[max_row][j];
        aug_y[max_row][j] = tmp;
    }
}

/* 消元 */
for (i = k + 1; i < 3; i++) {
    double factor_x = aug_x[i][k] / aug_x[k][k];
    double factor_y = aug_y[i][k] / aug_y[k][k];
    for (j = k; j < 4; j++) {
        aug_x[i][j] -= factor_x * aug_x[k][j];
        aug_y[i][j] -= factor_y * aug_y[k][j];
    }
}

/* 回代求解 */
double sol_x[3], sol_y[3];
for (i = 2; i >= 0; i--) {
    sol_x[i] = aug_x[i][3];
    sol_y[i] = aug_y[i][3];
    for (j = i + 1; j < 3; j++) {
        sol_x[i] -= aug_x[i][j] * sol_x[j];
        sol_y[i] -= aug_y[i][j] * sol_y[j];
    }
    sol_x[i] /= aug_x[i][i];
    sol_y[i] /= aug_y[i][i];
}

/* 填充仿射变换矩阵 */
out_matrix->m[0][0] = sol_x[0]; /* a11 */
out_matrix->m[0][1] = sol_x[1]; /* a12 */
out_matrix->m[0][2] = sol_x[2]; /* a13 (平移X) */
out_matrix->m[1][0] = sol_y[0]; /* a21 */
out_matrix->m[1][1] = sol_y[1]; /* a22 */
out_matrix->m[1][2] = sol_y[2]; /* a23 (平移Y) */
out_matrix->m[2][0] = 0.0;
out_matrix->m[2][1] = 0.0;
out_matrix->m[2][2] = 1.0;

return 0;
}

/* ===== 坐标空间管理 ===== */

/*
 * 初始化坐标变换管理器
 * @param dpi 点阵笔的DPI分辨率 (常见值: 300, 600)
 */
void coordinate_manager_init(double dpi) {
    memset(&g_coord_manager, 0, sizeof(g_coord_manager));
}

```

```

    g_coord_manager.dpi_resolution = dpi;
    g_coord_manager.distortion_enabled = 0;
}

/*
 * 注册一个新的坐标空间（对应一个页面/纸张）
 * 在使用特定页面前需先注册其坐标空间参数
 *
 * @param page_id      页面唯一标识
 * @param section_id   Anoto section编号
 * @param width_mm     纸张物理宽度
 * @param height_mm    纸张物理高度
 * @param screen_w     对应屏幕宽度像素
 * @param screen_h     对应屏幕高度像素
 * @return 空间索引, -1失败
 */
int register_coordinate_space(unsigned int page_id, unsigned int section_id,
                             double width_mm, double height_mm,
                             int screen_w, int screen_h) {
    if (g_coord_manager.space_count >= 64) {
        return -1;    /* 空间已满 */
    }

    int idx = g_coord_manager.space_count;
    CoordinateSpace *space = &g_coord_manager.spaces[idx];
    space->page_id = page_id;
    space->section_id = section_id;
    space->physical_width_mm = width_mm;
    space->physical_height_mm = height_mm;
    space->screen_width_px = screen_w;
    space->screen_height_px = screen_h;
    space->is_calibrated = 0;
    matrix_identity(&space->transform);

    g_coord_manager.space_count++;
    return idx;
}

/*
 * 对指定坐标空间执行标定
 * 使用用户提供的标定点对计算仿射变换矩阵
 */
int calibrate_space(int space_index, const CalibrationPair *pairs,
                    int pair_count) {
    if (space_index < 0 || space_index >= g_coord_manager.space_count) {
        return -1;
    }

    CoordinateSpace *space = &g_coord_manager.spaces[space_index];
    int ret = compute_affine_transform(pairs, pair_count, &space->transform);
    if (ret == 0) {
        space->is_calibrated = 1;
    }
    return ret;
}

/*

```



```

* 使用默认缩放（无旋转无畸变）进行快速标定
* 适用于标准A4纸张等无需精确标定的场景
*/
int calibrate_space_default(int space_index) {
    if (space_index < 0 || space_index >= g_coord_manager.space_count) {
        return -1;
    }

    CoordinateSpace *space = &g_coord_manager.spaces[space_index];
    matrix_identity(&space->transform);

    /* 简单线性缩放：物理mm → 屏幕px */
    double scale_x = (double)space->screen_width_px / space->physical_width_mm;
    double scale_y = (double)space->screen_height_px / space->physical_height_mm;

    space->transform.m[0][0] = scale_x;
    space->transform.m[1][1] = scale_y;
    space->is_calibrated = 1;

    return 0;
}

/* ===== 畸变校正 ===== */

/*
* 设置畸变校正参数
* 用于补偿摄像头镜头的径向和切向畸变
*/
void set_distortion_params(double k1, double k2, double p1, double p2,
                           double cx, double cy) {
    g_coord_manager.distortion.k1 = k1;
    g_coord_manager.distortion.k2 = k2;
    g_coord_manager.distortion.p1 = p1;
    g_coord_manager.distortion.p2 = p2;
    g_coord_manager.distortion.cx = cx;
    g_coord_manager.distortion.cy = cy;
    g_coord_manager.distortion_enabled = 1;
}

/*
* 对物理坐标应用畸变校正（去畸变）
* 使用Brown-Conrady模型的简化版本
*
* @param in    输入的物理坐标
* @param out   校正后的物理坐标
*/
void apply_distortion_correction(const Point2D *in, Point2D *out) {
    if (!g_coord_manager.distortion_enabled) {
        out->x = in->x;
        out->y = in->y;
        return;
    }

    DistortionParams *d = &g_coord_manager.distortion;

    /* 以畸变中心为原点 */
    double dx = in->x - d->cx;

```

```

double dy = in->y - d->cy;
double r2 = dx * dx + dy * dy;
double r4 = r2 * r2;

/* 径向畸变校正 */
double radial = 1.0 + d->k1 * r2 + d->k2 * r4;

/* 切向畸变校正 */
double tang_x = 2.0 * d->p1 * dx * dy + d->p2 * (r2 + 2.0 * dx * dx);
double tang_y = d->p1 * (r2 + 2.0 * dy * dy) + 2.0 * d->p2 * dx * dy;

out->x = d->cx + dx * radial + tang_x;
out->y = d->cy + dy * radial + tang_y;
}

/* ===== 坐标变换核心接口 ===== */

/*
 * 根据page_id查找对应的坐标空间索引
 */
int find_space_by_page(unsigned int page_id) {
    int i;
    for (i = 0; i < g_coord_manager.space_count; i++) {
        if (g_coord_manager.spaces[i].page_id == page_id) {
            return i;
        }
    }
    return -1;
}

/*
 * 完整坐标变换流水线：原始点阵码坐标 → 屏幕像素坐标
 *
 * 处理步骤：
 * 1. Anoto编码 → 物理坐标 (mm)
 * 2. 畸变校正 (如果启用)
 * 3. 仿射变换 → 屏幕坐标 (px)
 * 4. 边界裁剪 (确保不超出屏幕范围)
 *
 * @param raw_x      原始X坐标
 * @param raw_y      原始Y坐标
 * @param page_id    页面ID
 * @param out_screen 输出屏幕坐标
 * @return 0成功, -1未找到坐标空间, -2未标定
 */
int transform_coordinate(unsigned int raw_x, unsigned int raw_y,
                        unsigned int page_id, Point2D *out_screen) {
    if (out_screen == NULL) {
        return -1;
    }

    /* 查找坐标空间 */
    int idx = find_space_by_page(page_id);
    if (idx < 0) {
        return -1;
    }
}

```

```

CoordinateSpace *space = &g_coord_manager.spaces[idx];
if (!space->is_calibrated) {
    return -2;
}

/* 步骤1: 原始坐标 → 物理坐标 */
Point2D physical;
anoto_to_physical(raw_x, raw_y, space->section_id, &physical);

/* 步骤2: 畸变校正 */
Point2D corrected;
apply_distortion_correction(&physical, &corrected);

/* 步骤3: 仿射变换 → 屏幕坐标 */
AffineMatrix *mat = &space->transform;
out_screen->x = mat->m[0][0] * corrected.x
               + mat->m[0][1] * corrected.y
               + mat->m[0][2];
out_screen->y = mat->m[1][0] * corrected.x
               + mat->m[1][1] * corrected.y
               + mat->m[1][2];

/* 步骤4: 边界裁剪 */
if (out_screen->x < 0.0) out_screen->x = 0.0;
if (out_screen->y < 0.0) out_screen->y = 0.0;
if (out_screen->x > (double)space->screen_width_px) {
    out_screen->x = (double)space->screen_width_px;
}
if (out_screen->y > (double)space->screen_height_px) {
    out_screen->y = (double)space->screen_height_px;
}

return 0;
}

/*
 * 批量坐标变换 (优化版, 避免重复查找坐标空间)
 * 适用于一次性转换整条笔画的所有采样点
 *
 * @param raw_points 原始坐标数组, 每组2个unsigned int (x, y)
 * @param point_count 坐标点数量
 * @param page_id 页面ID
 * @param out_screen 输出屏幕坐标数组 (调用者负责分配内存)
 * @return 成功转换的点数
 */
int transform_batch(const unsigned int *raw_points, int point_count,
                   unsigned int page_id, Point2D *out_screen) {
    int idx = find_space_by_page(page_id);
    if (idx < 0 || out_screen == NULL) {
        return 0;
    }

    CoordinateSpace *space = &g_coord_manager.spaces[idx];
    if (!space->is_calibrated) {
        return 0;
    }
}

```

```

double dpi = g_coord_manager.dpi_resolution;
if (dpi < 1.0) dpi = 300.0;
double dots_to_mm = 25.4 / dpi;

AffineTransform *mat = &space->transform;
int converted = 0;
int i;

for (i = 0; i < point_count; i++) {
    /* 直接内联计算, 减少函数调用开销 */
    double px = (double)raw_points[i * 2] * dots_to_mm;
    double py = (double)raw_points[i * 2 + 1] * dots_to_mm;

    /* 畸变校正 (内联) */
    if (g_coord_manager.distortion_enabled) {
        DistortionParams *d = &g_coord_manager.distortion;
        double dx = px - d->cx;
        double dy = py - d->cy;
        double r2 = dx * dx + dy * dy;
        double radial = 1.0 + d->k1 * r2 + d->k2 * r2 * r2;
        px = d->cx + dx * radial + 2.0 * d->p1 * dx * dy
            + d->p2 * (r2 + 2.0 * dx * dx);
        py = d->cy + dy * radial + d->p1 * (r2 + 2.0 * dy * dy)
            + 2.0 * d->p2 * dx * dy;
    }

    /* 仿射变换 */
    double sx = mat->m[0][0] * px + mat->m[0][1] * py + mat->m[0][2];
    double sy = mat->m[1][0] * px + mat->m[1][1] * py + mat->m[1][2];

    /* 边界裁剪 */
    if (sx < 0.0) sx = 0.0;
    if (sy < 0.0) sy = 0.0;
    if (sx > (double)space->screen_width_px) sx = (double)space->screen_width_px;
    if (sy > (double)space->screen_height_px) sy = (double)space->screen_height_px;

    out_screen[i].x = sx;
    out_screen[i].y = sy;
    converted++;
}

return converted;
}

/*
 * 反向变换: 屏幕坐标 → 物理坐标
 * 用于在屏幕上点击后反推纸面物理位置
 * 需要计算仿射变换矩阵的逆矩阵
 */
int inverse_transform(double screen_x, double screen_y,
                     unsigned int page_id, Point2D *out_physical) {
    int idx = find_space_by_page(page_id);
    if (idx < 0 || out_physical == NULL) {
        return -1;
    }
}

CoordinateSpace *space = &g_coord_manager.spaces[idx];

```

```

AffineMatrix *mat = &space->transform;

/* 计算2x2子矩阵的行列式 */
double det = mat->m[0][0] * mat->m[1][1] - mat->m[0][1] * mat->m[1][0];
if (fabs(det) < 1e-12) {
    return -2;    /* 矩阵不可逆 */
}

double inv_det = 1.0 / det;

/* 减去平移分量 */
double tx = screen_x - mat->m[0][2];
double ty = screen_y - mat->m[1][2];

/* 应用逆矩阵 */
out_physical->x = inv_det * (mat->m[1][1] * tx - mat->m[0][1] * ty);
out_physical->y = inv_det * (mat->m[0][0] * ty - mat->m[1][0] * tx);

return 0;
}

```

core/stroke_smoother.c

```

/**
 * 自然写互动课堂应用开发SDK软件 V1.0
 * 笔迹平滑算法核心模块 - 笔迹坐标平滑与笔锋渲染
 *
 * 跨平台C语言核心库
 * 提供贝塞尔曲线平滑、笔锋宽度计算、坐标插值等算法
 * 确保各平台SDK输出一致的笔迹渲染效果
 */

#ifndef STROKE_SMOOTH_H
#define STROKE_SMOOTH_H

#include <stdint.h>
#include <stddef.h>
#include <math.h>

#ifdef __cplusplus
extern "C" {
#endif

/* ===== 常量定义 ===== */

#define MAX_SMOOTH_POINTS    4096    /* 平滑输出点缓冲区大小 */
#define MIN_POINT_DISTANCE  0.5f    /* 最小点间距（低于此值合并） */
#define BEZIER_SEGMENTS     8        /* 贝塞尔曲线分段数 */
#define PRESSURE_SMOOTH_FACTOR 0.3f /* 压力平滑因子 */

/* ===== 数据结构 ===== */

/** 二维浮点坐标点 */
typedef struct {

```

```

    float x;
    float y;
} Vec2f;

/** 带压力和时间戳的笔迹点 */
typedef struct {
    float x;
    float y;
    float pressure;    /* 0.0-1.0 */
    float width;       /* 计算后的笔画宽度 */
    uint32_t timestamp; /* 时间戳 */
} SmoothPoint;

/** 笔迹平滑器上下文 */
typedef struct {
    /* 输入点缓冲区（最近4个点，用于三次贝塞尔） */
    SmoothPoint input_buffer[4];
    int buffer_count;

    /* 输出点缓冲区 */
    SmoothPoint output_buffer[MAX_SMOOTH_POINTS];
    int output_count;

    /* 笔画宽度配置 */
    float min_width;    /* 最小笔画宽度 */
    float max_width;    /* 最大笔画宽度 */
    float velocity_scale; /* 速度对宽度的影响系数 */

    /* 上一点的平滑压力值 */
    float last_smooth_pressure;

    /* 统计信息 */
    uint32_t total_input_points;
    uint32_t total_output_points;
} StrokeSmoother;

/* ===== 数学工具函数 ===== */

/** 两点间欧氏距离 */
static inline float vec2f_distance(Vec2f a, Vec2f b) {
    float dx = b.x - a.x;
    float dy = b.y - a.y;
    return sqrtf(dx * dx + dy * dy);
}

/** 两点间线性插值 */
static inline Vec2f vec2f_lerp(Vec2f a, Vec2f b, float t) {
    Vec2f result;
    result.x = a.x + (b.x - a.x) * t;
    result.y = a.y + (b.y - a.y) * t;
    return result;
}

/** 浮点数线性插值 */
static inline float float_lerp(float a, float b, float t) {
    return a + (b - a) * t;
}

```

```

/** 将值裁剪到范围 [min_val, max_val] */
static inline float float_clamp(float value, float min_val, float max_val) {
    if (value < min_val) return min_val;
    if (value > max_val) return max_val;
    return value;
}

/* ===== 贝塞尔曲线算法 ===== */

/**
 * 计算三次贝塞尔曲线上的点
 *  $B(t) = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t) t^2 P_2 + t^3 P_3$ 
 *
 * 用于平滑连接相邻坐标点，消除折角使笔画圆润
 */
static Vec2f cubic_bezier(Vec2f p0, Vec2f p1, Vec2f p2, Vec2f p3, float t) {
    float u = 1.0f - t;
    float tt = t * t;
    float uu = u * u;
    float uuu = uu * u;
    float ttt = tt * t;

    Vec2f point;
    point.x = uuu * p0.x + 3.0f * uu * t * p1.x + 3.0f * u * tt * p2.x + ttt * p3.x;
    point.y = uuu * p0.y + 3.0f * uu * t * p1.y + 3.0f * u * tt * p2.y + ttt * p3.y;
    return point;
}

/**
 * 使用Catmull-Rom样条生成贝塞尔控制点
 * 从4个数据点(p0, p1, p2, p3)计算p1到p2之间的贝塞尔控制点
 * 确保曲线经过原始数据点 (C1连续)
 */
static void catmull_rom_to_bezier(
    Vec2f p0, Vec2f p1, Vec2f p2, Vec2f p3,
    Vec2f* cp1_out, Vec2f* cp2_out
) {
    float tension = 0.5f; /* 张力系数, 0.5为标准Catmull-Rom */
    cp1_out->x = p1.x + (p2.x - p0.x) * tension / 3.0f;
    cp1_out->y = p1.y + (p2.y - p0.y) * tension / 3.0f;
    cp2_out->x = p2.x - (p3.x - p1.x) * tension / 3.0f;
    cp2_out->y = p2.y - (p3.y - p1.y) * tension / 3.0f;
}

/* ===== 笔画宽度计算 ===== */

/**
 * 根据压力和速度计算笔画宽度
 * 模拟真实毛笔/钢笔的笔锋效果:
 * - 压力越大, 笔画越粗
 * - 速度越快, 笔画越细 (模拟快写时的飞白效果)
 * - 起笔/收笔处渐变细化
 */
static float calculate_stroke_width(
    float pressure, float velocity,
    float min_width, float max_width, float velocity_scale

```

```

) {
    /* 压力影响: 压力0→最细, 压力1→最粗 */
    float pressure_width = min_width + (max_width - min_width) * pressure;

    /* 速度衰减: 速度快时笔画变细 */
    float velocity_factor = 1.0f / (1.0f + velocity * velocity_scale);

    float width = pressure_width * velocity_factor;
    return float_clamp(width, min_width, max_width);
}

/* ===== 笔迹平滑器API ===== */

/**
 * 初始化笔迹平滑器
 */
static void smoother_init(StrokeSmoother* ctx, float min_width, float max_width) {
    ctx->buffer_count = 0;
    ctx->output_count = 0;
    ctx->min_width = min_width;
    ctx->max_width = max_width;
    ctx->velocity_scale = 0.005f;
    ctx->last_smooth_pressure = 0.5f;
    ctx->total_input_points = 0;
    ctx->total_output_points = 0;
}

/**
 * 输入一个新的坐标点
 * 当缓冲区积累到4个点时, 自动生成贝塞尔曲线平滑点
 * 返回新生成的平滑点数量
 */
static int smoother_add_point(StrokeSmoother* ctx, float x, float y,
                             float pressure, uint32_t timestamp) {
    ctx->total_input_points++;

    /* 压力平滑 (低通滤波器, 避免压力值跳变) */
    float smooth_pressure = ctx->last_smooth_pressure +
        PRESSURE_SMOOTH_FACTOR * (pressure - ctx->last_smooth_pressure);
    ctx->last_smooth_pressure = smooth_pressure;

    /* 添加到输入缓冲区 */
    int idx = ctx->buffer_count;
    if (idx >= 4) {
        /* 缓冲区满, 移位 */
        ctx->input_buffer[0] = ctx->input_buffer[1];
        ctx->input_buffer[1] = ctx->input_buffer[2];
        ctx->input_buffer[2] = ctx->input_buffer[3];
        idx = 3;
    }

    ctx->input_buffer[idx].x = x;
    ctx->input_buffer[idx].y = y;
    ctx->input_buffer[idx].pressure = smooth_pressure;
    ctx->input_buffer[idx].timestamp = timestamp;
    ctx->buffer_count = idx + 1;
}

```



```

/* 不足4个点时直接输出原始点 */
if (ctx->buffer_count < 4) {
    if (ctx->output_count < MAX_SMOOTH_POINTS) {
        /* 计算速度和宽度 */
        float velocity = 0;
        if (ctx->buffer_count >= 2) {
            Vec2f prev = {ctx->input_buffer[ctx->buffer_count-2].x, ctx->
input_buffer[ctx->buffer_count-2].y};
            Vec2f curr = {x, y};
            float dt = (float)(timestamp - ctx->input_buffer[ctx->buffer_count-
2].timestamp);
            if (dt > 0) velocity = vec2f_distance(prev, curr) / dt * 1000.0f;
        }

        float width = calculate_stroke_width(smooth_pressure, velocity,
            ctx->min_width, ctx->max_width, ctx->velocity_scale);

        SmoothPoint sp = {x, y, smooth_pressure, width, timestamp};
        ctx->output_buffer[ctx->output_count++] = sp;
        ctx->total_output_points++;
        return 1;
    }
    return 0;
}

/* 4个点准备好, 生成贝塞尔曲线 */
Vec2f p0 = {ctx->input_buffer[0].x, ctx->input_buffer[0].y};
Vec2f p1 = {ctx->input_buffer[1].x, ctx->input_buffer[1].y};
Vec2f p2 = {ctx->input_buffer[2].x, ctx->input_buffer[2].y};
Vec2f p3 = {ctx->input_buffer[3].x, ctx->input_buffer[3].y};

/* 计算贝塞尔控制点 */
Vec2f cp1, cp2;
catmull_rom_to_bezier(p0, p1, p2, p3, &cp1, &cp2);

/* 在p1到p2之间生成平滑点 */
int new_points = 0;
for (int i = 0; i <= BEZIER_SEGMENTS; i++) {
    if (ctx->output_count >= MAX_SMOOTH_POINTS) break;

    float t = (float)i / BEZIER_SEGMENTS;
    Vec2f pt = cubic_bezier(p1, cp1, cp2, p2, t);

    /* 插值压力和时间戳 */
    float interp_pressure = float_lerp(ctx->input_buffer[1].pressure,
        ctx->input_buffer[2].pressure, t);
    uint32_t interp_time = (uint32_t)float_lerp(
        (float)ctx->input_buffer[1].timestamp,
        (float)ctx->input_buffer[2].timestamp, t);

    /* 计算速度 */
    float velocity = 0;
    if (ctx->output_count > 0) {
        SmoothPoint* prev = &ctx->output_buffer[ctx->output_count - 1];
        Vec2f prev_v = {prev->x, prev->y};
        float dt = (float)(interp_time - prev->timestamp);
        if (dt > 0) velocity = vec2f_distance(prev_v, pt) / dt * 1000.0f;
    }
}

```

```

    }

    /* 计算笔画宽度 */
    float width = calculate_stroke_width(interp_pressure, velocity,
        ctx->min_width, ctx->max_width, ctx->velocity_scale);

    /* 距离过滤: 跳过距上一点太近的点 */
    if (ctx->output_count > 0) {
        SmoothPoint* prev = &ctx->output_buffer[ctx->output_count - 1];
        Vec2f prev_v = {prev->x, prev->y};
        if (vec2f_distance(prev_v, pt) < MIN_POINT_DISTANCE) continue;
    }

    SmoothPoint sp = {pt.x, pt.y, interp_pressure, width, interp_time};
    ctx->output_buffer[ctx->output_count++] = sp;
    ctx->total_output_points++;
    new_points++;
}

return new_points;
}

/**
 * 结束当前笔画 (抬笔时调用)
 * 输出最后一段贝塞尔曲线的收尾点
 */
static int smoother_end_stroke(StrokeSmoother* ctx) {
    int new_points = 0;

    /* 输出缓冲区中剩余的点 */
    if (ctx->buffer_count >= 2 && ctx->output_count < MAX_SMOOTH_POINTS) {
        int last = ctx->buffer_count - 1;
        float width = calculate_stroke_width(
            ctx->input_buffer[last].pressure * 0.5f, 0, /* 收笔处宽度减半 */
            ctx->min_width, ctx->max_width, ctx->velocity_scale);

        SmoothPoint sp = {
            ctx->input_buffer[last].x, ctx->input_buffer[last].y,
            ctx->input_buffer[last].pressure, width,
            ctx->input_buffer[last].timestamp
        };
        ctx->output_buffer[ctx->output_count++] = sp;
        new_points++;
    }

    /* 重置输入缓冲区 */
    ctx->buffer_count = 0;
    ctx->last_smooth_pressure = 0.5f;

    return new_points;
}

/**
 * 获取平滑后的输出点
 */
static inline const SmoothPoint* smoother_get_output(const StrokeSmoother* ctx) {
    return ctx->output_buffer;
}

```

```

}

/**
 * 获取输出点数量
 */
static inline int smoother_get_output_count(const StrokeSmoother* ctx) {
    return ctx->output_count;
}

/**
 * 清除输出缓冲区
 */
static inline void smoother_clear_output(StrokeSmoother* ctx) {
    ctx->output_count = 0;
}

/**
 * 获取统计信息
 */
static inline void smoother_get_stats(const StrokeSmoother* ctx,
    uint32_t* input_count, uint32_t* output_count) {
    if (input_count) *input_count = ctx->total_input_points;
    if (output_count) *output_count = ctx->total_output_points;
}

#ifdef __cplusplus
}
#endif

#endif /* STROKE_SMOOTHER_H */

```

model/

model/PenDevice.java

```

/*
 * 自然写互动课堂应用开发SDK软件 V1.0
 * PenDevice - 点阵笔设备数据模型
 *
 * 描述: 封装点阵笔的设备信息、连接状态、能力参数等
 */

package com.writech.sdk.model;

import java.io.Serializable;

/**
 * 点阵笔设备模型
 * 包含设备基本信息、硬件参数和连接状态
 */
public class PenDevice implements Serializable {

    private static final long serialVersionUID = 1L;

```

```
/* ===== 基本信息 ===== */

/** 设备MAC地址 (唯一标识) */
private String macAddress;

/** 设备名称 (用户可自定义) */
private String deviceName;

/** 设备型号 (如 WP-200, WP-300) */
private String modelName;

/** 固件版本号 (如 2.1.5) */
private String firmwareVersion;

/** 硬件版本号 */
private String hardwareVersion;

/** 设备序列号 */
private String serialNumber;

/* ===== 硬件能力 ===== */

/** 采样率 (Hz, 常见值: 100, 200) */
private int sampleRate;

/** 压力感应级别 (常见值: 1024, 2048, 4096) */
private int pressureLevels;

/** 坐标分辨率 (DPI, 常见值: 300, 600) */
private int coordinateDpi;

/** 是否支持倾斜角检测 */
private boolean tiltSupported;

/** BLE协议版本 (4.2 / 5.0 / 5.3) */
private String bleVersion;

/** 电池容量 (mAh) */
private int batteryCapacity;

/* ===== 运行状态 ===== */

/** 连接状态枚举 */
public enum ConnectionState {
    DISCONNECTED,      /* 未连接 */
    CONNECTING,        /* 正在连接 */
    CONNECTED,         /* 已连接 */
    RECONNECTING       /* 正在重连 */
}

/** 当前连接状态 */
private ConnectionState connectionState = ConnectionState.DISCONNECTED;

/** 当前电量百分比 (0-100) */
private int batteryLevel;
```

```

/** 是否正在充电 */
private boolean isCharging;

/** 是否正在书写（笔尖接触纸面） */
private boolean isWriting;

/** 信号强度RSSI (dBm) */
private int rssi;

/** 最后一次通信时间（毫秒时间戳） */
private long lastCommunicationTime;

/** 累计书写时长（秒） */
private long totalWritingDuration;

/** 绑定的学生ID */
private String boundStudentId;

/** 绑定的学生姓名 */
private String boundStudentName;

/* ===== 构造函数 ===== */

public PenDevice() {
}

public PenDevice(String macAddress, String deviceName) {
    this.macAddress = macAddress;
    this.deviceName = deviceName;
    this.sampleRate = 100;
    this.pressureLevels = 4096;
    this.coordinateDpi = 300;
}

/* ===== Getter / Setter ===== */

public String getMacAddress() { return macAddress; }
public void setMacAddress(String macAddress) { this.macAddress = macAddress; }

public String getDeviceName() { return deviceName; }
public void setDeviceName(String deviceName) { this.deviceName = deviceName; }

public String getModelName() { return modelName; }
public void setModelName(String modelName) { this.modelName = modelName; }

public String getFirmwareVersion() { return firmwareVersion; }
public void setFirmwareVersion(String firmwareVersion) { this.firmwareVersion =
firmwareVersion; }

public String getHardwareVersion() { return hardwareVersion; }
public void setHardwareVersion(String v) { this.hardwareVersion = v; }

public String getSerialNumber() { return serialNumber; }
public void setSerialNumber(String serialNumber) { this.serialNumber = serialNumber;
}

public int getSampleRate() { return sampleRate; }

```

```

    public void setSampleRate(int sampleRate) { this.sampleRate = sampleRate; }

    public int getPressureLevels() { return pressureLevels; }
    public void setPressureLevels(int pressureLevels) { this.pressureLevels =
pressureLevels; }

    public int getCoordinateDpi() { return coordinateDpi; }
    public void setCoordinateDpi(int coordinateDpi) { this.coordinateDpi =
coordinateDpi; }

    public boolean isTiltSupported() { return tiltSupported; }
    public void setTiltSupported(boolean tiltSupported) { this.tiltSupported =
tiltSupported; }

    public String getBleVersion() { return bleVersion; }
    public void setBleVersion(String bleVersion) { this.bleVersion = bleVersion; }

    public int getBatteryCapacity() { return batteryCapacity; }
    public void setBatteryCapacity(int batteryCapacity) { this.batteryCapacity =
batteryCapacity; }

    public ConnectionState getConnectionState() { return connectionState; }
    public void setConnectionState(ConnectionState state) { this.connectionState =
state; }

    public int getBatteryLevel() { return batteryLevel; }
    public void setBatteryLevel(int batteryLevel) { this.batteryLevel = batteryLevel; }

    public boolean isCharging() { return isCharging; }
    public void setCharging(boolean charging) { isCharging = charging; }

    public boolean isWriting() { return isWriting; }
    public void setWriting(boolean writing) { isWriting = writing; }

    public int getRssi() { return rssi; }
    public void setRssi(int rssi) { this.rssi = rssi; }

    public long getLastCommunicationTime() { return lastCommunicationTime; }
    public void setLastCommunicationTime(long t) { this.lastCommunicationTime = t; }

    public long getTotalWritingDuration() { return totalWritingDuration; }
    public void setTotalWritingDuration(long d) { this.totalWritingDuration = d; }

    public String getBoundStudentId() { return boundStudentId; }
    public void setBoundStudentId(String id) { this.boundStudentId = id; }

    public String getBoundStudentName() { return boundStudentName; }
    public void setBoundStudentName(String name) { this.boundStudentName = name; }

    /* ===== 便捷方法 ===== */

    /** 是否已连接 */
    public boolean isConnected() {
        return connectionState == ConnectionState.CONNECTED;
    }

    /** 电量是否低 (<= 10%) */

```

```

public boolean isLowBattery() {
    return batteryLevel <= 10 && !isCharging;
}

/** 获取设备显示名称（优先显示学生姓名） */
public String getDisplayName() {
    if (boundStudentName != null && !boundStudentName.isEmpty()) {
        return boundStudentName + "的笔";
    }
    return deviceName != null ? deviceName : "WritechPen-" + macAddress;
}

@Override
public String toString() {
    return "PenDevice{" +
        "mac='" + macAddress + '\'' +
        ", name='" + deviceName + '\'' +
        ", model='" + modelName + '\'' +
        ", state=" + connectionState +
        ", battery=" + batteryLevel + "%" +
        ", writing=" + isWriting +
        '}';
}

@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    PenDevice that = (PenDevice) o;
    return macAddress != null && macAddress.equals(that.macAddress);
}

@Override
public int hashCode() {
    return macAddress != null ? macAddress.hashCode() : 0;
}
}

```

model/RecognitionResult.java

```

/*
 * 自然写互动课堂应用开发SDK软件 V1.0
 * RecognitionResult - 识别结果数据模型
 *
 * 描述: 封装OCR识别、数学公式识别、笔顺评分等各类识别结果
 */

package com.writech.sdk.model;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

/**

```

```

* 识别结果统一模型
* 支持多种识别类型的结果封装
*/
public class RecognitionResult implements Serializable {

    private static final long serialVersionUID = 1L;

    /* ===== 识别类型常量 ===== */

    /** 手写文字识别 */
    public static final int TYPE_HANDWRITING = 0;

    /** 数学公式识别 */
    public static final int TYPE_MATH = 1;

    /** 笔顺评分 */
    public static final int TYPE_STROKE_ORDER = 2;

    /** 作文评分 */
    public static final int TYPE_ESSAY = 3;

    /* ===== 候选结果内部类 ===== */

    /** 单个候选识别结果 */
    public static class Candidate implements Serializable {
        private static final long serialVersionUID = 1L;

        /** 识别文本 */
        public String text;

        /** 置信度 (0.0~1.0) */
        public float confidence;

        /** 候选排名 */
        public int rank;

        public Candidate() {}

        public Candidate(String text, float confidence, int rank) {
            this.text = text;
            this.confidence = confidence;
            this.rank = rank;
        }

        @Override
        public String toString() {
            return "Candidate{'" + text + "', conf=" + confidence + "}";
        }
    }

    /** 笔顺评分详情 */
    public static class StrokeOrderDetail implements Serializable {
        private static final long serialVersionUID = 1L;

        /** 评分笔画序号 */
        public int strokeIndex;
    }
}

```



```

/** 该笔是否正确 */
public boolean isCorrect;

/** 标准笔顺名称 (如"横"、"竖"、"撇") */
public String standardStrokeName;

/** 实际书写的笔画类型 */
public String actualStrokeName;

/** 笔画形态相似度 (0.0~1.0) */
public float shapeSimilarity;

public StrokeOrderDetail() {}

@Override
public String toString() {
    return "Stroke#" + strokeIndex + ": " + (isCorrect ? "正确" : "错误")
        + " (标准:" + standardStrokeName + ", 实际:" + actualStrokeName + ")";
}
}

/** 作文评分详情 */
public static class EssayScoreDetail implements Serializable {
    private static final long serialVersionUID = 1L;

    /** 内容分 (百分制) */
    public float contentScore;

    /** 结构分 */
    public float structureScore;

    /** 语言分 */
    public float languageScore;

    /** 书写规范分 */
    public float handwritingScore;

    /** 总分 */
    public float totalScore;

    /** 评语 */
    public String comment;

    /** 优点列表 */
    public List<String> highlights = new ArrayList<>();

    /** 改进建议列表 */
    public List<String> suggestions = new ArrayList<>();

    public EssayScoreDetail() {}
}

/* ===== 结果字段 ===== */

/** 识别请求ID (对应任务ID) */
private int requestId;

```

```
/** 识别类型 */
private int recognitionType;

/** 识别是否成功 */
private boolean success;

/** 错误码（成功时为0） */
private int errorCode;

/** 错误消息 */
private String errorMessage;

/** 主要识别结果文本 */
private String resultText;

/** 主要结果置信度 */
private float confidence;

/** 候选结果列表（按置信度降序） */
private List<Candidate> candidates;

/** 笔顺评分详情（仅笔顺类型） */
private List<StrokeOrderDetail> strokeOrderDetails;

/** 笔顺总分（0-100） */
private float strokeOrderScore;

/** 笔顺正确笔画数 */
private int correctStrokeCount;

/** 笔顺总笔画数 */
private int totalStrokeCount;

/** 作文评分详情（仅作文类型） */
private EssayScoreDetail essayDetail;

/** 数学公式LaTeX表示（仅数学类型） */
private String mathLatex;

/** 数学计算结果（如果是计算题） */
private String mathAnswer;

/** 识别耗时（毫秒） */
private long processingTimeMs;

/** 结果来源（"online"在线 / "offline"离线 / "cache"缓存） */
private String source;

/** 识别时间戳 */
private long timestamp;

/* ===== 构造函数 ===== */

public RecognitionResult() {
    this.candidates = new ArrayList<>();
    this.strokeOrderDetails = new ArrayList<>();
    this.timestamp = System.currentTimeMillis();
}
```

```

    }

    /** 创建成功结果 */
    public static RecognitionResult success(int requestId, int type, String text, float
confidence) {
        RecognitionResult result = new RecognitionResult();
        result.requestId = requestId;
        result.recognitionType = type;
        result.success = true;
        result.errorCode = 0;
        result.resultText = text;
        result.confidence = confidence;
        return result;
    }

    /** 创建失败结果 */
    public static RecognitionResult failure(int requestId, int errorCode, String
message) {
        RecognitionResult result = new RecognitionResult();
        result.requestId = requestId;
        result.success = false;
        result.errorCode = errorCode;
        result.errorMessage = message;
        return result;
    }

    /* ===== Getter / Setter ===== */

    public int getRequestId() { return requestId; }
    public void setRequestId(int id) { this.requestId = id; }

    public int getRecognitionType() { return recognitionType; }
    public void setRecognitionType(int type) { this.recognitionType = type; }

    public boolean isSuccess() { return success; }
    public void setSuccess(boolean success) { this.success = success; }

    public int getErrorCode() { return errorCode; }
    public void setErrorCode(int code) { this.errorCode = code; }

    public String getErrorMessage() { return errorMessage; }
    public void setErrorMessage(String msg) { this.errorMessage = msg; }

    public String getResultText() { return resultText; }
    public void setResultText(String text) { this.resultText = text; }

    public float getConfidence() { return confidence; }
    public void setConfidence(float c) { this.confidence = c; }

    public List<Candidate> getCandidates() { return candidates; }
    public void setCandidates(List<Candidate> c) { this.candidates = c; }

    public void addCandidate(String text, float confidence, int rank) {
        candidates.add(new Candidate(text, confidence, rank));
    }

    public List<StrokeOrderDetail> getStrokeOrderDetails() { return strokeOrderDetails;

```

```

}

    public void setStrokeOrderDetails(List<StrokeOrderDetail> d) {
this.strokeOrderDetails = d; }

    public float getStrokeOrderScore() { return strokeOrderScore; }
    public void setStrokeOrderScore(float s) { this.strokeOrderScore = s; }

    public int getCorrectStrokeCount() { return correctStrokeCount; }
    public void setCorrectStrokeCount(int c) { this.correctStrokeCount = c; }

    public int getTotalStrokeCount() { return totalStrokeCount; }
    public void setTotalStrokeCount(int t) { this.totalStrokeCount = t; }

    public EssayScoreDetail getEssayDetail() { return essayDetail; }
    public void setEssayDetail(EssayScoreDetail d) { this.essayDetail = d; }

    public String getMathLatex() { return mathLatex; }
    public void setMathLatex(String latex) { this.mathLatex = latex; }

    public String getMathAnswer() { return mathAnswer; }
    public void setMathAnswer(String answer) { this.mathAnswer = answer; }

    public long getProcessingTimeMs() { return processingTimeMs; }
    public void setProcessingTimeMs(long ms) { this.processingTimeMs = ms; }

    public String getSource() { return source; }
    public void setSource(String source) { this.source = source; }

    public long getTimestamp() { return timestamp; }
    public void setTimestamp(long t) { this.timestamp = t; }

    /* ===== 便捷方法 ===== */

    /** 获取最佳候选结果 */
    public Candidate getBestCandidate() {
        return candidates.isEmpty() ? null : candidates.get(0);
    }

    /** 获取笔顺正确率 */
    public float getStrokeOrderAccuracy() {
        return totalStrokeCount > 0 ? (float) correctStrokeCount / totalStrokeCount : 0;
    }

    /** 获取识别类型的中文描述 */
    public String getTypeDescription() {
        switch (recognitionType) {
            case TYPE_HANDWRITING: return "手写识别";
            case TYPE_MATH: return "数学识别";
            case TYPE_STROKE_ORDER: return "笔顺评分";
            case TYPE_ESSAY: return "作文评分";
            default: return "未知类型";
        }
    }

    @Override
    public String toString() {
        if (success) {

```

```

        return "RecognitionResult{type=" + getTypeDescription()
            + ", text='" + resultText + "'"
            + ", confidence=" + confidence
            + ", source=" + source
            + ", time=" + processingTimeMs + "ms}";
    } else {
        return "RecognitionResult{FAILED, code=" + errorCode
            + ", msg='" + errorMessage + "'}";
    }
}
}
}

```

model/StrokePath.java

```

/*
 * 自然写互动课堂应用开发SDK软件 V1.0
 * StrokePath - 笔迹路径数据模型
 *
 * 描述: 封装一条完整笔画的坐标序列、属性和元数据
 */

package com.writech.sdk.model;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

/**
 * 笔迹路径模型
 * 代表从落笔到抬笔的一条完整笔画数据
 */
public class StrokePath implements Serializable {

    private static final long serialVersionUID = 1L;

    /* ===== 采样点内部类 ===== */

    /** 单个笔迹采样点 */
    public static class Point implements Serializable {
        private static final long serialVersionUID = 1L;

        /** X坐标（屏幕像素或物理mm，取决于坐标空间） */
        public float x;

        /** Y坐标 */
        public float y;

        /** 压力值（归一化 0.0~1.0） */
        public float pressure;

        /** 时间戳（相对于笔画开始时间的毫秒偏移） */
        public long timeOffset;

        /** 笔尖倾斜角度（度，0-90，0为垂直，部分笔支持） */
    }
}

```

```

    public float tiltAngle;

    /** 笔尖方位角（度，0-360，部分笔支持） */
    public float azimuthAngle;

    public Point() {}

    public Point(float x, float y, float pressure, long timeOffset) {
        this.x = x;
        this.y = y;
        this.pressure = pressure;
        this.timeOffset = timeOffset;
    }

    @Override
    public String toString() {
        return "(" + x + "," + y + ",p=" + pressure + ",t=" + timeOffset + ")";
    }
}

/* ===== 笔画属性 ===== */

/** 笔画唯一ID */
private String strokeId;

/** 来源笔设备MAC地址 */
private String penMac;

/** 学生ID */
private String studentId;

/** 页面ID（标识书写所在页面） */
private String pageId;

/** 笔画开始时间（绝对时间戳毫秒） */
private long startTimestamp;

/** 笔画结束时间 */
private long endTimestamp;

/** 笔画颜色（ARGB） */
private int color = 0xFF000000;

/** 笔画基础线宽（像素） */
private float baseWidth = 3.0f;

/** 采样点列表 */
private List<Point> points;

/* ===== 分析结果（由OCR/AI引擎填充） ===== */

/** 识别的文字内容 */
private String recognizedText;

/** 识别置信度 */
private float recognitionConfidence;

```

```

/** 笔序号（在整个书写序列中的顺序） */
private int strokeOrder;

/** 是否为有效笔画（排除误触等） */
private boolean isValid = true;

/* ===== 构造函数 ===== */

public StrokePath() {
    this.points = new ArrayList<>();
}

public StrokePath(String strokeId, String penMac) {
    this.strokeId = strokeId;
    this.penMac = penMac;
    this.points = new ArrayList<>();
    this.startTimestamp = System.currentTimeMillis();
}

/* ===== 点操作方法 ===== */

/** 添加采样点 */
public void addPoint(float x, float y, float pressure, long timeOffset) {
    points.add(new Point(x, y, pressure, timeOffset));
}

/** 添加采样点（含倾斜角） */
public void addPointWithTilt(float x, float y, float pressure,
                             long timeOffset, float tilt, float azimuth) {
    Point p = new Point(x, y, pressure, timeOffset);
    p.tiltAngle = tilt;
    p.azimuthAngle = azimuth;
    points.add(p);
}

/** 获取采样点数量 */
public int getPointCount() {
    return points.size();
}

/** 获取指定索引的采样点 */
public Point getPoint(int index) {
    if (index >= 0 && index < points.size()) {
        return points.get(index);
    }
    return null;
}

/** 获取所有采样点 */
public List<Point> getPoints() {
    return points;
}

/* ===== 笔画几何计算 ===== */

/** 计算笔画总长度（像素） */
public float calculateLength() {

```

```

        float length = 0;
        for (int i = 1; i < points.size(); i++) {
            Point p0 = points.get(i - 1);
            Point p1 = points.get(i);
            float dx = p1.x - p0.x;
            float dy = p1.y - p0.y;
            length += (float) Math.sqrt(dx * dx + dy * dy);
        }
        return length;
    }

    /** 计算笔画包围盒 */
    public float[] getBoundingBox() {
        if (points.isEmpty()) return new float[]{0, 0, 0, 0};

        float minX = Float.MAX_VALUE, minY = Float.MAX_VALUE;
        float maxX = Float.MIN_VALUE, maxY = Float.MIN_VALUE;

        for (Point p : points) {
            if (p.x < minX) minX = p.x;
            if (p.y < minY) minY = p.y;
            if (p.x > maxX) maxX = p.x;
            if (p.y > maxY) maxY = p.y;
        }

        return new float[]{minX, minY, maxX, maxY};
    }

    /** 计算平均书写速度 (像素/毫秒) */
    public float calculateAverageSpeed() {
        if (points.size() < 2) return 0;

        float totalLength = calculateLength();
        long duration = points.get(points.size() - 1).timeOffset -
            points.get(0).timeOffset;

        return duration > 0 ? totalLength / duration : 0;
    }

    /** 计算平均压力 */
    public float calculateAveragePressure() {
        if (points.isEmpty()) return 0;
        float sum = 0;
        for (Point p : points) {
            sum += p.pressure;
        }
        return sum / points.size();
    }

    /** 获取书写持续时间 (毫秒) */
    public long getDuration() {
        if (points.size() < 2) return 0;
        return points.get(points.size() - 1).timeOffset - points.get(0).timeOffset;
    }

    /** ===== 序列化方法 ===== */

```



```

/**
 * 将笔画数据序列化为紧凑的二进制格式
 * 用于BLE传输和本地缓存
 *
 * 格式:
 * [4字节 点数][每个点: 4字节x + 4字节y + 2字节pressure + 4字节timeOffset]
 */
public byte[] toBytes() {
    int pointCount = points.size();
    byte[] data = new byte[4 + pointCount * 14];

    /* 写入点数 (大端序) */
    data[0] = (byte) ((pointCount >> 24) & 0xFF);
    data[1] = (byte) ((pointCount >> 16) & 0xFF);
    data[2] = (byte) ((pointCount >> 8) & 0xFF);
    data[3] = (byte) (pointCount & 0xFF);

    int offset = 4;
    for (Point p : points) {
        /* 写入X坐标 (float → 4字节) */
        int fx = Float.floatToIntBits(p.x);
        data[offset++] = (byte) ((fx >> 24) & 0xFF);
        data[offset++] = (byte) ((fx >> 16) & 0xFF);
        data[offset++] = (byte) ((fx >> 8) & 0xFF);
        data[offset++] = (byte) (fx & 0xFF);

        /* 写入Y坐标 */
        int fy = Float.floatToIntBits(p.y);
        data[offset++] = (byte) ((fy >> 24) & 0xFF);
        data[offset++] = (byte) ((fy >> 16) & 0xFF);
        data[offset++] = (byte) ((fy >> 8) & 0xFF);
        data[offset++] = (byte) (fy & 0xFF);

        /* 写入压力值 (归一化后*65535转uint16) */
        int pressure16 = (int) (p.pressure * 65535);
        data[offset++] = (byte) ((pressure16 >> 8) & 0xFF);
        data[offset++] = (byte) (pressure16 & 0xFF);

        /* 写入时间偏移 (uint32) */
        long t = p.timeOffset;
        data[offset++] = (byte) ((t >> 24) & 0xFF);
        data[offset++] = (byte) ((t >> 16) & 0xFF);
        data[offset++] = (byte) ((t >> 8) & 0xFF);
        data[offset++] = (byte) (t & 0xFF);
    }

    return data;
}

/* ===== Getter / Setter ===== */

public String getStrokeId() { return strokeId; }
public void setStrokeId(String strokeId) { this.strokeId = strokeId; }

public String getPenMac() { return penMac; }
public void setPenMac(String penMac) { this.penMac = penMac; }

```

```

public String getStudentId() { return studentId; }
public void setStudentId(String studentId) { this.studentId = studentId; }

public String getPageId() { return pageId; }
public void setPageId(String pageId) { this.pageId = pageId; }

public long getStartTimestamp() { return startTimestamp; }
public void setStartTimestamp(long t) { this.startTimestamp = t; }

public long getEndTimestamp() { return endTimestamp; }
public void setEndTimestamp(long t) { this.endTimestamp = t; }

public int getColor() { return color; }
public void setColor(int color) { this.color = color; }

public float getBaseWidth() { return baseWidth; }
public void setBaseWidth(float w) { this.baseWidth = w; }

public String getRecognizedText() { return recognizedText; }
public void setRecognizedText(String text) { this.recognizedText = text; }

public float getRecognitionConfidence() { return recognitionConfidence; }
public void setRecognitionConfidence(float c) { this.recognitionConfidence = c; }

public int getStrokeOrder() { return strokeOrder; }
public void setStrokeOrder(int order) { this.strokeOrder = order; }

public boolean isValid() { return isValid; }
public void setValid(boolean valid) { isValid = valid; }

@Override
public String toString() {
    return "StrokePath{id='" + strokeId + "', points=" + points.size()
        + ", duration=" + getDuration() + "ms"
        + ", text='" + recognizedText + "'}";
}
}

```