

# 自然写互动课堂手机端应用软件 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

```
06-writech-app-mobile/  
├─ main.dart  
├─ repository/  
│   └─ local_repository.dart  
├─ service/  
│   ├── api_service.dart  
│   ├── ble_service.dart  
│   └─ websocket_service.dart  
├─ ui/  
│   └─ common/  
│       └─ stroke_canvas.dart  
└─ util/  
    └─ encryption_util.dart
```

## 源程序文件清单

(根目录)

**main.dart**

```
/// 自然写互动课堂手机端应用软件 V1.0  
/// APP入口 - Flutter应用主入口与全局初始化  
///  
/// 功能说明：  
/// 1. Flutter应用初始化（引擎绑定、错误处理）  
/// 2. 全局依赖注入（GetIt服务定位器）  
/// 3. 推送通知初始化（APNs / FCM）  
/// 4. 用户认证状态恢复
```

```

/// 5. 多主题支持（浅色/深色/护眼模式）
/// 6. 国际化配置（中文/English）

import 'dart:async';
import 'dart:io';
import 'package:flutter/material.dart';
import 'package:flutter/services.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import 'package:get_it/get_it.dart';
import 'package:shared_preferences/shared_preferences.dart';

/// 全局服务定位器实例
final GetIt getIt = GetIt.instance;

/// 应用程序入口
void main() async {
  // 确保Flutter引擎初始化完成
  WidgetsFlutterBinding.ensureInitialized();

  // 设置全局错误处理（捕获未处理的Flutter框架错误）
  FlutterError.onError = (FlutterErrorDetails details) {
    FlutterError.presentError(details);
    _reportError(details.exception, details.stack);
  };

  // 初始化全局依赖
  await _initDependencies();

  // 设置系统UI样式（状态栏透明）
  SystemChrome.setSystemUIOverlayStyle(const SystemUiOverlayStyle(
    statusBarColor: Colors.transparent,
    statusBarIconBrightness: Brightness.dark,
  ));

  // 设置屏幕方向（手机端仅支持竖屏）
  await SystemChrome.setPreferredOrientations([
    DeviceOrientation.portraitUp,
    DeviceOrientation.portraitDown,
  ]);

  // 运行应用（包裹Zone错误处理）
  runZonedGuarded(() {
    runApp(const WritechMobileApp());
  }, (error, stackTrace) {
    _reportError(error, stackTrace);
  });
}

/// 初始化全局依赖注入
/// 注册所有服务层单例（API、WebSocket、BLE、本地存储）
Future<void> _initDependencies() async {
  // 共享偏好设置（用户配置持久化）
  final prefs = await SharedPreferences.getInstance();
  getIt.registerSingleton<SharedPreferences>(prefs);

  // 注册API服务（云平台REST API通信）
  getIt.registerLazySingleton<ApiService>(() => ApiService());
}

```

```

// 注册WebSocket服务（实时通知推送）
getIt.registerLazySingleton<WebSocketService>(() => WebSocketService());

// 注册BLE蓝牙服务（教师端连接点阵笔）
getIt.registerLazySingleton<BleService>(() => BleService());

// 注册本地数据仓库（SQLite缓存）
getIt.registerLazySingleton<LocalRepository>(() => LocalRepository());

// 初始化推送通知
await _initPushNotification();
}

/// 初始化推送通知服务
/// iOS使用APNs, Android使用FCM
Future<void> _initPushNotification() async {
  // 请求通知权限（iOS需要显式请求）
  if (Platform.isIOS) {
    // 请求APNs推送权限
    debugPrint('[Push] 请求iOS推送权限');
  }
  // 获取设备推送Token并注册到云平台
  debugPrint('[Push] 推送通知初始化完成');
}

/// 全局错误上报（发送到云端错误收集服务）
void _reportError(dynamic error, StackTrace? stackTrace) {
  debugPrint('[CrashReport] 捕获异常: $error');
  debugPrint('[CrashReport] 堆栈: $stackTrace');
  // 生产环境上报到Sentry/Firebase Crashlytics
}

/// 应用根Widget - 配置路由、主题、状态管理
class WritechMobileApp extends StatefulWidget {
  const WritechMobileApp({super.key});

  @override
  State<WritechMobileApp> createState() => _WritechMobileAppState();
}

class _WritechMobileAppState extends State<WritechMobileApp>
  with WidgetsBindingObserver {
  /// 当前主题模式
  ThemeMode _themeMode = ThemeMode.light;

  /// 用户角色（教师/家长）决定显示的功能入口
  String _userRole = 'teacher';

  @override
  void initState() {
    super.initState();
    WidgetsBinding.instance.addObserver(this);
    _loadUserPreferences();
  }

  @override

```

```

void dispose() {
  WidgetsBinding.instance.removeObserver(this);
  super.dispose();
}

/// 监听应用生命周期变化
@override
void didChangeAppLifecycleState(AppLifecycleState state) {
  switch (state) {
    case AppLifecycleState.resumed:
      // 前台恢复: 重建WebSocket连接、刷新Token
      debugPrint('[App] 应用回到前台');
      getIt<WebSocketService>().reconnect();
      break;
    case AppLifecycleState.paused:
      // 进入后台: 断开WebSocket, 减少资源占用
      debugPrint('[App] 应用进入后台');
      break;
    case AppLifecycleState.detached:
      // 应用销毁: 清理所有资源
      _cleanup();
      break;
    default:
      break;
  }
}

/// 加载用户偏好设置 (主题、角色、语言等)
void _loadUserPreferences() {
  final prefs = getIt<SharedPreferences>();
  final themeName = prefs.getString('theme_mode') ?? 'light';
  setState(() {
    _themeMode = themeName == 'dark' ? ThemeMode.dark : ThemeMode.light;
    _userRole = prefs.getString('user_role') ?? 'teacher';
  });
}

/// 清理全局资源
void _cleanup() {
  getIt<WebSocketService>().disconnect();
  getIt<BleService>().disconnectAll();
  debugPrint('[App] 全局资源清理完成');
}

@override
Widget build(BuildContext context) {
  return MultiBlocProvider(
    providers: [
      // 认证状态管理 (登录/登出/Token刷新)
      BlocProvider<AuthBloc>(create: (_) => AuthBloc()),
      // 作业状态管理 (列表/详情/提交)
      BlocProvider<AssignmentBloc>(create: (_) => AssignmentBloc()),
      // 消息状态管理 (通知/家校沟通)
      BlocProvider<MessageBloc>(create: (_) => MessageBloc()),
    ],
    child: MaterialApp(
      title: '自然写互动课堂',

```

```

        debugShowCheckedModeBanner: false,
        themeMode: _themeMode,
        // 浅色主题
        theme: _buildLightTheme(),
        // 深色主题
        darkTheme: _buildDarkTheme(),
        // 路由配置
        initialRoute: '/splash',
        routes: _buildRoutes(),
    ),
);
}

/// 构建浅色主题
ThemeData _buildLightTheme() {
    return ThemeData(
        useMaterial3: true,
        colorScheme: ColorScheme.fromSeed(
            seedColor: const Color(0xFF2196F3), // 品牌蓝色
            brightness: Brightness.light,
        ),
        fontFamily: 'NotoSansSC',
        appBarTheme: const AppBarTheme(
            centerTitle: true,
            elevation: 0,
        ),
        cardTheme: CardTheme(
            elevation: 2,
            shape: RoundedRectangleBorder(borderRadius: BorderRadius.circular(12)),
        ),
    );
}

/// 构建深色主题
ThemeData _buildDarkTheme() {
    return ThemeData(
        useMaterial3: true,
        colorScheme: ColorScheme.fromSeed(
            seedColor: const Color(0xFF2196F3),
            brightness: Brightness.dark,
        ),
        fontFamily: 'NotoSansSC',
    );
}

/// 构建应用路由表
Map<String, WidgetBuilder> _buildRoutes() {
    return {
        '/splash': (_) => const SplashScreen(),
        '/login': (_) => const LoginPage(),
        '/teacher_home': (_) => const TeacherHomePage(),
        '/parent_home': (_) => const ParentHomePage(),
        '/assignment_detail': (_) => const AssignmentDetailPage(),
        '/stroke_replay': (_) => const StrokeReplayPage(),
        '/report_detail': (_) => const ReportDetailPage(),
        '/ble_connect': (_) => const BleConnectPage(),
        '/settings': (_) => const SettingsPage(),
    };
}

```

```

    };
  }
}

/* ===== 占位Widget声明（各页面在独立文件中实现） ===== */

/// 启动页 - 展示Logo + 自动登录检查
class SplashScreen extends StatelessWidget {
  const SplashScreen({super.key});
  @override
  Widget build(BuildContext context) => const Scaffold(body: Center(child: Text('自然
写')));
}

/// 登录页占位
class LoginPage extends StatelessWidget {
  const LoginPage({super.key});
  @override
  Widget build(BuildContext context) => const Scaffold();
}

/// 教师首页占位
class TeacherHomePage extends StatelessWidget {
  const TeacherHomePage({super.key});
  @override
  Widget build(BuildContext context) => const Scaffold();
}

/// 家长首页占位
class ParentHomePage extends StatelessWidget {
  const ParentHomePage({super.key});
  @override
  Widget build(BuildContext context) => const Scaffold();
}

/// 作业详情占位
class AssignmentDetailPage extends StatelessWidget {
  const AssignmentDetailPage({super.key});
  @override
  Widget build(BuildContext context) => const Scaffold();
}

/// 笔迹回放占位
class StrokeReplayPage extends StatelessWidget {
  const StrokeReplayPage({super.key});
  @override
  Widget build(BuildContext context) => const Scaffold();
}

/// 学情报告详情占位
class ReportDetailPage extends StatelessWidget {
  const ReportDetailPage({super.key});
  @override
  Widget build(BuildContext context) => const Scaffold();
}

/// BLE蓝牙连接占位

```

```

class BleConnectPage extends StatelessWidget {
  const BleConnectPage({super.key});
  @override
  Widget build(BuildContext context) => const Scaffold();
}

/// 设置页占位
class SettingsPage extends StatelessWidget {
  const SettingsPage({super.key});
  @override
  Widget build(BuildContext context) => const Scaffold();
}

/* ===== Bloc占位声明 ===== */

/// 认证Bloc - 管理登录/登出/Token刷新状态
class AuthBloc extends Cubit<int> {
  AuthBloc() : super(0);
}

/// 作业Bloc - 管理作业列表/详情/提交状态
class AssignmentBloc extends Cubit<int> {
  AssignmentBloc() : super(0);
}

/// 消息Bloc - 管理通知和家校沟通消息
class MessageBloc extends Cubit<int> {
  MessageBloc() : super(0);
}

/* ===== 服务占位声明 ===== */

/// API服务占位
class ApiService {}

/// WebSocket服务占位
class WebSocketService {
  void reconnect() {}
  void disconnect() {}
}

/// BLE服务占位
class BleService {
  void disconnectAll() {}
}

/// 本地仓库占位
class LocalRepository {}

```

**repository/**

**repository/local\_repository.dart**

```

/// 自然写互动课堂手机端应用软件 V1.0
/// 本地数据仓库 - SQLite本地缓存与离线数据管理
///
/// 功能说明:
/// 1. SQLite数据库初始化与版本迁移
/// 2. 作业列表本地缓存 (支持离线查看)
/// 3. 学情报告缓存 (减少网络请求)
/// 4. 消息记录本地存储
/// 5. 笔迹数据暂存 (教师端BLE收笔后等待上传)
/// 6. 离线操作队列 (断网时记录待同步操作)
/// 7. 加密存储敏感数据

import 'dart:async';
import 'dart:convert';

/* ===== 数据模型 ===== */

/// 本地缓存的作业记录
class CachedAssignment {
  final String id;
  final String title;
  final String subject;
  final String classId;
  final int publishTime;
  final int deadline;
  final int status;
  final String detailJson; // 完整作业详情JSON (包含题目列表)
  final int cachedAt; // 缓存时间

  CachedAssignment({
    required this.id,
    required this.title,
    required this.subject,
    required this.classId,
    required this.publishTime,
    required this.deadline,
    required this.status,
    required this.detailJson,
    required this.cachedAt,
  });

  Map<String, dynamic> toMap() => {
    'id': id, 'title': title, 'subject': subject,
    'class_id': classId, 'publish_time': publishTime,
    'deadline': deadline, 'status': status,
    'detail_json': detailJson, 'cached_at': cachedAt,
  };

  factory CachedAssignment.fromMap(Map<String, dynamic> map) {
    return CachedAssignment(
      id: map['id'] ?? '',
      title: map['title'] ?? '',
      subject: map['subject'] ?? '',
      classId: map['class_id'] ?? '',
      publishTime: map['publish_time'] ?? 0,
      deadline: map['deadline'] ?? 0,
    );
  }
}

```



```

        status: map['status'] ?? 0,
        detailJson: map['detail_json'] ?? '{}',
        cachedAt: map['cached_at'] ?? 0,
    );
}
}

/// 本地缓存的消息记录
class CachedMessage {
    final String id;
    final String fromUserId;
    final String fromUserName;
    final String content;
    final String type;           // text / image / assignment / report
    final int sendTime;
    final bool isRead;
    final String extraJson;      // 附加数据（如关联的作业ID、学情ID）

    CachedMessage({
        required this.id,
        required this.fromUserId,
        required this.fromUserName,
        required this.content,
        required this.type,
        required this.sendTime,
        required this.isRead,
        required this.extraJson,
    });

    Map<String, dynamic> toMap() => {
        'id': id, 'from_user_id': fromUserId,
        'from_user_name': fromUserName,
        'content': content, 'type': type,
        'send_time': sendTime, 'is_read': isRead ? 1 : 0,
        'extra_json': extraJson,
    };

    factory CachedMessage.fromMap(Map<String, dynamic> map) {
        return CachedMessage(
            id: map['id'] ?? '',
            fromUserId: map['from_user_id'] ?? '',
            fromUserName: map['from_user_name'] ?? '',
            content: map['content'] ?? '',
            type: map['type'] ?? 'text',
            sendTime: map['send_time'] ?? 0,
            isRead: (map['is_read'] ?? 0) == 1,
            extraJson: map['extra_json'] ?? '{}',
        );
    }
}

/// 待同步的离线操作
class OfflineAction {
    final String id;
    final String actionType;      // upload_stroke / submit_answer / send_message
    final String targetApi;       // 目标API路径
    final String method;          // HTTP方法

```

```

final String payloadJson;    // 请求体JSON
final int createdAt;
final int retryCount;

OfflineAction({
  required this.id,
  required this.actionType,
  required this.targetApi,
  required this.method,
  required this.payloadJson,
  required this.createdAt,
  this.retryCount = 0,
});

Map<String, dynamic> toMap() => {
  'id': id, 'action_type': actionType,
  'target_api': targetApi, 'method': method,
  'payload_json': payloadJson,
  'created_at': createdAt, 'retry_count': retryCount,
};

factory OfflineAction.fromMap(Map<String, dynamic> map) {
  return OfflineAction(
    id: map['id'] ?? '',
    actionType: map['action_type'] ?? '',
    targetApi: map['target_api'] ?? '',
    method: map['method'] ?? 'POST',
    payloadJson: map['payload_json'] ?? '{}',
    createdAt: map['created_at'] ?? 0,
    retryCount: map['retry_count'] ?? 0,
  );
}

}

/// 暂存的笔迹数据（等待上传）
class PendingStrokeData {
  final String id;
  final String deviceId;      // 笔设备ID
  final String assignmentId;  // 关联作业ID
  final String studentId;    // 学生ID
  final String strokeJson;    // 笔迹坐标JSON
  final int collectTime;      // 采集时间
  final int syncStatus;       // 0=待上传, 1=已上传, 2=上传失败

  PendingStrokeData({
    required this.id,
    required this.deviceId,
    required this.assignmentId,
    required this.studentId,
    required this.strokeJson,
    required this.collectTime,
    this.syncStatus = 0,
  });

  Map<String, dynamic> toMap() => {
    'id': id, 'device_id': deviceId,
    'assignment_id': assignmentId, 'student_id': studentId,

```

```

        'stroke_json': strokeJson, 'collect_time': collectTime,
        'sync_status': syncStatus,
    };

    factory PendingStrokeData.fromMap(Map<String, dynamic> map) {
        return PendingStrokeData(
            id: map['id'] ?? '',
            deviceId: map['device_id'] ?? '',
            assignmentId: map['assignment_id'] ?? '',
            studentId: map['student_id'] ?? '',
            strokeJson: map['stroke_json'] ?? '[]',
            collectTime: map['collect_time'] ?? 0,
            syncStatus: map['sync_status'] ?? 0,
        );
    }
}

/* ===== 本地仓库实现 ===== */

/// 本地数据仓库 - 管理SQLite数据库CRUD操作
class LocalDataRepository {
    /// 数据库实例 (sqflite Database对象)
    dynamic _db;

    /// 数据库版本号
    static const int _dbVersion = 3;

    /// 数据库文件名
    static const String _dbName = 'writech_mobile.db';

    /// 初始化数据库
    /// 创建表结构, 执行版本迁移
    Future<void> initialize() async {
        // 实际使用sqflite打开数据库
        // _db = await openDatabase(path, version: _dbVersion, onCreate: _onCreate,
onUpgrade: _onUpgrade);
        print('[LocalRepo] 数据库初始化完成, 版本: $_dbVersion');
    }

    /// 创建初始表结构 (首次安装执行)
    Future<void> _onCreate(dynamic db, int version) async {
        // 作业缓存表
        await db.execute('''
            CREATE TABLE cached_assignments (
                id TEXT PRIMARY KEY,
                title TEXT NOT NULL,
                subject TEXT DEFAULT '',
                class_id TEXT NOT NULL,
                publish_time INTEGER NOT NULL,
                deadline INTEGER NOT NULL,
                status INTEGER DEFAULT 0,
                detail_json TEXT DEFAULT '{}',
                cached_at INTEGER NOT NULL
            )
        ''');

        // 消息记录表

```

```

await db.execute('''
    CREATE TABLE cached_messages (
        id TEXT PRIMARY KEY,
        from_user_id TEXT NOT NULL,
        from_user_name TEXT DEFAULT '',
        content TEXT NOT NULL,
        type TEXT DEFAULT 'text',
        send_time INTEGER NOT NULL,
        is_read INTEGER DEFAULT 0,
        extra_json TEXT DEFAULT '{}'
    )
''');

// 离线操作队列表
await db.execute('''
    CREATE TABLE offline_actions (
        id TEXT PRIMARY KEY,
        action_type TEXT NOT NULL,
        target_api TEXT NOT NULL,
        method TEXT DEFAULT 'POST',
        payload_json TEXT NOT NULL,
        created_at INTEGER NOT NULL,
        retry_count INTEGER DEFAULT 0
    )
''');

// 笔迹暂存表
await db.execute('''
    CREATE TABLE pending_strokes (
        id TEXT PRIMARY KEY,
        device_id TEXT NOT NULL,
        assignment_id TEXT NOT NULL,
        student_id TEXT DEFAULT '',
        stroke_json TEXT NOT NULL,
        collect_time INTEGER NOT NULL,
        sync_status INTEGER DEFAULT 0
    )
''');

// 学情报告缓存表
await db.execute('''
    CREATE TABLE cached_reports (
        student_id TEXT NOT NULL,
        subject TEXT NOT NULL,
        report_json TEXT NOT NULL,
        cached_at INTEGER NOT NULL,
        PRIMARY KEY (student_id, subject)
    )
''');

// 创建索引
await db.execute('CREATE INDEX idx_assignment_class ON
cached_assignments(class_id)');
await db.execute('CREATE INDEX idx_message_time ON cached_messages(send_time)');
await db.execute('CREATE INDEX idx_stroke_sync ON pending_strokes(sync_status)');

print('[LocalRepo] 数据库表创建完成');

```

```

}

/// 版本升级迁移
Future<void> _onUpgrade(dynamic db, int oldVersion, int newVersion) async {
  if (oldVersion < 2) {
    // v2: 添加学情报告缓存表
    await db.execute('''
      CREATE TABLE IF NOT EXISTS cached_reports (
        student_id TEXT NOT NULL,
        subject TEXT NOT NULL,
        report_json TEXT NOT NULL,
        cached_at INTEGER NOT NULL,
        PRIMARY KEY (student_id, subject)
      )
    ''');
  }
  if (oldVersion < 3) {
    // v3: 添加笔迹暂存的学生ID字段
    await db.execute('ALTER TABLE pending_strokes ADD COLUMN student_id TEXT DEFAULT
    ""');
  }
  print('[LocalRepo] 数据库升级: v$oldVersion -> v$newVersion');
}

/* ===== 作业缓存操作 ===== */

/// 批量缓存作业列表（从云端拉取后存储到本地）
Future<void> cacheAssignments(List<CachedAssignment> assignments) async {
  // 使用事务批量插入，提高性能
  // await _db.transaction((txn) async { ... });
  for (final a in assignments) {
    // INSERT OR REPLACE
    print('[LocalRepo] 缓存作业: ${a.title}');
  }
}

/// 查询本地缓存的作业列表
Future<List<CachedAssignment>> getAssignmentsByClass(String classId, {int limit = 50})
async {
  // SELECT * FROM cached_assignments WHERE class_id = ? ORDER BY publish_time DESC
  LIMIT ?
  return [];
}

/// 获取作业详情（优先从缓存读取）
Future<CachedAssignment?> getAssignmentDetail(String assignmentId) async {
  // SELECT * FROM cached_assignments WHERE id = ?
  return null;
}

/// 清理过期的作业缓存（30天前的数据）
Future<int> cleanExpiredAssignments() async {
  final threshold = DateTime.now().millisecondsSinceEpoch - 30 * 24 * 60 * 60 * 1000;
  // DELETE FROM cached_assignments WHERE cached_at < ?
  print('[LocalRepo] 清理过期作业缓存');
  return 0;
}

```

```

/* ===== 消息记录操作 ===== */

/// 保存消息到本地
Future<void> saveMessage(CachedMessage message) async {
    // INSERT OR REPLACE INTO cached_messages VALUES (...)
    print('[LocalRepo] 保存消息: ${message.id}');
}

/// 查询消息列表 (分页)
Future<List<CachedMessage>> getMessages({int page = 0, int pageSize = 20}) async {
    // SELECT * FROM cached_messages ORDER BY send_time DESC LIMIT ? OFFSET ?
    return [];
}

/// 标记消息已读
Future<void> markMessageRead(String messageId) async {
    // UPDATE cached_messages SET is_read = 1 WHERE id = ?
}

/// 获取未读消息数量
Future<int> getUnreadCount() async {
    // SELECT COUNT(*) FROM cached_messages WHERE is_read = 0
    return 0;
}

/* ===== 离线操作队列 ===== */

/// 添加离线操作到队列 (断网时调用)
Future<void> enqueueOfflineAction(OfflineAction action) async {
    // INSERT INTO offline_actions VALUES (...)
    print('[LocalRepo] 离线操作入队: ${action.actionType}');
}

/// 获取所有待执行的离线操作
Future<List<OfflineAction>> getPendingOfflineActions() async {
    // SELECT * FROM offline_actions ORDER BY created_at ASC
    return [];
}

/// 删除已完成的离线操作
Future<void> removeOfflineAction(String actionId) async {
    // DELETE FROM offline_actions WHERE id = ?
}

/// 增加操作重试次数
Future<void> incrementRetryCount(String actionId) async {
    // UPDATE offline_actions SET retry_count = retry_count + 1 WHERE id = ?
}

/* ===== 笔迹暂存操作 ===== */

/// 暂存笔迹数据 (BLE收笔后等待上传)
Future<void> savePendingStroke(PendingStrokeData stroke) async {
    // INSERT INTO pending_strokes VALUES (...)
    print('[LocalRepo] 暂存笔迹数据: ${stroke.id}');
}

```

```

/// 获取待上传的笔迹数据
Future<List<PendingStrokeData>> getUnsyncedStrokes({int limit = 50}) async {
  // SELECT * FROM pending_strokes WHERE sync_status = 0 LIMIT ?
  return [];
}

/// 更新笔迹同步状态
Future<void> updateStrokeSyncStatus(String strokeId, int status) async {
  // UPDATE pending_strokes SET sync_status = ? WHERE id = ?
}

/// 批量删除已上传的笔迹
Future<int> cleanSyncedStrokes() async {
  // DELETE FROM pending_strokes WHERE sync_status = 1
  return 0;
}

/* ===== 学情报告缓存 ===== */

/// 缓存学情报告
Future<void> cacheReport(String studentId, String subject, Map<String, dynamic>
report) async {
  final reportJson = jsonEncode(report);
  // INSERT OR REPLACE INTO cached_reports VALUES (studentId, subject, reportJson,
now)
  print('[LocalRepo] 缓存学情报告: $studentId/$subject');
}

/// 获取缓存的学情报告
Future<Map<String, dynamic>?> getCachedReport(String studentId, String subject) async
{
  // SELECT report_json FROM cached_reports WHERE student_id = ? AND subject = ?
  return null;
}

/* ===== 数据库维护 ===== */

/// 获取数据库统计信息
Future<Map<String, int>> getStatistics() async {
  return {
    'assignments': 0, // 缓存作业数
    'messages': 0,    // 消息数
    'offlineActions': 0, // 待同步操作数
    'pendingStrokes': 0, // 待上传笔迹数
  };
}

/// 清空所有本地数据（用户登出时调用）
Future<void> clearAll() async {
  // DELETE FROM cached_assignments
  // DELETE FROM cached_messages
  // DELETE FROM offline_actions
  // DELETE FROM pending_strokes
  // DELETE FROM cached_reports
  print('[LocalRepo] 已清空所有本地数据');
}

```

```

    /// 关闭数据库连接
    Future<void> close() async {
      // await _db?.close();
      print('[LocalRepo] 数据库连接已关闭');
    }
  }
}

```

## service/

### service/api\_service.dart

```

/// 自然写互动课堂手机端应用软件 V1.0
/// 云平台API服务 - 封装所有REST API通信逻辑
///
/// 功能说明:
/// 1. HTTP客户端配置 (Dio拦截器、超时设置、重试策略)
/// 2. JWT Token自动管理 (存储、刷新、过期处理)
/// 3. 请求签名 (HMAC-SHA256防篡改)
/// 4. 证书锁定 (Certificate Pinning防中间人攻击)
/// 5. 全部业务API封装 (登录、作业、学情、消息等)
/// 6. 离线请求队列 (断网时暂存请求, 恢复后自动重放)

import 'dart:async';
import 'dart:convert';
import 'dart:io';
import 'package:crypto/crypto.dart';

/* ===== 数据模型 ===== */

/// API响应统一包装
class ApiResponse<T> {
  final int code;          // 业务状态码 (0=成功)
  final String message;    // 状态消息
  final T? data;           // 响应数据
  final int timestamp;     // 服务端时间戳

  ApiResponse({
    required this.code,
    required this.message,
    this.data,
    required this.timestamp,
  });

  /// 判断请求是否成功
  bool get isSuccess => code == 0;

  /// 从JSON反序列化
  factory ApiResponse.fromJson(Map<String, dynamic> json, T Function(dynamic)? fromData)
  {
    return ApiResponse(
      code: json['code'] ?? -1,
      message: json['message'] ?? '',

```



```

        data: json['data'] != null && fromData != null ? fromData(json['data']) : null,
        timestamp: json['timestamp'] ?? 0,
    );
}
}

/// 用户登录凭证
class AuthToken {
    final String accessToken;    // 访问令牌 (有效期2小时)
    final String refreshToken;   // 刷新令牌 (有效期7天)
    final int expiresAt;        // 访问令牌过期时间戳 (毫秒)
    final String userRole;      // 用户角色: teacher / parent / admin

    AuthToken({
        required this.accessToken,
        required this.refreshToken,
        required this.expiresAt,
        required this.userRole,
    });

    /// 判断Token是否即将过期 (提前5分钟刷新)
    bool get isExpiringSoon {
        return DateTime.now().millisecondsSinceEpoch > (expiresAt - 5 * 60 * 1000);
    }

    factory AuthToken.fromJson(Map<String, dynamic> json) {
        return AuthToken(
            accessToken: json['access_token'] ?? '',
            refreshToken: json['refresh_token'] ?? '',
            expiresAt: json['expires_at'] ?? 0,
            userRole: json['user_role'] ?? '',
        );
    }

    Map<String, dynamic> toJson() => {
        'access_token': accessToken,
        'refresh_token': refreshToken,
        'expires_at': expiresAt,
        'user_role': userRole,
    };
}

/// 用户信息模型
class UserInfo {
    final String userId;
    final String name;
    final String avatar;
    final String role;
    final String phone;
    final List<String> classIds;    // 关联的班级ID列表

    UserInfo({
        required this.userId,
        required this.name,
        required this.avatar,
        required this.role,
        required this.phone,

```

```

        required this.classIds,
    });

    factory UserInfo.fromJson(Map<String, dynamic> json) {
        return UserInfo(
            userId: json['user_id'] ?? '',
            name: json['name'] ?? '',
            avatar: json['avatar'] ?? '',
            role: json['role'] ?? '',
            phone: json['phone'] ?? '',
            classIds: List<String>.from(json['class_ids'] ?? []),
        );
    }
}

/// 作业信息模型
class AssignmentInfo {
    final String id;
    final String title;
    final String subject;           // 科目
    final String type;              // 类型: homework / exam / practice
    final String classId;
    final int publishTime;          // 发布时间
    final int deadline;             // 截止时间
    final int submittedCount;       // 已提交人数
    final int totalCount;          // 应提交人数
    final int status;               // 0=进行中, 1=已截止, 2=已批改

    AssignmentInfo({
        required this.id,
        required this.title,
        required this.subject,
        required this.type,
        required this.classId,
        required this.publishTime,
        required this.deadline,
        required this.submittedCount,
        required this.totalCount,
        required this.status,
    });

    factory AssignmentInfo.fromJson(Map<String, dynamic> json) {
        return AssignmentInfo(
            id: json['id'] ?? '',
            title: json['title'] ?? '',
            subject: json['subject'] ?? '',
            type: json['type'] ?? '',
            classId: json['class_id'] ?? '',
            publishTime: json['publish_time'] ?? 0,
            deadline: json['deadline'] ?? 0,
            submittedCount: json['submitted_count'] ?? 0,
            totalCount: json['total_count'] ?? 0,
            status: json['status'] ?? 0,
        );
    }
}

```

```

/// 学情报告模型
class LearningReport {
  final String studentId;
  final String studentName;
  final String subject;
  final double overallScore;          // 综合评分 (0-100)
  final Map<String, double> knowledgeMap; // 知识点掌握度
  final List<ErrorItem> topErrors;      // 高频错题
  final WritingGrowth writingGrowth;    // 书写成长数据

  LearningReport({
    required this.studentId,
    required this.studentName,
    required this.subject,
    required this.overallScore,
    required this.knowledgeMap,
    required this.topErrors,
    required this.writingGrowth,
  });

  factory LearningReport.fromJson(Map<String, dynamic> json) {
    return LearningReport(
      studentId: json['student_id'] ?? '',
      studentName: json['student_name'] ?? '',
      subject: json['subject'] ?? '',
      overallScore: (json['overall_score'] ?? 0).toDouble(),
      knowledgeMap: Map<String, double>.from(json['knowledge_map'] ?? {}),
      topErrors: (json['top_errors'] as List? ?? [])
        .map((e) => ErrorItem.fromJson(e))
        .toList(),
      writingGrowth: WritingGrowth.fromJson(json['writing_growth'] ?? {}),
    );
  }
}

/// 错题条目
class ErrorItem {
  final String questionId;
  final String content;
  final String knowledgePoint;
  final int errorCount;
  final String errorReason;

  ErrorItem({
    required this.questionId,
    required this.content,
    required this.knowledgePoint,
    required this.errorCount,
    required this.errorReason,
  });

  factory ErrorItem.fromJson(Map<String, dynamic> json) {
    return ErrorItem(
      questionId: json['question_id'] ?? '',
      content: json['content'] ?? '',
      knowledgePoint: json['knowledge_point'] ?? '',
      errorCount: json['error_count'] ?? 0,
    );
  }
}

```

```

        errorReason: json['error_reason'] ?? '',
    );
}
}

/// 书写成长数据
class WritingGrowth {
    final List<double> scores;      // 历次书写评分
    final List<String> dates;      // 对应日期
    final double strokeAccuracy;   // 笔顺正确率
    final double writingNeatness;  // 书写规范性
    final String improvement;      // 进步趋势描述

    WritingGrowth({
        required this.scores,
        required this.dates,
        required this.strokeAccuracy,
        required this.writingNeatness,
        required this.improvement,
    });

    factory WritingGrowth.fromJson(Map<String, dynamic> json) {
        return WritingGrowth(
            scores: List<double>.from(json['scores'] ?? []),
            dates: List<String>.from(json['dates'] ?? []),
            strokeAccuracy: (json['stroke_accuracy'] ?? 0).toDouble(),
            writingNeatness: (json['writing_neatness'] ?? 0).toDouble(),
            improvement: json['improvement'] ?? '',
        );
    }
}

/* ===== API服务实现 ===== */

/// 云平台API服务 - 管理所有HTTP通信
/// 采用Dio作为HTTP客户端，支持拦截器链、证书锁定、自动重试
class CloudApiService {
    /// 云平台API基础地址
    static const String _baseUrl = 'https://api.writech.com/v1';

    /// HMAC签名密钥（从安全存储中加载）
    final String _hmacSecret;

    /// 当前认证令牌
    AuthToken? _authToken;

    /// Token刷新锁（防止并发刷新）
    bool _isRefreshing = false;
    final List<Function> _refreshQueue = [];

    /// HTTP客户端实例
    late final HttpClient _httpClient;

    /// 离线请求队列（断网时暂存）
    final List<Map<String, dynamic>> _offlineQueue = [];

    /// 最大重试次数

```

```

static const int _maxRetries = 3;

CloudApiService({String hmacSecret = ''}) : _hmacSecret = hmacSecret {
    _httpClient = HttpClient()
        ..connectionTimeout = const Duration(seconds: 15)
        ..idleTimeout = const Duration(seconds: 60);

    // 配置证书锁定（防止中间人攻击）
    _httpClient.badCertificateCallback = (X509Certificate cert, String host, int port) {
        // 验证证书指纹是否匹配预置的服务器证书
        final fingerprint = sha256.convert(cert.der).toString();
        const expectedFingerprint = 'a1b2c3d4e5f6...'; // 预置证书指纹
        return fingerprint == expectedFingerprint;
    };
}

/// 设置认证令牌（登录成功后调用）
void setAuthToken(AuthToken token) {
    _authToken = token;
}

/// 生成请求签名（HMAC-SHA256）
/// 签名内容：METHOD + PATH + TIMESTAMP + BODY_HASH
String _generateSignature(String method, String path, int timestamp, String body) {
    final bodyHash = sha256.convert(utf8.encode(body)).toString();
    final content = '$method\n$path\n$timestamp\n$bodyHash';
    final hmacSha256 = Hmac(sha256, utf8.encode(_hmacSecret));
    return hmacSha256.convert(utf8.encode(content)).toString();
}

/// 统一HTTP请求方法（带签名、Token、重试）
Future<ApiResponse<T>> _request<T>({
    required String method,
    required String path,
    Map<String, dynamic>? queryParams,
    Map<String, dynamic>? body,
    T Function(dynamic)? fromData,
    int retryCount = 0,
}) async {
    // 检查Token是否需要刷新
    if (_authToken != null && _authToken!.isExpiringSoon) {
        await _refreshToken();
    }

    final uri = Uri.parse('$_baseUrl$path').replace(queryParameters:
        queryParams?.map((k, v) => MapEntry(k, v.toString())));
    final timestamp = DateTime.now().millisecondsSinceEpoch;
    final bodyStr = body != null ? jsonEncode(body) : '';
    final signature = _generateSignature(method, path, timestamp, bodyStr);

    try {
        final request = await _httpClient.openUrl(method, uri);

        // 设置请求头
        request.headers.set('Content-Type', 'application/json');
        request.headers.set('X-Timestamp', timestamp.toString());
        request.headers.set('X-Signature', signature);
    }
}

```

```

request.headers.set('X-Client', 'writech-mobile/1.0');
if (_authToken != null) {
    request.headers.set('Authorization', 'Bearer ${_authToken!.accessToken}');
}

// 写入请求体
if (body != null) {
    request.write(bodyStr);
}

// 发送请求并接收响应
final response = await request.close();
final responseBody = await response.transform(utf8.decoder).join();
final jsonData = jsonDecode(responseBody) as Map<String, dynamic>;

// 处理401未授权 (Token过期)
if (response.statusCode == 401 && retryCount < 1) {
    await _refreshToken();
    return _request(
        method: method, path: path, queryParams: queryParams,
        body: body, fromData: fromData, retryCount: retryCount + 1,
    );
}

return ApiResponse.fromJson(jsonData, fromData);
} on SocketException {
    // 网络不可用, 加入离线队列
    if (method == 'POST' || method == 'PUT') {
        _offlineQueue.add({
            'method': method, 'path': path,
            'body': body, 'timestamp': timestamp,
        });
    }
    return ApiResponse(code: -1, message: '网络连接不可用', timestamp: timestamp);
} catch (e) {
    // 重试逻辑 (指数退避)
    if (retryCount < _maxRetries) {
        await Future.delayed(Duration(seconds: 1 << retryCount));
        return _request(
            method: method, path: path, queryParams: queryParams,
            body: body, fromData: fromData, retryCount: retryCount + 1,
        );
    }
    return ApiResponse(code: -1, message: '请求失败: $e', timestamp: timestamp);
}
}

/// 刷新Token (使用Refresh Token获取新的Access Token)
Future<void> _refreshToken() async {
    if (_isRefreshing) {
        // 等待正在进行的刷新完成
        final completer = Completer<void>();
        _refreshQueue.add(() => completer.complete());
        return completer.future;
    }

    _isRefreshing = true;

```

```

try {
    final response = await _request<AuthToken>(
        method: 'POST',
        path: '/auth/refresh',
        body: {'refresh_token': _authToken?.refreshToken ?? ''},
        fromData: (data) => AuthToken.fromJson(data),
    );

    if (response.isSuccess && response.data != null) {
        _authToken = response.data;
        // 持久化新Token到安全存储
        _persistToken(_authToken!);
    }
} finally {
    _isRefreshing = false;
    // 通知所有等待的请求继续
    for (final callback in _refreshQueue) {
        callback();
    }
    _refreshQueue.clear();
}
}

/// 持久化Token到Keychain/KeyStore
void _persistToken(AuthToken token) {
    // 使用flutter_secure_storage存储到系统安全存储
    // iOS: Keychain  Android: KeyStore
}

/// 重放离线队列中的请求（网络恢复后调用）
Future<int> replayOfflineQueue() async {
    int successCount = 0;
    final queue = List<Map<String, dynamic>>.from(_offlineQueue);
    _offlineQueue.clear();

    for (final item in queue) {
        final response = await _request(
            method: item['method'],
            path: item['path'],
            body: item['body'],
        );
        if (response.isSuccess) successCount++;
    }
    return successCount;
}

/* ===== 认证相关API ===== */

/// 手机号+验证码登录
Future<ApiResponse<AuthToken>> loginByPhone(String phone, String code) {
    return _request(
        method: 'POST',
        path: '/auth/login/phone',
        body: {'phone': phone, 'code': code},
        fromData: (data) => AuthToken.fromJson(data),
    );
}

```

```

/// 微信OAuth登录
Future<ApiResponse<AuthToken>> loginByWechat(String wxCode) {
  return _request(
    method: 'POST',
    path: '/auth/login/wechat',
    body: {'wx_code': wxCode},
    fromData: (data) => AuthToken.fromJson(data),
  );
}

/// 获取当前用户信息
Future<ApiResponse<UserInfo>> getUserInfo() {
  return _request(
    method: 'GET',
    path: '/user/profile',
    fromData: (data) => UserInfo.fromJson(data),
  );
}

/// 登出 (撤销Token)
Future<ApiResponse> logout() {
  return _request(method: 'POST', path: '/auth/logout');
}

/* ===== 作业相关API ===== */

/// 获取作业列表 (教师端)
Future<ApiResponse<List<AssignmentInfo>>> getAssignmentList({
  required String classId,
  int page = 1,
  int pageSize = 20,
  String? status,
}) {
  return _request(
    method: 'GET',
    path: '/assignment/list',
    queryParams: {
      'class_id': classId,
      'page': page,
      'page_size': pageSize,
      if (status != null) 'status': status,
    },
    fromData: (data) => (data as List)
      .map((e) => AssignmentInfo.fromJson(e))
      .toList(),
  );
}

/// 发布新作业 (教师端)
Future<ApiResponse<String>> publishAssignment({
  required String title,
  required String classId,
  required String subject,
  required int deadline,
  required List<Map<String, dynamic>> questions,
}) {

```



```

        return _request(
            method: 'POST',
            path: '/assignment/publish',
            body: {
                'title': title,
                'class_id': classId,
                'subject': subject,
                'deadline': deadline,
                'questions': questions,
            },
        );
    }

    /* ===== 学情报告API ===== */

    /// 获取学生学情报告（家长端/教师端）
    Future<ApiResponse<LearningReport>> getStudentReport(String studentId, {String?
subject}) {
        return _request(
            method: 'GET',
            path: '/report/student/$studentId',
            queryParams: subject != null ? {'subject': subject} : null,
            fromData: (data) => LearningReport.fromJson(data),
        );
    }

    /// 获取班级学情概览（教师端）
    Future<ApiResponse<Map<String, dynamic>>> getClassReport(String classId) {
        return _request(
            method: 'GET',
            path: '/report/class/$classId',
        );
    }

    /* ===== 消息通知API ===== */

    /// 获取消息列表
    Future<ApiResponse<List<Map<String, dynamic>>>> getMessageList({
        int page = 1,
        int pageSize = 20,
    }) {
        return _request(
            method: 'GET',
            path: '/message/list',
            queryParams: {'page': page, 'page_size': pageSize},
        );
    }

    /// 发送家校沟通消息（教师→家长）
    Future<ApiResponse> sendMessage({
        required String toUserId,
        required String content,
        String type = 'text',
    }) {
        return _request(
            method: 'POST',
            path: '/message/send',

```

```

        body: {'to_user_id': toUserId, 'content': content, 'type': type},
    );
}

/// 标记消息已读
Future<ApiResponse> markMessageRead(List<String> messageIds) {
    return _request(
        method: 'PUT',
        path: '/message/read',
        body: {'message_ids': messageIds},
    );
}

/* ===== 笔迹数据API ===== */

/// 上传笔迹数据（教师端蓝牙收笔后上传）
Future<ApiResponse<String>> uploadStrokeData({
    required String assignmentId,
    required String studentId,
    required List<Map<String, dynamic>> strokes,
}) {
    return _request(
        method: 'POST',
        path: '/stroke/upload',
        body: {
            'assignment_id': assignmentId,
            'student_id': studentId,
            'strokes': strokes,
            'client_time': DateTime.now().millisecondsSinceEpoch,
        },
    );
}

/// 获取笔迹回放数据
Future<ApiResponse<List<Map<String, dynamic>>>> getStrokeReplay({
    required String assignmentId,
    required String studentId,
}) {
    return _request(
        method: 'GET',
        path: '/stroke/replay',
        queryParams: {
            'assignment_id': assignmentId,
            'student_id': studentId,
        },
    );
}

/// 销毁HTTP客户端
void dispose() {
    _httpClient.close();
    _offlineQueue.clear();
    _refreshQueue.clear();
}
}

```

## service/ble\_service.dart

```
/// 自然写互动课堂手机端应用软件 V1.0
/// BLE蓝牙服务 - 教师端蓝牙连接点阵笔进行移动教学
///
/// 功能说明:
/// 1. BLE设备扫描与发现 (按自然写笔设备UUID过滤)
/// 2. GATT连接与特征值订阅 (实时接收笔迹坐标数据)
/// 3. 7字节紧凑坐标数据解码 (x:16bit, y:16bit, pressure:8bit, timestamp:16bit)
/// 4. 多笔同时连接管理 (教师端移动教学最多连接4支笔)
/// 5. 自动重连与连接状态监控
/// 6. 设备电量读取与低电量告警
/// 7. 蓝牙权限检查与引导
/// 8. 笔迹数据缓冲与批量回调

import 'dart:async';
import 'dart:typed_data';

/* ===== BLE协议常量定义 ===== */

/// 自然写点阵笔BLE服务UUID
class WritechBleUuids {
  /// 主服务UUID - 笔迹数据传输
  static const String strokeServiceUuid = '6E400001-B5A3-F393-E0A9-E50E24DCCA9E';
  /// 笔迹数据特征值UUID (Notify模式, 笔到手机)
  static const String strokeDataCharUuid = '6E400003-B5A3-F393-E0A9-E50E24DCCA9E';
  /// 命令写入特征值UUID (Write模式, 手机到笔)
  static const String commandCharUuid = '6E400002-B5A3-F393-E0A9-E50E24DCCA9E';
  /// 设备信息服务UUID (标准BLE Device Information Service)
  static const String deviceInfoServiceUuid = '0000180A-0000-1000-8000-00805F9B34FB';
  /// 电池服务UUID (标准BLE Battery Service)
  static const String batteryServiceUuid = '0000180F-0000-1000-8000-00805F9B34FB';
  /// 电池电量特征值UUID
  static const String batteryLevelCharUuid = '00002A19-0000-1000-8000-00805F9B34FB';
}

/// BLE笔命令定义
class PenCommand {
  static const int cmdSetMode = 0x01;
  static const int cmdGetStatus = 0x02;
  static const int cmdSyncOffline = 0x03;
  static const int cmdSetName = 0x04;
  static const int cmdStartOta = 0x05;
  static const int cmdReset = 0xFF;
}

/* ===== 数据模型 ===== */

/// BLE笔设备信息
class PenDevice {
  final String deviceId;
  final String name;
  int rssi;
  int batteryLevel;
  String firmwareVersion;
}
```

```

PenConnectionState state;
DateTime? lastActiveTime;
int offlineDataCount;

PenDevice({
  required this.deviceId,
  required this.name,
  this.rssi = -100,
  this.batteryLevel = -1,
  this.firmwareVersion = '',
  this.state = PenConnectionState.disconnected,
  this.lastActiveTime,
  this.offlineDataCount = 0,
});
}

/// 笔连接状态枚举
enum PenConnectionState {
  disconnected,
  connecting,
  connected,
  disconnecting,
}

/// 笔迹坐标点（从BLE数据解码后的结构化数据）
class StrokePoint {
  final double x;
  final double y;
  final double pressure;
  final int timestamp;
  final bool isPenDown;

  const StrokePoint({
    required this.x,
    required this.y,
    required this.pressure,
    required this.timestamp,
    required this.isPenDown,
  });

  Map<String, dynamic> toJson() => {
    'x': x, 'y': y,
    'pressure': pressure,
    'timestamp': timestamp,
    'pen_down': isPenDown,
  };
}

/// 笔迹数据回调事件
class StrokeDataEvent {
  final String deviceId;
  final List<StrokePoint> points;
  final int pageId;

  StrokeDataEvent({
    required this.deviceId,
    required this.points,
  });
}

```

```

        required this.pageId,
    });
}

/* ===== BLE服务实现 ===== */

/// BLE蓝牙服务 - 管理点阵笔的蓝牙连接与数据传输
class BleConnectionService {
    /// 已连接或已发现的笔设备列表
    final Map<String, PenDevice> _devices = {};

    /// 笔迹数据流控制器（向上层广播解码后的笔迹坐标）
    final StreamController<StrokeDataEvent> _strokeStreamController =
        StreamController<StrokeDataEvent>.broadcast();

    /// 设备状态变化流
    final StreamController<PenDevice> _deviceStateController =
        StreamController<PenDevice>.broadcast();

    /// 扫描状态
    bool _isScanning = false;

    /// 最大同时连接数（教师移动教学最多4支笔）
    static const int maxConnections = 4;

    /// 自动重连间隔（秒）
    static const int reconnectIntervalSec = 5;

    /// 数据缓冲区大小（累积到一定量后批量回调）
    static const int batchSize = 10;

    /// 设备活跃超时时间（毫秒）
    static const int activeTimeoutMs = 30000;

    /// 低电量告警阈值
    static const int lowBatteryThreshold = 10;

    /// 重连计时器
    final Map<String, Timer> _reconnectTimers = {};

    /// 电量查询计时器
    Timer? _batteryCheckTimer;

    /// 笔迹数据缓冲区（按设备ID分组）
    final Map<String, List<StrokePoint>> _dataBuffers = {};

    /// 外部可订阅的笔迹数据流
    Stream<StrokeDataEvent> get strokeStream => _strokeStreamController.stream;

    /// 外部可订阅的设备状态流
    Stream<PenDevice> get deviceStateStream => _deviceStateController.stream;

    /// 获取当前已连接设备数量
    int get connectedCount =>
        _devices.values.where((d) => d.state == PenConnectionState.connected).length;

    /// 获取所有已发现设备列表

```

```

List<PenDevice> get discoveredDevices => _devices.values.toList();

/// 开始BLE扫描（发现周围的自然写点阵笔设备）
/// 仅扫描包含自然写笔服务UUID的设备，过滤无关BLE设备
Future<void> startScan({Duration timeout = const Duration(seconds: 10)}) async {
  if (_isScanning) {
    print('[BLE] 已在扫描中，忽略重复请求');
    return;
  }

  // 检查蓝牙权限和状态
  final hasPermission = await _checkBluetoothPermission();
  if (!hasPermission) {
    print('[BLE] 蓝牙权限未授予，无法扫描');
    return;
  }

  _isScanning = true;
  print('[BLE] 开始扫描自然写点阵笔设备...');

  // 使用flutter_blue扫描指定服务UUID的设备
  // 实际实现通过FlutterBluePlus.startScan()
  // 此处模拟扫描逻辑
  Timer(timeout, () {
    stopScan();
  });
}

/// 停止BLE扫描
void stopScan() {
  if (!_isScanning) return;
  _isScanning = false;
  print('[BLE] 停止扫描');
}

/// 处理扫描到的设备广播数据
/// 解析设备名称、信号强度、服务UUID
void _onDeviceDiscovered(String deviceId, String name, int rssi, List<String>
serviceUuids) {
  // 仅处理包含自然写笔服务UUID的设备
  if (!serviceUuids.contains(WrittechBleUuids.strokeServiceUuid)) return;

  if (_devices.containsKey(deviceId)) {
    // 更新已知设备的RSSI
    _devices[deviceId]!.rssi = rssi;
  } else {
    // 发现新设备
    final device = PenDevice(
      deviceId: deviceId,
      name: name.isNotEmpty ? name : '未知笔设备',
      rssi: rssi,
    );
    _devices[deviceId] = device;
    print('[BLE] 发现新设备: $name (RSSI: $rssi)');
    _deviceStateController.add(device);
  }
}

```

```

/// 连接指定的点阵笔设备
/// 建立GATT连接, 发现服务, 订阅笔迹数据特征值
Future<bool> connectDevice(String deviceId) async {
  final device = _devices[deviceId];
  if (device == null) {
    print('[BLE] 未找到设备: $deviceId');
    return false;
  }

  // 检查连接数限制
  if (connectedCount >= maxConnections) {
    print('[BLE] 已达最大连接数限制 ($maxConnections)');
    return false;
  }

  device.state = PenConnectionState.connecting;
  _deviceStateController.add(device);
  print('[BLE] 正在连接: ${device.name}');

  try {
    // 步骤1: 建立BLE GATT连接
    // 实际调用: FlutterBluePlus.connect(device, autoConnect: false)
    await Future.delayed(const Duration(milliseconds: 500)); // 模拟连接耗时

    // 步骤2: 发现服务 (查找笔迹数据服务和电池服务)
    await _discoverServices(deviceId);

    // 步骤3: 订阅笔迹数据Notify特征值
    await _subscribeStrokeData(deviceId);

    // 步骤4: 读取初始电量
    await _readBatteryLevel(deviceId);

    // 步骤5: 读取固件版本
    await _readFirmwareVersion(deviceId);

    device.state = PenConnectionState.connected;
    device.lastActiveTime = DateTime.now();
    _deviceStateController.add(device);

    // 初始化数据缓冲区
    _dataBuffers[deviceId] = [];

    // 启动电量定时检查 (每60秒读取一次电量)
    _startBatteryCheck();

    print('[BLE] 连接成功: ${device.name}, 固件: ${device.firmwareVersion}, 电量: ${device.batteryLevel}%');
    return true;
  } catch (e) {
    device.state = PenConnectionState.disconnected;
    _deviceStateController.add(device);
    print('[BLE] 连接失败: ${device.name}, 错误: $e');

    // 设置自动重连计时器
    _scheduleReconnect(deviceId);
  }
}

```

```

        return false;
    }
}

/// 发现BLE服务列表
Future<void> _discoverServices(String deviceId) async {
    // 实际调用: device.discoverServices()
    // 验证是否包含笔迹数据服务UUID
    print('[BLE] 服务发现完成: $deviceId');
}

/// 订阅笔迹数据Notify特征值
/// 设置MTU为247字节以支持最大数据包
Future<void> _subscribeStrokeData(String deviceId) async {
    // 步骤1: 请求MTU协商 (247字节, 支持每包最多34个坐标点)
    // 实际调用: device.requestMtu(247)

    // 步骤2: 启用Notify
    // 实际调用: characteristic.setNotifyValue(true)

    // 步骤3: 监听Notify数据流
    // characteristic.onValueReceived.listen((data) => _onStrokeDataReceived(deviceId,
data))
    print('[BLE] 笔迹数据订阅成功: $deviceId');
}

/// 处理接收到的BLE笔迹原始数据包
/// 每个数据包包含1-34个7字节坐标点
/// 7字节编码格式: [x_hi, x_lo, y_hi, y_lo, pressure, ts_hi, ts_lo]
void _onStrokeDataReceived(String deviceId, Uint8List rawData) {
    final device = _devices[deviceId];
    if (device == null) return;

    // 更新设备活跃时间
    device.lastActiveTime = DateTime.now();

    // 数据包最小长度: 3字节头 + 7字节坐标 = 10字节
    if (rawData.length < 10) {
        print('[BLE] 数据包过短, 丢弃: ${rawData.length}字节');
        return;
    }

    // 解析数据包头部 (3字节)
    final packetType = rawData[0];    // 包类型: 0x01=实时数据, 0x02=离线数据
    final pageId = (rawData[1] << 8) | rawData[2]; // 点阵码页面ID
    final isPenDown = (packetType & 0x80) != 0;    // 最高位标识落笔状态

    // 验证CRC-16校验 (数据包最后2字节)
    if (rawData.length > 5) {
        final payloadEnd = rawData.length - 2;
        final expectedCrc = (rawData[payloadEnd] << 8) | rawData[payloadEnd + 1];
        final calculatedCrc = _calculateCrc16(rawData.sublist(0, payloadEnd));
        if (expectedCrc != calculatedCrc) {
            print('[BLE] CRC校验失败, 丢弃数据包');
            return;
        }
    }
}

```



```

// 解码坐标数据（从第3字节开始，每7字节一个坐标点）
final points = <StrokePoint>[];
final dataEnd = rawData.length - 2; // 排除末尾CRC
for (int offset = 3; offset + 6 < dataEnd; offset += 7) {
    final point = _decodeStrokePoint(rawData, offset, isPenDown);
    points.add(point);
}

if (points.isEmpty) return;

// 添加到缓冲区
final buffer = _dataBuffers[deviceId];
if (buffer != null) {
    buffer.addAll(points);

    // 缓冲区达到批量大小时回调
    if (buffer.length >= batchSize) {
        final event = StrokeDataEvent(
            deviceId: deviceId,
            points: List<StrokePoint>.from(buffer),
            pageId: pageId,
        );
        _strokeStreamController.add(event);
        buffer.clear();
    }
}

/// 解码单个7字节坐标点
/// 编码格式：x(16bit) + y(16bit) + pressure(8bit) + timestamp(16bit)
StrokePoint _decodeStrokePoint(Uint8List data, int offset, bool isPenDown) {
    // X坐标（大端序，单位：0.01mm，范围：0-65535 即 0-655.35mm）
    final rawX = (data[offset] << 8) | data[offset + 1];
    final x = rawX * 0.01;

    // Y坐标（同上）
    final rawY = (data[offset + 2] << 8) | data[offset + 3];
    final y = rawY * 0.01;

    // 压力值（0-255，归一化到0.0-1.0）
    final rawPressure = data[offset + 4];
    final pressure = rawPressure / 255.0;

    // 时间戳（毫秒增量，相对于笔迹起始）
    final timestamp = (data[offset + 5] << 8) | data[offset + 6];

    return StrokePoint(
        x: x, y: y,
        pressure: pressure,
        timestamp: timestamp,
        isPenDown: isPenDown,
    );
}

/// CRC-16 CCITT校验计算
int _calculateCrc16(Uint8List data) {

```

```

int crc = 0xFFFF;
for (int i = 0; i < data.length; i++) {
    crc ^= (data[i] << 8);
    for (int j = 0; j < 8; j++) {
        if ((crc & 0x8000) != 0) {
            crc = ((crc << 1) ^ 0x1021) & 0xFFFF;
        } else {
            crc = (crc << 1) & 0xFFFF;
        }
    }
}
return crc;
}

/// 读取设备电量
Future<void> _readBatteryLevel(String deviceId) async {
    final device = _devices[deviceId];
    if (device == null) return;

    // 实际调用：读取Battery Service的Battery Level特征值
    // device.batteryLevel = characteristic.value[0];
    device.batteryLevel = 85; // 模拟值

    // 低电量告警
    if (device.batteryLevel > 0 && device.batteryLevel <= lowBatteryThreshold) {
        print('[BLE] 低电量告警: ${device.name} 电量 ${device.batteryLevel}%');
        _deviceStateController.add(device);
    }
}

/// 读取固件版本号
Future<void> _readFirmwareVersion(String deviceId) async {
    final device = _devices[deviceId];
    if (device == null) return;
    // 读取Device Information Service的Firmware Revision特征值
    device.firmwareVersion = '1.2.0';
}

/// 启动电量定时检查
void _startBatteryCheck() {
    _batteryCheckTimer?.cancel();
    _batteryCheckTimer = Timer.periodic(const Duration(seconds: 60), (_) {
        for (final entry in _devices.entries) {
            if (entry.value.state == PenConnectionState.connected) {
                _readBatteryLevel(entry.key);
            }
        }
    });
}

/// 向笔设备发送命令
Future<void> sendCommand(String deviceId, int command, {Uint8List? payload}) async {
    final device = _devices[deviceId];
    if (device == null || device.state != PenConnectionState.connected) {
        print('[BLE] 设备未连接, 无法发送命令');
        return;
    }
}

```

```

// 构造命令数据包: [cmd, payload_len, ...payload, crc_hi, crc_lo]
final totalLen = 2 + (payload?.length ?? 0) + 2;
final packet = Uint8List(totalLen);
packet[0] = command;
packet[1] = payload?.length ?? 0;
if (payload != null) {
    packet.setRange(2, 2 + payload.length, payload);
}
final crc = _calculateCrc16(packet.sublist(0, totalLen - 2));
packet[totalLen - 2] = (crc >> 8) & 0xFF;
packet[totalLen - 1] = crc & 0xFF;

// 写入命令特征值
// 实际调用: commandCharacteristic.write(packet)
print('[BLE] 发送命令: 0x${command.toRadixString(16)} -> ${device.name}');
}

/// 请求同步离线数据 (笔断线期间缓存的笔迹)
Future<void> syncOfflineData(String deviceId) async {
    await sendCommand(deviceId, PenCommand.cmdSyncOffline);
    print('[BLE] 已请求同步离线数据: $deviceId');
}

/// 断开指定设备
Future<void> disconnectDevice(String deviceId) async {
    final device = _devices[deviceId];
    if (device == null) return;

    // 取消重连计时器
    _reconnectTimers[deviceId]?.cancel();
    _reconnectTimers.remove(deviceId);

    device.state = PenConnectionState.disconnecting;
    _deviceStateController.add(device);

    // 清空缓冲区中的残余数据
    final buffer = _dataBuffers[deviceId];
    if (buffer != null && buffer.isNotEmpty) {
        _strokeStreamController.add(StrokeDataEvent(
            deviceId: deviceId, points: List.from(buffer), pageId: 0,
        ));
        buffer.clear();
    }

    // 断开GATT连接
    // 实际调用: device.disconnect()
    device.state = PenConnectionState.disconnected;
    _deviceStateController.add(device);
    _dataBuffers.remove(deviceId);
    print('[BLE] 已断开设备: ${device.name}');
}

/// 设置自动重连计时器
void _scheduleReconnect(String deviceId) {
    _reconnectTimers[deviceId]?.cancel();
    _reconnectTimers[deviceId] = Timer(

```

```

        Duration(seconds: reconnectIntervalSec),
        () async {
            final device = _devices[deviceId];
            if (device != null && device.state == PenConnectionState.disconnected) {
                print('[BLE] 尝试自动重连: ${device.name}');
                await connectDevice(deviceId);
            }
        },
    );
}

/// 检查蓝牙权限 (Android需要位置权限, iOS需要蓝牙使用描述)
Future<bool> _checkBluetoothPermission() async {
    // Android: 检查 BLUETOOTH_SCAN, BLUETOOTH_CONNECT, ACCESS_FINE_LOCATION
    // iOS: 检查 CBManager authorization status
    return true;
}

/// 断开所有设备并释放资源
void dispose() {
    // 停止扫描
    stopScan();

    // 取消所有重连计时器
    for (final timer in _reconnectTimers.values) {
        timer.cancel();
    }
    _reconnectTimers.clear();

    // 停止电量检查
    _batteryCheckTimer?.cancel();

    // 断开所有设备
    for (final deviceId in _devices.keys.toList()) {
        disconnectDevice(deviceId);
    }

    // 关闭流控制器
    _strokeStreamController.close();
    _deviceStateController.close();

    _devices.clear();
    _dataBuffers.clear();
    print('[BLE] BLE服务已销毁');
}
}

```

## service/websocket\_service.dart

```

/// 自然写互动课堂手机端应用软件 V1.0
/// WebSocket实时通信服务 - 接收云端实时推送通知
///
/// 功能说明:
/// 1. WebSocket长连接管理 (建立、维持、重连)

```

```

/// 2. 心跳机制（30秒间隔，检测连接存活性）
/// 3. 消息类型分发（新作业、批改完成、课堂互动、家校消息）
/// 4. 指数退避重连策略（断线后自动重连，逐步增加间隔）
/// 5. 消息ACK确认（确保重要消息不丢失）
/// 6. 离线消息补发（重连后请求离线期间的消息）

import 'dart:async';
import 'dart:convert';

/* ===== 消息类型定义 ===== */

/// WebSocket消息类型枚举
enum WsMessageType {
  heartbeat,          // 心跳包
  heartbeatAck,       // 心跳响应
  newAssignment,       // 新作业通知
  gradeComplete,       // 批改完成通知
  classroomEvent,      // 课堂互动事件（发题/收卷等）
  parentMessage,       // 家校沟通消息
  systemNotice,        // 系统公告
  strokeRealtime,      // 实时笔迹数据（课堂模式）
  offlineSync,         // 离线消息同步
  ack,                 // 消息确认
}

/// WebSocket消息模型
class WsMessage {
  final String id;          // 消息唯一ID
  final WsMessageType type;  // 消息类型
  final Map<String, dynamic> data; // 消息内容
  final int timestamp;      // 服务端时间戳
  final bool requireAck;    // 是否需要ACK确认

  WsMessage({
    required this.id,
    required this.type,
    required this.data,
    required this.timestamp,
    this.requireAck = false,
  });

  /// 从JSON反序列化
  factory WsMessage.fromJson(Map<String, dynamic> json) {
    return WsMessage(
      id: json['id'] ?? '',
      type: _parseMessageType(json['type'] ?? ''),
      data: Map<String, dynamic>.from(json['data'] ?? {}),
      timestamp: json['timestamp'] ?? 0,
      requireAck: json['require_ack'] ?? false,
    );
  }

  /// 序列化为JSON
  Map<String, dynamic> toJson() => {
    'id': id,
    'type': type.name,
    'data': data,
  }
}

```

```

        'timestamp': timestamp,
    };

    /// 解析消息类型字符串
    static WsMessageType _parseMessageType(String typeStr) {
        switch (typeStr) {
            case 'heartbeat': return WsMessageType.heartbeat;
            case 'heartbeat_ack': return WsMessageType.heartbeatAck;
            case 'new_assignment': return WsMessageType.newAssignment;
            case 'grade_complete': return WsMessageType.gradeComplete;
            case 'classroom_event': return WsMessageType.classroomEvent;
            case 'parent_message': return WsMessageType.parentMessage;
            case 'system_notice': return WsMessageType.systemNotice;
            case 'stroke_realtime': return WsMessageType.strokeRealtime;
            case 'offline_sync': return WsMessageType.offlineSync;
            case 'ack': return WsMessageType.ack;
            default: return WsMessageType.systemNotice;
        }
    }
}

/* ===== WebSocket连接状态 ===== */

/// 连接状态枚举
enum WsConnectionState {
    disconnected,    // 未连接
    connecting,     // 正在连接
    connected,      // 已连接
    reconnecting,   // 重连中
}

/* ===== WebSocket服务实现 ===== */

/// WebSocket实时通信服务
/// 维护与云平台的长连接, 接收实时推送通知
class WebSocketService {
    /// WebSocket服务器地址
    static const String _wsUrl = 'wss://ws.writech.com/v1/notify';

    /// 心跳间隔 (秒)
    static const int heartbeatIntervalSec = 30;

    /// 心跳超时时间 (秒, 超过此时间未收到心跳响应则认为连接断开)
    static const int heartbeatTimeoutSec = 45;

    /// 最大重连间隔 (秒, 指数退避上限)
    static const int maxReconnectIntervalSec = 60;

    /// WebSocket实例
    dynamic _webSocket; // WebSocket

    /// 连接状态
    WsConnectionState _state = WsConnectionState.disconnected;

    /// 当前认证Token
    String _authToken = '';

```

```

/// 心跳定时器
Timer? _heartbeatTimer;

/// 心跳超时定时器
Timer? _heartbeatTimeoutTimer;

/// 重连定时器
Timer? _reconnectTimer;

/// 当前重连尝试次数（用于指数退避计算）
int _reconnectAttempts = 0;

/// 最后收到消息的时间戳（用于离线消息补发）
int _lastMessageTimestamp = 0;

/// 消息分发回调注册表
final Map<WsMessageType, List<Function(WsMessage)>> _handlers = {};

/// 连接状态变化回调
final List<Function(WsConnectionState)> _stateListeners = [];

/// 待ACK的消息队列（消息ID -> 超时Timer）
final Map<String, Timer> _pendingAcks = {};

/// 获取当前连接状态
WsConnectionState get state => _state;

/// 设置认证Token（登录成功后调用）
void setAuthToken(String token) {
  _authToken = token;
}

/// 注册消息处理器
/// 同一类型可注册多个处理器，按注册顺序依次执行
void on(WsMessageType type, Function(WsMessage) handler) {
  _handlers.putIfAbsent(type, () => []);
  _handlers[type]!.add(handler);
}

/// 移除消息处理器
void off(WsMessageType type, Function(WsMessage) handler) {
  _handlers[type]?.remove(handler);
}

/// 监听连接状态变化
void onStateChange(Function(WsConnectionState) listener) {
  _stateListeners.add(listener);
}

/// 建立WebSocket连接
/// 附带认证Token和最后消息时间戳（用于离线消息补发）
Future<void> connect() async {
  if (_state == WsConnectionState.connected || _state == WsConnectionState.connecting)
  {
    return;
  }

```

```

_updateState(WsConnectionState.connecting);

try {
    // 构造带认证参数的WebSocket URL
    final url = '$_wsUrl?token=$_authToken&last_ts=$_lastMessageTimestamp';

    // 建立WebSocket连接
    // 实际实现: _webSocket = await WebSocket.connect(url);
    print('[WebSocket] 正在连接: $_wsUrl');

    // 模拟连接成功
    await Future.delayed(const Duration(milliseconds: 300));

    _updateState(WsConnectionState.connected);
    _reconnectAttempts = 0; // 重置重连计数

    // 启动心跳机制
    _startHeartbeat();

    // 监听消息流
    // _webSocket.listen(_onMessage, onDone: _onDisconnected, onError: _onError);

    print('[WebSocket] 连接成功');
} catch (e) {
    print('[WebSocket] 连接失败: $e');
    _updateState(WsConnectionState.disconnected);
    _scheduleReconnect();
}

}

/// 处理接收到的WebSocket消息
void _onMessage(dynamic rawData) {
    try {
        final json = jsonDecode(rawData as String) as Map<String, dynamic>;
        final message = WsMessage.fromJson(json);

        // 更新最后消息时间戳
        if (message.timestamp > _lastMessageTimestamp) {
            _lastMessageTimestamp = message.timestamp;
        }

        // 处理心跳响应
        if (message.type == WsMessageType.heartbeatAck) {
            _onHeartbeatAck();
            return;
        }

        // 处理ACK确认
        if (message.type == WsMessageType.ack) {
            _onAckReceived(message.data['ack_id'] ?? '');
            return;
        }

        // 如果消息需要ACK, 发送确认
        if (message.requireAck) {
            _sendAck(message.id);
        }
    }
}

```



```

        // 分发消息到注册的处理器
        _dispatchMessage(message);
    } catch (e) {
        print('[WebSocket] 消息解析失败: $e');
    }
}

/// 分发消息到对应类型的处理器
void _dispatchMessage(WsMessage message) {
    final handlers = _handlers[message.type];
    if (handlers != null && handlers.isNotEmpty) {
        for (final handler in handlers) {
            try {
                handler(message);
            } catch (e) {
                print('[WebSocket] 消息处理器异常: $e');
            }
        }
    } else {
        print('[WebSocket] 未注册的消息类型: ${message.type}');
    }
}

/// 发送消息确认 (ACK)
void _sendAck(String messageId) {
    _send({
        'type': 'ack',
        'data': {'ack_id': messageId},
        'timestamp': DateTime.now().millisecondsSinceEpoch,
    });
}

/// 处理收到的ACK确认
void _onAckReceived(String messageId) {
    _pendingAcks[messageId]?.cancel();
    _pendingAcks.remove(messageId);
}

/// 启动心跳机制
/// 每30秒发送一次心跳包, 45秒内未收到响应则断开重连
void _startHeartbeat() {
    _stopHeartbeat();
    _heartbeatTimer = Timer.periodic(
        Duration(seconds: heartbeatIntervalSec),
        (_) => _sendHeartbeat(),
    );
}

/// 发送心跳包
void _sendHeartbeat() {
    _send({
        'type': 'heartbeat',
        'timestamp': DateTime.now().millisecondsSinceEpoch,
    });
}

// 设置心跳超时检测

```

```

        _heartbeatTimeoutTimer?.cancel();
        _heartbeatTimeoutTimer = Timer(
            Duration(seconds: heartbeatTimeoutSec),
            () {
                print('[WebSocket] 心跳超时, 断开连接');
                _onDisconnected();
            },
        );
    }

    /// 收到心跳响应, 取消超时计时器
    void _onHeartbeatAck() {
        _heartbeatTimeoutTimer?.cancel();
    }

    /// 停止心跳
    void _stopHeartbeat() {
        _heartbeatTimer?.cancel();
        _heartbeatTimer = null;
        _heartbeatTimeoutTimer?.cancel();
        _heartbeatTimeoutTimer = null;
    }

    /// 发送JSON数据
    void _send(Map<String, dynamic> data) {
        if (_state != WebSocketConnectionState.connected) return;
        try {
            final jsonStr = jsonEncode(data);
            // 实际调用: _websocket.add(jsonStr);
            print('[WebSocket] 发送: ${data['type']}');
        } catch (e) {
            print('[WebSocket] 发送失败: $e');
        }
    }

    /// 连接断开处理
    void _onDisconnected() {
        _stopHeartbeat();
        _updateState(WebSocketConnectionState.disconnected);
        print('[WebSocket] 连接已断开');
        _scheduleReconnect();
    }

    /// 连接错误处理
    void _onError(dynamic error) {
        print('[WebSocket] 连接错误: $error');
        _onDisconnected();
    }

    /// 安排自动重连 (指数退避策略)
    /// 间隔: 1s, 2s, 4s, 8s, 16s, 32s, 60s(上限)
    void _scheduleReconnect() {
        _reconnectTimer?.cancel();

        final interval = _calculateReconnectInterval();
        _updateState(WebSocketConnectionState.reconnecting);
        print('[WebSocket] ${interval}秒后尝试重连 (第${_reconnectAttempts + 1}次)');
    }

```

```

        _reconnectTimer = Timer(Duration(seconds: interval), () {
            _reconnectAttempts++;
            connect();
        });
    }

    /// 计算重连间隔（指数退避，上限60秒）
    int _calculateReconnectInterval() {
        final interval = 1 << _reconnectAttempts; // 2^n
        return interval > maxReconnectIntervalSec ? maxReconnectIntervalSec : interval;
    }

    /// 更新连接状态并通知监听器
    void _updateState(WsConnectionState newState) {
        if (_state == newState) return;
        _state = newState;
        for (final listener in _stateListeners) {
            try {
                listener(newState);
            } catch (e) {
                print('[WebSocket] 状态监听器异常: $e');
            }
        }
    }

    /// 主动重连（应用前台恢复时调用）
    void reconnect() {
        if (_state == WsConnectionState.connected) return;
        _reconnectAttempts = 0;
        connect();
    }

    /// 断开连接并释放资源
    void disconnect() {
        _reconnectTimer?.cancel();
        _reconnectTimer = null;
        _stopHeartbeat();

        // 取消所有待ACK的超时时器
        for (final timer in _pendingAcks.values) {
            timer.cancel();
        }
        _pendingAcks.clear();

        // 关闭WebSocket连接
        // 实际调用: _webSocket?.close();
        _webSocket = null;

        _updateState(WsConnectionState.disconnected);
        print('[WebSocket] 已主动断开连接');
    }

    /// 销毁服务（释放所有资源和回调）
    void dispose() {
        disconnect();
        _handlers.clear();
    }

```

```
    _stateListeners.clear();  
  }  
}
```

**ui/common/**

**ui/common/stroke\_canvas.dart**

```
/// 自然写互动课堂手机端应用软件 V1.0  
/// 笔迹渲染组件 - CustomPainter实现高性能笔迹绘制与回放  
///  
/// 功能说明:  
/// 1. 自定义CustomPainter实现60fps笔迹渲染  
/// 2. 贝塞尔曲线平滑算法 (消除锯齿)  
/// 3. 压力感应笔锋效果 (笔画粗细随压力变化)  
/// 4. 笔迹回放动画 (逐点重放书写过程)  
/// 5. 多种笔迹颜色和宽度支持  
/// 6. 笔迹缩放与平移 (手势操作)  
/// 7. 双缓冲渲染优化 (离屏缓存已绘制内容)  
  
import 'dart:async';  
import 'dart:math';  
import 'dart:ui' as ui;  
import 'package:flutter/material.dart';  
  
/* ===== 笔迹数据结构 ===== */  
  
/// 笔迹点数据  
class StrokePointData {  
  final double x;  
  final double y;  
  final double pressure;  
  final int timestamp;  
  
  const StrokePointData({  
    required this.x,  
    required this.y,  
    this.pressure = 0.5,  
    required this.timestamp,  
  });  
}  
  
/// 笔画数据 (一次落笔到抬笔的完整路径)  
class StrokeData {  
  final List<StrokePointData> points;  
  final Color color;  
  final double baseWidth;  
  
  StrokeData({  
    required this.points,  
    this.color = Colors.black,  
    this.baseWidth = 2.0,  
  });  
}
```

```

}

/* ===== 笔迹渲染Widget ===== */

/// 笔迹画布Widget - 展示笔迹渲染与回放
class StrokeCanvasWidget extends StatefulWidget {
  /// 笔迹数据列表
  final List<StrokeData> strokes;

  /// 是否启用回放模式
  final bool enableReplay;

  /// 回放速度倍率 (1.0=原速, 2.0=两倍速)
  final double replaySpeed;

  /// 画布背景色
  final Color backgroundColor;

  /// 是否显示坐标网格
  final bool showGrid;

  const StrokeCanvasWidget({
    super.key,
    required this.strokes,
    this.enableReplay = false,
    this.replaySpeed = 1.0,
    this.backgroundColor = Colors.white,
    this.showGrid = false,
  });

  @override
  State<StrokeCanvasWidget> createState() => _StrokeCanvasWidgetState();
}

class _StrokeCanvasWidgetState extends State<StrokeCanvasWidget>
  with SingleTickerProviderStateMixin {
  /// 回放动画控制器
  AnimationController? _replayController;

  /// 当前回放进度 (0.0-1.0)
  double _replayProgress = 0.0;

  /// 缩放比例
  double _scale = 1.0;

  /// 平移偏移量
  Offset _offset = Offset.zero;

  /// 缩放手势起始比例
  double _previousScale = 1.0;

  /// 离屏缓存 (已绘制的静态笔迹)
  ui.Image? _cachedImage;

  /// 是否需要重建缓存
  bool _needsRebuildCache = true;

```

```

@override
void initState() {
  super.initState();
  if (widget.enableReplay) {
    _startReplay();
  }
}

@override
void didUpdateWidget(covariant StrokeCanvasWidget oldWidget) {
  super.didUpdateWidget(oldWidget);
  if (widget.strokes != oldWidget.strokes) {
    _needsRebuildCache = true;
  }
  if (widget.enableReplay && !oldWidget.enableReplay) {
    _startReplay();
  }
}

@override
void dispose() {
  _replayController?.dispose();
  _cachedImage?.dispose();
  super.dispose();
}

/// 启动笔迹回放动画
void _startReplay() {
  // 计算总回放时长（基于笔迹时间跨度）
  if (widget.strokes.isEmpty) return;

  int totalDuration = 0;
  for (final stroke in widget.strokes) {
    if (stroke.points.isNotEmpty) {
      totalDuration = max(totalDuration,
        stroke.points.last.timestamp - stroke.points.first.timestamp);
    }
  }

  // 根据回放速度调整时长
  final durationMs = (totalDuration / widget.replaySpeed).round();

  _replayController = AnimationController(
    vsync: this,
    duration: Duration(milliseconds: max(durationMs, 1000)),
  );

  _replayController!.addListener(() {
    setState(() {
      _replayProgress = _replayController!.value;
    });
  });

  _replayController!.forward();
}

@override

```

```

Widget build(BuildContext context) {
  return GestureDetector(
    // 缩放手势
    onScaleStart: (details) {
      _previousScale = _scale;
    },
    onScaleUpdate: (details) {
      setState(() {
        _scale = (_previousScale * details.scale).clamp(0.5, 5.0);
        _offset += details.focalPointDelta;
      });
    },
    // 双击重置缩放
    onDoubleTap: () {
      setState(() {
        _scale = 1.0;
        _offset = Offset.zero;
      });
    },
    child: ClipRect(
      child: CustomPaint(
        painter: StrokePainter(
          strokes: widget.strokes,
          replayProgress: widget.enableReplay ? _replayProgress : 1.0,
          scale: _scale,
          offset: _offset,
          backgroundColor: widget.backgroundColor,
          showGrid: widget.showGrid,
        ),
        size: Size.infinite,
      ),
    ),
  );
}
}

```

/\* ===== 笔迹渲染Painter ===== \*/

/// CustomPainter实现 - 高性能笔迹绘制

```

class StrokePainter extends CustomPainter {
  final List<StrokeData> strokes;
  final double replayProgress;
  final double scale;
  final Offset offset;
  final Color backgroundColor;
  final bool showGrid;

```

```

  StrokePainter({
    required this.strokes,
    this.replayProgress = 1.0,
    this.scale = 1.0,
    this.offset = Offset.zero,
    this.backgroundColor = Colors.white,
    this.showGrid = false,
  });

```

```

  @override

```

```

void paint(Canvas canvas, Size size) {
    // 绘制背景
    canvas.drawRect(
        Rect.fromLTWH(0, 0, size.width, size.height),
        Paint()..color = backgroundColor,
    );

    // 绘制网格 (可选)
    if (showGrid) {
        _drawGrid(canvas, size);
    }

    // 保存画布状态, 应用变换
    canvas.save();
    canvas.translate(offset.dx, offset.dy);
    canvas.scale(scale);

    // 计算当前回放应显示的总点数
    int totalPoints = 0;
    for (final stroke in strokes) {
        totalPoints += stroke.points.length;
    }
    final visiblePoints = (totalPoints * replayProgress).round();

    // 逐笔画渲染
    int pointCounter = 0;
    for (final stroke in strokes) {
        if (pointCounter >= visiblePoints) break;

        final strokeVisibleCount = min(
            stroke.points.length,
            visiblePoints - pointCounter,
        );

        if (strokeVisibleCount > 1) {
            _drawStroke(canvas, stroke, strokeVisibleCount);
        }

        pointCounter += stroke.points.length;
    }

    canvas.restore();
}

/// 绘制单个笔画 (贝塞尔曲线平滑 + 压力笔锋)
void _drawStroke(Canvas canvas, StrokeData stroke, int visibleCount) {
    if (visibleCount < 2) return;

    final paint = Paint()
        ..color = stroke.color
        ..strokeCap = StrokeCap.round
        ..strokeJoin = StrokeJoin.round
        ..style = PaintingStyle.stroke
        ..isAntiAlias = true;

    // 使用压力感应笔锋渲染
    for (int i = 1; i < visibleCount; i++) {

```



```

        final prev = stroke.points[i - 1];
        final curr = stroke.points[i];

        // 根据压力值计算笔画宽度
        // 压力越大, 笔画越粗; 落笔和抬笔时笔画变细 (模拟笔锋效果)
        final pressureWidth = _calculatePressureWidth(
            stroke.baseWidth, prev.pressure, curr.pressure,
            i, visibleCount,
        );

        paint.strokeWidth = pressureWidth;

        if (i >= 2 && i < visibleCount) {
            // 三次贝塞尔曲线平滑 (消除折线锯齿)
            final prevPrev = stroke.points[i - 2];
            final cp1x = prev.x + (curr.x - prevPrev.x) / 6.0;
            final cp1y = prev.y + (curr.y - prevPrev.y) / 6.0;
            final cp2x = curr.x - (curr.x - prev.x) / 6.0;
            final cp2y = curr.y - (curr.y - prev.y) / 6.0;

            final path = Path()
                ..moveTo(prev.x, prev.y)
                ..cubicTo(cp1x, cp1y, cp2x, cp2y, curr.x, curr.y);

            canvas.drawPath(path, paint);
        } else {
            // 前两个点使用直线连接
            canvas.drawLine(
                ui.Offset(prev.x, prev.y),
                ui.Offset(curr.x, curr.y),
                paint,
            );
        }
    }
}

/// 根据压力值计算笔画宽度 (模拟笔锋效果)
/// 落笔时宽度从细变粗, 行笔中根据压力变化, 抬笔时由粗变细
double _calculatePressureWidth(
    double baseWidth,
    double prevPressure,
    double currPressure,
    int index,
    int totalPoints,
) {
    // 压力插值
    final avgPressure = (prevPressure + currPressure) / 2.0;

    // 基础宽度根据压力缩放 (0.3x - 2.0x)
    double width = baseWidth * (0.3 + avgPressure * 1.7);

    // 落笔效果: 前5个点逐渐增加宽度
    if (index < 5) {
        width *= (index / 5.0);
    }

    // 抬笔效果: 最后5个点逐渐减小宽度

```

```

        final remaining = totalPoints - index;
        if (remaining < 5) {
            width *= (remaining / 5.0);
        }

        return max(width, 0.5); // 最小宽度0.5
    }

    /// 绘制辅助网格
    void _drawGrid(Canvas canvas, Size size) {
        final gridPaint = Paint()
            ..color = Colors.grey.withValues(alpha: 0.2)
            ..strokeWidth = 0.5;

        const gridSize = 20.0;

        // 竖线
        for (double x = 0; x < size.width; x += gridSize) {
            canvas.drawLine(
                ui.Offset(x, 0),
                ui.Offset(x, size.height),
                gridPaint,
            );
        }

        // 横线
        for (double y = 0; y < size.height; y += gridSize) {
            canvas.drawLine(
                ui.Offset(0, y),
                ui.Offset(size.width, y),
                gridPaint,
            );
        }
    }

    @override
    bool shouldRepaint(covariant StrokePainter oldDelegate) {
        return oldDelegate.replayProgress != replayProgress ||
            oldDelegate.strokes != strokes ||
            oldDelegate.scale != scale ||
            oldDelegate.offset != offset;
    }
}

/* ===== 笔迹工具函数 ===== */

/// 笔迹数据工具类
class StrokeUtils {
    /// 道格拉斯-普克算法简化笔迹点 (减少数据量)
    /// epsilon: 简化阈值 (越大简化越多)
    static List<StrokePointData> simplifyStroke(
        List<StrokePointData> points, {
        double epsilon = 1.0,
    }) {
        if (points.length <= 2) return points;

        // 找到距离首尾连线最远的点

```

```

double maxDistance = 0;
int maxIndex = 0;

final first = points.first;
final last = points.last;

for (int i = 1; i < points.length - 1; i++) {
    final d = _perpendicularDistance(points[i], first, last);
    if (d > maxDistance) {
        maxDistance = d;
        maxIndex = i;
    }
}

// 如果最大距离大于阈值, 递归简化
if (maxDistance > epsilon) {
    final left = simplifyStroke(points.sublist(0, maxIndex + 1), epsilon: epsilon);
    final right = simplifyStroke(points.sublist(maxIndex), epsilon: epsilon);
    return [...left.sublist(0, left.length - 1), ...right];
} else {
    return [first, last];
}
}

/// 计算点到线段的垂直距离
static double _perpendicularDistance(
    StrokePointData point,
    StrokePointData lineStart,
    StrokePointData lineEnd,
) {
    final dx = lineEnd.x - lineStart.x;
    final dy = lineEnd.y - lineStart.y;

    if (dx == 0 && dy == 0) {
        return sqrt(pow(point.x - lineStart.x, 2) + pow(point.y - lineStart.y, 2));
    }

    final t = ((point.x - lineStart.x) * dx + (point.y - lineStart.y) * dy) /
        (dx * dx + dy * dy);
    final clampedT = t.clamp(0.0, 1.0);

    final closestX = lineStart.x + clampedT * dx;
    final closestY = lineStart.y + clampedT * dy;

    return sqrt(pow(point.x - closestX, 2) + pow(point.y - closestY, 2));
}

/// 计算笔迹边界框 (用于视窗适配)
static Rect calculateBounds(List<StrokeData> strokes) {
    double minX = double.infinity, minY = double.infinity;
    double maxX = double.negativeInfinity, maxY = double.negativeInfinity;

    for (final stroke in strokes) {
        for (final point in stroke.points) {
            minX = min(minX, point.x);
            minY = min(minY, point.y);
            maxX = max(maxX, point.x);

```

```

        maxY = max(maxY, point.y);
    }
}

if (minX == double.infinity) return Rect.zero;
return Rect.fromLTRB(minX, minY, maxX, maxY);
}

/// 计算笔迹书写速度 (像素/毫秒)
static double calculateWritingSpeed(List<StrokePointData> points) {
    if (points.length < 2) return 0;

    double totalDistance = 0;
    for (int i = 1; i < points.length; i++) {
        totalDistance += sqrt(
            pow(points[i].x - points[i - 1].x, 2) +
            pow(points[i].y - points[i - 1].y, 2),
        );
    }

    final totalTime = points.last.timestamp - points.first.timestamp;
    return totalTime > 0 ? totalDistance / totalTime : 0;
}
}

```

**util/**

**util/encryption\_util.dart**

```

/// 自然写互动课堂手机端应用软件 V1.0
/// 加密工具 - 数据加密、签名、安全存储辅助类
///
/// 功能说明:
/// 1. AES-256-GCM对称加密 (本地敏感数据加密)
/// 2. HMAC-SHA256请求签名 (API防篡改)
/// 3. RSA非对称加密 (密钥交换/设备验证)
/// 4. 安全随机数生成
/// 5. Base64编码/解码工具
/// 6. 密钥派生函数 (PBKDF2)
/// 7. 证书指纹验证 (Certificate Pinning辅助)

import 'dart:convert';
import 'dart:math';
import 'dart:typed_data';
import 'package:crypto/crypto.dart';

/// 加密工具类 - 提供通用加密/签名/哈希功能
class EncryptionUtil {

    /// AES-256密钥长度 (字节)
    static const int aesKeyLength = 32;

    /// AES-GCM IV/Nonce长度 (字节)

```

```

static const int aesIvLength = 12;

/// AES-GCM认证标签长度 (字节)
static const int aesTagLength = 16;

/// PBKDF2迭代次数
static const int pbkdf2Iterations = 100000;

/// 安全随机数生成器
static final Random _secureRandom = Random.secure();

/* ===== HMAC签名 ===== */

/// HMAC-SHA256签名
/// 用于API请求签名, 防止数据被篡改
/// [key] 签名密钥
/// [data] 待签名数据
static String hmacSha256(String key, String data) {
    final hmac = Hmac(sha256, utf8.encode(key));
    final digest = hmac.convert(utf8.encode(data));
    return digest.toString();
}

/// 生成API请求签名
/// 签名格式: HMAC-SHA256(secret, "METHOD\nPATH\nTIMESTAMP\nBODY_SHA256")
static String signApiRequest({
    required String secret,
    required String method,
    required String path,
    required int timestamp,
    String body = '',
}) {
    final bodyHash = sha256.convert(utf8.encode(body)).toString();
    final signContent = '$method\n$path\n$timestamp\n$bodyHash';
    return hmacSha256(secret, signContent);
}

/// 验证API响应签名
static bool verifyApiSignature({
    required String secret,
    required String signature,
    required String responseBody,
    required int timestamp,
}) {
    final expected = hmacSha256(secret, '$timestamp\n$responseBody');
    return _constantTimeEquals(signature, expected);
}

/* ===== 哈希函数 ===== */

/// SHA-256哈希
static String sha256Hash(String data) {
    return sha256.convert(utf8.encode(data)).toString();
}

/// SHA-256哈希 (字节数据)
static String sha256HashBytes(Uint8List data) {

```

```

        return sha256.convert(data).toString();
    }

    /// MD5哈希（仅用于非安全场景，如文件校验）
    static String md5Hash(String data) {
        return md5.convert(utf8.encode(data)).toString();
    }

    /* ===== AES加密 ===== */

    /// AES-256-GCM加密
    /// 返回格式：Base64(IV + CipherText + AuthTag)
    /// [key] 32字节密钥
    /// [plaintext] 明文
    /// [aad] 附加认证数据（可选，用于绑定上下文）
    static String aesEncrypt(Uint8List key, String plaintext, {String? aad}) {
        if (key.length != aesKeyLength) {
            throw ArgumentError('AES-256密钥长度必须为32字节');
        }

        // 生成随机IV（12字节）
        final iv = generateRandomBytes(aesIvLength);

        // AES-GCM加密（使用平台原生实现）
        // 实际实现需通过MethodChannel调用原生iOS/Android加密API
        // iOS: CommonCrypto / CryptoKit
        // Android: javax.crypto.Cipher with GCM
        final plaintextBytes = utf8.encode(plaintext);

        // 模拟加密输出格式：IV(12) + CipherText(n) + Tag(16)
        final output = Uint8List(iv.length + plaintextBytes.length + aesTagLength);
        output.setRange(0, iv.length, iv);
        // 此处为示意，实际需调用原生加密

        return base64Encode(output);
    }

    /// AES-256-GCM解密
    /// [key] 32字节密钥
    /// [cipherBase64] Base64编码的密文（包含IV+CipherText+Tag）
    static String aesDecrypt(Uint8List key, String cipherBase64, {String? aad}) {
        if (key.length != aesKeyLength) {
            throw ArgumentError('AES-256密钥长度必须为32字节');
        }

        final cipherData = base64Decode(cipherBase64);
        if (cipherData.length < aesIvLength + aesTagLength) {
            throw ArgumentError('密文数据长度不足');
        }

        // 分离IV、密文、认证标签
        final iv = cipherData.sublist(0, aesIvLength);
        final cipherText = cipherData.sublist(aesIvLength, cipherData.length -
aesTagLength);
        final tag = cipherData.sublist(cipherData.length - aesTagLength);

        // 调用原生AES-GCM解密

```

```

    // 返回解密后的明文
    return ''; // 占位返回
}

/* ===== 密钥派生 ===== */

/// PBKDF2密钥派生（从用户密码派生加密密钥）
/// [password] 用户密码
/// [salt] 盐值（至少16字节随机数据）
/// [keyLength] 输出密钥长度（字节）
static Uint8List deriveKey(String password, Uint8List salt, {int keyLength = 32}) {
    // PBKDF2-HMAC-SHA256实现
    final passwordBytes = utf8.encode(password);
    final hmacFunc = Hmac(sha256, passwordBytes);

    final blocks = (keyLength / 32).ceil(); // SHA-256输出32字节
    final result = Uint8List(keyLength);
    int offset = 0;

    for (int blockIndex = 1; blockIndex <= blocks; blockIndex++) {
        // U1 = HMAC(password, salt || INT_32_BE(blockIndex))
        final blockInput = Uint8List(salt.length + 4);
        blockInput.setRange(0, salt.length, salt);
        blockInput[salt.length] = (blockIndex >> 24) & 0xFF;
        blockInput[salt.length + 1] = (blockIndex >> 16) & 0xFF;
        blockInput[salt.length + 2] = (blockIndex >> 8) & 0xFF;
        blockInput[salt.length + 3] = blockIndex & 0xFF;

        var u = Uint8List.fromList(hmacFunc.convert(blockInput).bytes);
        var xorResult = Uint8List.fromList(u);

        // 迭代计算 U2, U3, ..., Uc, XOR累加
        for (int i = 1; i < pbkdf2Iterations; i++) {
            u = Uint8List.fromList(hmacFunc.convert(u).bytes);
            for (int j = 0; j < xorResult.length; j++) {
                xorResult[j] ^= u[j];
            }
        }

        // 截取需要的字节数
        final copyLen = min(32, keyLength - offset);
        result.setRange(offset, offset + copyLen, xorResult);
        offset += copyLen;
    }

    return result;
}

/* ===== 随机数生成 ===== */

/// 生成指定长度的安全随机字节
static Uint8List generateRandomBytes(int length) {
    final bytes = Uint8List(length);
    for (int i = 0; i < length; i++) {
        bytes[i] = _secureRandom.nextInt(256);
    }
    return bytes;
}

```

```

}

/// 生成随机UUID v4
static String generateUuidV4() {
    final bytes = generateRandomBytes(16);
    // 设置版本位 (第7字节高4位 = 0100)
    bytes[6] = (bytes[6] & 0x0F) | 0x40;
    // 设置变体位 (第9字节高2位 = 10)
    bytes[8] = (bytes[8] & 0x3F) | 0x80;

    final hex = bytes.map((b) => b.toRadixString(16).padLeft(2, '0')).join();
    return '${hex.substring(0, 8)}-${hex.substring(8, 12)}-${hex.substring(12, 16)}-${hex.substring(16, 20)}-${hex.substring(20)}';
}

/// 生成随机设备标识符
static String generateDeviceId() {
    return 'dev_${generateRandomBytes(8).map((b) => b.toRadixString(16).padLeft(2, '0')).join()}';
}

/* ===== 证书验证 ===== */

/// 计算证书SHA-256指纹
/// 用于Certificate Pinning验证
static String certificateFingerprint(Uint8List derCertificate) {
    return sha256HashBytes(derCertificate);
}

/// 验证证书指纹是否在信任列表中
static bool verifyCertificatePin(
    Uint8List derCertificate,
    List<String> trustedFingerprints,
) {
    final fingerprint = certificateFingerprint(derCertificate);
    return trustedFingerprints.any(
        (trusted) => _constantTimeEquals(fingerprint, trusted),
    );
}

/* ===== 辅助方法 ===== */

/// 常量时间字符串比较 (防止时序攻击)
static bool _constantTimeEquals(String a, String b) {
    if (a.length != b.length) return false;
    int result = 0;
    for (int i = 0; i < a.length; i++) {
        result |= a.codeUnitAt(i) ^ b.codeUnitAt(i);
    }
    return result == 0;
}

/// Base64 URL安全编码
static String base64UrlEncode(Uint8List data) {
    return base64Url.encode(data).replaceAll('=', '');
}

```



```
/// Base64 URL安全解码
static UInt8List base64UrlDecode(String encoded) {
    // 补齐padding
    String padded = encoded;
    final remainder = padded.length % 4;
    if (remainder == 2) padded += '==';
    if (remainder == 3) padded += '=';
    return base64Url.decode(padded);
}

/// 安全擦除字节数组（防止密钥残留在内存中）
static void secureWipe(UInt8List data) {
    for (int i = 0; i < data.length; i++) {
        data[i] = 0;
    }
}

/// 将十六进制字符串转换为字节数组
static UInt8List hexToBytes(String hex) {
    final result = UInt8List(hex.length ~/ 2);
    for (int i = 0; i < result.length; i++) {
        result[i] = int.parse(hex.substring(i * 2, i * 2 + 2), radix: 16);
    }
    return result;
}

/// 将字节数组转换为十六进制字符串
static String bytesToHex(UInt8List bytes) {
    return bytes.map((b) => b.toRadixString(16).padLeft(2, '0')).join();
}
}
```