

自然写互动课堂平板端应用软件 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

```
10-writech-app-pad/  
├─ main.dart  
├─ bloc/  
│   └─ homework_bloc.dart  
├─ eye_care/  
│   └─ eye_care_manager.dart  
├─ renderer/  
│   └─ strokePainter.dart  
├─ repository/  
│   └─ local_repository.dart  
└─ service/  
    ├── api_service.dart  
    └─ ble_service.dart
```

源程序文件清单

(根目录)

main.dart

```
/// 自然写互动课堂平板端应用软件 V1.0  
/// APP入口 - Flutter平板端应用初始化  
///  
/// 功能说明：  
/// 1. 平板端应用初始化（Pad自适应布局配置）  
/// 2. 学生端/教师端双模式切换  
/// 3. 护眼模式初始化（色温调节、使用时长监控）  
/// 4. 全局Bloc状态管理注入
```

```

/// 5. 离线模式支持（断网时可继续作答）

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

/// 应用入口
void main() async {
  WidgetsFlutterBinding.ensureInitialized();

  // 全局错误处理
  FlutterError.onError = (FlutterErrorDetails details) {
    FlutterError.presentError(details);
    debugPrint('[CrashReport] ${details.exception}');
  };

  // 设置系统UI（平板端支持横屏+竖屏）
  await SystemChrome.setPreferredOrientations([
    DeviceOrientation.portraitUp,
    DeviceOrientation.portraitDown,
    DeviceOrientation.landscapeLeft,
    DeviceOrientation.landscapeRight,
  ]);

  // 初始化全局服务
  await _initServices();

  runZonedGuarded(() {
    runApp(const WritechPadApp());
  }, (error, stack) {
    debugPrint('[CrashReport] $error\n$stack');
  });
}

/// 初始化全局服务
Future<void> _initServices() async {
  debugPrint('[App] 服务初始化开始');
  // 初始化数据库、网络、BLE、护眼模块
  debugPrint('[App] 服务初始化完成');
}

/// 平板端应用根Widget
class WritechPadApp extends StatefulWidget {
  const WritechPadApp({super.key});

  @override
  State<WritechPadApp> createState() => _WritechPadAppState();
}

class _WritechPadAppState extends State<WritechPadApp>
  with WidgetsBindingObserver {
  /// 当前用户模式（学生/教师）
  String _userMode = 'student';

  /// 护眼模式是否开启
  bool _eyeCareEnabled = false;

```

```

/// 色温滤镜值 (0.0=正常, 1.0=最暖)
double _colorTemperature = 0.0;

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

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

@override
void didChangeAppLifecycleState(AppLifecycleState state) {
  if (state == AppLifecycleState.resumed) {
    debugPrint('[App] 应用恢复前台');
  } else if (state == AppLifecycleState.paused) {
    debugPrint('[App] 应用进入后台');
  }
}

@override
Widget build(BuildContext context) {
  return MaterialApp(
    title: '自然写互动课堂',
    debugShowCheckedModeBanner: false,
    theme: ThemeData(
      useMaterial3: true,
      colorScheme: ColorScheme.fromSeed(
        seedColor: const Color(0xFF4CAF50),
        brightness: Brightness.light,
      ),
      fontFamily: 'NotoSansSC',
    ),
    // 护眼色温滤镜叠加
    builder: (context, child) {
      if (_eyeCareEnabled && _colorTemperature > 0) {
        return ColorFiltered(
          colorFilter: ColorFilter.matrix(_buildWarmMatrix(_colorTemperature)),
          child: child,
        );
      }
      return child ?? const SizedBox();
    },
    initialRoute: '/splash',
    routes: {
      '/splash': (_) => const _SplashPage(),
      '/login': (_) => const _LoginPage(),
      '/student_home': (_) => const _StudentHomePage(),
      '/teacher_home': (_) => const _TeacherHomePage(),
      '/homework': (_) => const _HomeworkPage(),
      '/practice': (_) => const _PracticePage(),
      '/error_book': (_) => const _ErrorBookPage(),
    },
  );
}

```

```

        '/settings': (_) => const _SettingsPage(),
    },
);
}

/// 构建暖色温矩阵 (护眼模式)
List<double> _buildWarmMatrix(double intensity) {
    final r = 1.0;
    final g = 1.0 - intensity * 0.1;
    final b = 1.0 - intensity * 0.3;
    return [
        r, 0, 0, 0, 0,
        0, g, 0, 0, 0,
        0, 0, b, 0, 0,
        0, 0, 0, 1, 0,
    ];
}

// 占位页面声明
class _SplashPage extends StatelessWidget {
    const _SplashPage();
    @override
    Widget build(BuildContext context) => const Scaffold(body: Center(child: Text('自然写')));
}

class _LoginPage extends StatelessWidget {
    const _LoginPage();
    @override
    Widget build(BuildContext context) => const Scaffold();
}

class _StudentHomePage extends StatelessWidget {
    const _StudentHomePage();
    @override
    Widget build(BuildContext context) => const Scaffold();
}

class _TeacherHomePage extends StatelessWidget {
    const _TeacherHomePage();
    @override
    Widget build(BuildContext context) => const Scaffold();
}

class _HomeworkPage extends StatelessWidget {
    const _HomeworkPage();
    @override
    Widget build(BuildContext context) => const Scaffold();
}

class _PracticePage extends StatelessWidget {
    const _PracticePage();
    @override
    Widget build(BuildContext context) => const Scaffold();
}

class _ErrorBookPage extends StatelessWidget {
    const _ErrorBookPage();
    @override
    Widget build(BuildContext context) => const Scaffold();
}

class _SettingsPage extends StatelessWidget {

```

```
const _SettingsPage();  
@override  
Widget build(BuildContext context) => const Scaffold();  
}
```

bloc/

bloc/homework_bloc.dart

```
// 自然写互动课堂平板端应用软件 V1.0  
// bloc/homework_bloc.dart - 作业状态管理 (Bloc模式)  
  
import 'dart:async';  
  
/// 作业状态枚举  
enum HomeworkStatus {  
  /// 待完成  
  pending,  
  
  /// 进行中 (已开始作答)  
  inProgress,  
  
  /// 已提交  
  submitted,  
  
  /// 已批改  
  graded,  
  
  /// 已过期  
  expired,  
}  
  
/// 作业数据模型  
class HomeworkItem {  
  final String id;  
  final String title;  
  final String subject;  
  final String teacherName;  
  final HomeworkStatus status;  
  final DateTime? assignedAt;  
  final DateTime? deadline;  
  final DateTime? submittedAt;  
  final int? score;  
  final int totalQuestions;  
  final int answeredQuestions;  
  final String? coverImageUrl;  
  
  HomeworkItem({  
    required this.id,  
    required this.title,  
    required this.subject,  
    required this.teacherName,  
    this.status = HomeworkStatus.pending,  
  })
```

```

        this.assignedAt,
        this.deadline,
        this.submittedAt,
        this.score,
        this.totalQuestions = 0,
        this.answeredQuestions = 0,
        this.coverImageUrl,
    });

    /// 是否已过截止时间
    bool get isOverdue =>
        deadline != null && DateTime.now().isAfter(deadline!);

    /// 作答进度百分比
    double get progress => totalQuestions > 0
        ? answeredQuestions / totalQuestions
        : 0.0;

    /// 从JSON解析
    factory HomeworkItem.fromJson(Map<String, dynamic> json) {
        return HomeworkItem(
            id: json['id'] ?? '',
            title: json['title'] ?? '',
            subject: json['subject'] ?? '',
            teacherName: json['teacher_name'] ?? '',
            status: _parseStatus(json['status']),
            assignedAt: json['assigned_at'] != null
                ? DateTime.tryParse(json['assigned_at'])
                : null,
            deadline: json['deadline'] != null
                ? DateTime.tryParse(json['deadline'])
                : null,
            submittedAt: json['submitted_at'] != null
                ? DateTime.tryParse(json['submitted_at'])
                : null,
            score: json['score'],
            totalQuestions: json['total_questions'] ?? 0,
            answeredQuestions: json['answered_questions'] ?? 0,
            coverImageUrl: json['cover_image_url'],
        );
    }

    /// 解析状态字符串
    static HomeworkStatus _parseStatus(String? status) {
        switch (status) {
            case 'pending':
                return HomeworkStatus.pending;
            case 'in_progress':
                return HomeworkStatus.inProgress;
            case 'submitted':
                return HomeworkStatus.submitted;
            case 'graded':
                return HomeworkStatus.graded;
            case 'expired':
                return HomeworkStatus.expired;
            default:
                return HomeworkStatus.pending;
        }
    }

```

```

    }
}
}

/// 作业详情中的题目数据
class HomeworkQuestion {
    final String id;
    final int index;
    final String type;
    final String content;
    final String? imageUrl;
    final List<String>? options;
    final String? correctAnswer;
    final String? studentAnswer;
    final List<Map<String, dynamic>>? studentStrokes;
    final int? questionScore;
    final int? earnedScore;
    final String? teacherComment;

    HomeworkQuestion({
        required this.id,
        required this.index,
        required this.type,
        required this.content,
        this.imageUrl,
        this.options,
        this.correctAnswer,
        this.studentAnswer,
        this.studentStrokes,
        this.questionScore,
        this.earnedScore,
        this.teacherComment,
    });

    /// 从JSON解析
    factory HomeworkQuestion.fromJson(Map<String, dynamic> json) {
        return HomeworkQuestion(
            id: json['id'] ?? '',
            index: json['index'] ?? 0,
            type: json['type'] ?? 'write',
            content: json['content'] ?? '',
            imageUrl: json['image_url'],
            options: json['options'] != null
                ? List<String>.from(json['options'])
                : null,
            correctAnswer: json['correct_answer'],
            studentAnswer: json['student_answer'],
            studentStrokes: json['student_strokes'] != null
                ? List<Map<String, dynamic>>.from(json['student_strokes'])
                : null,
            questionScore: json['question_score'],
            earnedScore: json['earned_score'],
            teacherComment: json['teacher_comment'],
        );
    }
}

```

```
// =====
//  Bloc Events (作业相关事件定义)
// =====

/// 作业事件基类
abstract class HomeworkEvent {}

/// 加载作业列表事件
class LoadHomeworkListEvent extends HomeworkEvent {
  final HomeworkStatus? filterStatus;
  final int page;
  final bool refresh;

  LoadHomeworkListEvent({
    this.filterStatus,
    this.page = 1,
    this.refresh = false,
  });
}

/// 下载作业详情事件 (用于离线作答)
class DownloadHomeworkEvent extends HomeworkEvent {
  final String homeworkId;
  DownloadHomeworkEvent(this.homeworkId);
}

/// 保存作答进度事件 (本地暂存)
class SaveAnswerProgressEvent extends HomeworkEvent {
  final String homeworkId;
  final String questionId;
  final String? textAnswer;
  final List<Map<String, dynamic>>? strokeData;

  SaveAnswerProgressEvent({
    required this.homeworkId,
    required this.questionId,
    this.textAnswer,
    this.strokeData,
  });
}

/// 提交作业事件
class SubmitHomeworkEvent extends HomeworkEvent {
  final String homeworkId;
  SubmitHomeworkEvent(this.homeworkId);
}

/// 查看批改结果事件
class ViewGradeResultEvent extends HomeworkEvent {
  final String homeworkId;
  ViewGradeResultEvent(this.homeworkId);
}

// =====
//  Bloc States (作业相关状态定义)
// =====
```



```

/// 作业状态基类
abstract class HomeworkState {}

/// 初始状态
class HomeworkInitialState extends HomeworkState {}

/// 加载中状态
class HomeworkLoadingState extends HomeworkState {
    final String? message;
    HomeworkLoadingState({this.message});
}

/// 作业列表加载成功状态
class HomeworkListLoadedState extends HomeworkState {
    final List<HomeworkItem> homeworks;
    final bool hasMore;
    final int currentPage;
    final HomeworkStatus? currentFilter;

    /// 各状态的作业计数统计
    final Map<HomeworkStatus, int> statusCounts;

    HomeworkListLoadedState({
        required this.homeworks,
        this.hasMore = false,
        this.currentPage = 1,
        this.currentFilter,
        this.statusCounts = const {},
    });
}

/// 作业详情加载成功状态
class HomeworkDetailLoadedState extends HomeworkState {
    final HomeworkItem homework;
    final List<HomeworkQuestion> questions;
    final bool isOfflineAvailable;

    HomeworkDetailLoadedState({
        required this.homework,
        required this.questions,
        this.isOfflineAvailable = false,
    });
}

/// 作答进度保存成功状态
class AnswerSavedState extends HomeworkState {
    final String homeworkId;
    final String questionId;
    final int answeredCount;
    final int totalCount;

    AnswerSavedState({
        required this.homeworkId,
        required this.questionId,
        required this.answeredCount,
        required this.totalCount,
    });
}

```

```

}

/// 作业提交成功状态
class HomeworkSubmittedState extends HomeworkState {
  final String homeworkId;
  final DateTime submittedAt;

  HomeworkSubmittedState({
    required this.homeworkId,
    required this.submittedAt,
  });
}

/// 批改结果状态
class GradeResultState extends HomeworkState {
  final HomeworkItem homework;
  final List<HomeworkQuestion> questions;
  final int totalScore;
  final int earnedScore;
  final String? overallComment;

  GradeResultState({
    required this.homework,
    required this.questions,
    required this.totalScore,
    required this.earnedScore,
    this.overallComment,
  });
}

/// 错误状态
class HomeworkErrorState extends HomeworkState {
  final String message;
  final String? actionType;

  HomeworkErrorState({
    required this.message,
    this.actionType,
  });
}

// =====
// HomeworkBloc 实现
// =====

/// 作业状态管理Bloc
/// 管理作业列表加载、下载、作答、提交、查看批改结果等完整流程
class HomeworkBloc {
  /// 当前状态
  HomeworkState _state = HomeworkInitialState();

  /// 状态流控制器
  final StreamController<HomeworkState> _stateController =
    StreamController<HomeworkState>.broadcast();

  /// 本地缓存的作业列表
  List<HomeworkItem> _cachedHomeworks = [];

```

```

/// 本地缓存的作答进度 {homeworkId: {questionId: answerData}}
final Map<String, Map<String, dynamic>> _answerCache = {};

/// 获取当前状态
HomeworkState get state => _state;

/// 状态流
Stream<HomeworkState> get stateStream => _stateController.stream;

/// 发射新状态
void _emit(HomeworkState newState) {
  _state = newState;
  _stateController.add(newState);
}

/// 处理事件分发
void add(HomeworkEvent event) {
  if (event is LoadHomeworkListEvent) {
    _handleLoadList(event);
  } else if (event is DownloadHomeworkEvent) {
    _handleDownload(event);
  } else if (event is SaveAnswerProgressEvent) {
    _handleSaveAnswer(event);
  } else if (event is SubmitHomeworkEvent) {
    _handleSubmit(event);
  } else if (event is ViewGradeResultEvent) {
    _handleViewGrade(event);
  }
}

/// 处理加载作业列表
Future<void> _handleLoadList(LoadHomeworkListEvent event) async {
  try {
    _emit(HomeworkLoadingState(message: '正在加载作业列表...'));

    // 调用API获取作业列表
    // final response = await PadApiService.instance.getHomeworkList(
    //   page: event.page,
    //   status: event.filterStatus?.name,
    // );

    // 模拟数据处理逻辑
    if (event.refresh) {
      _cachedHomeworks.clear();
    }

    // 统计各状态作业数量
    final statusCounts = <HomeworkStatus, int>{};
    for (final hw in _cachedHomeworks) {
      statusCounts[hw.status] = (statusCounts[hw.status] ?? 0) + 1;
    }

    // 根据筛选条件过滤
    List<HomeworkItem> filtered = _cachedHomeworks;
    if (event.filterStatus != null) {
      filtered = _cachedHomeworks

```

```

        .where((hw) => hw.status == event.filterStatus)
        .toList();
    }

    _emit(HomeworkListLoadedState(
        homeworks: filtered,
        hasMore: false,
        currentPage: event.page,
        currentFilter: event.filterStatus,
        statusCounts: statusCounts,
    ));
} catch (e) {
    _emit(HomeworkErrorState(
        message: '加载作业列表失败: $e',
        actionType: 'load_list',
    ));
}
}

/// 处理下载作业详情（支持离线作答）
Future<void> _handleDownload(DownloadHomeworkEvent event) async {
    try {
        _emit(HomeworkLoadingState(message: '正在下载作业内容...'));

        // 调用API下载作业详情
        // final response = await PadApiService.instance.downloadHomework(
        //     event.homeworkId,
        // );

        // 将作业内容缓存到本地SQLite（支持离线作答）
        // await LocalRepository.instance.cacheHomework(...)

        // _emit(HomeworkDetailLoadedState(...));
    } catch (e) {
        _emit(HomeworkErrorState(
            message: '下载作业失败: $e',
            actionType: 'download',
        ));
    }
}

/// 处理保存作答进度（本地暂存，支持断点续答）
Future<void> _handleSaveAnswer(SaveAnswerProgressEvent event) async {
    try {
        // 更新内存缓存
        _answerCache.putIfAbsent(event.homeworkId, () => {});
        _answerCache[event.homeworkId]![event.questionId] = {
            'text_answer': event.textAnswer,
            'stroke_data': event.strokeData,
            'saved_at': DateTime.now().toIso8601String(),
        };

        // 持久化到本地数据库
        // await LocalRepository.instance.saveAnswerProgress(...)

        // 计算已作答题目数
        final answeredCount = _answerCache[event.homeworkId]?.length ?? 0;
    }
}

```

```

        _emit(AnswerSavedState(
            homeworkId: event.homeworkId,
            questionId: event.questionId,
            answeredCount: answeredCount,
            totalCount: 0, // 从缓存的作业详情中获取
        ));
    } catch (e) {
        _emit(HomeworkErrorState(
            message: '保存作答进度失败: $e',
            actionType: 'save_answer',
        ));
    }
}

/// 处理提交作业
Future<void> _handleSubmit(SubmitHomeworkEvent event) async {
    try {
        _emit(HomeworkLoadingState(message: '正在提交作业...'));

        // 收集所有作答数据
        final answers = _answerCache[event.homeworkId] ?? {};

        // 构建提交数据 (含笔迹页面数据)
        final strokePages = answers.entries.map((entry) {
            return {
                'question_id': entry.key,
                'answer': entry.value,
            };
        }).toList();

        // 调用API提交
        // final response = await PadApiService.instance.submitHomework(
        //     homeworkId: event.homeworkId,
        //     strokePages: strokePages,
        // );

        // 提交成功后清除本地缓存
        _answerCache.remove(event.homeworkId);

        _emit(HomeworkSubmittedState(
            homeworkId: event.homeworkId,
            submittedAt: DateTime.now(),
        ));
    } catch (e) {
        _emit(HomeworkErrorState(
            message: '提交作业失败: $e',
            actionType: 'submit',
        ));
    }
}

/// 处理查看批改结果
Future<void> _handleViewGrade(ViewGradeResultEvent event) async {
    try {
        _emit(HomeworkLoadingState(message: '正在加载批改结果...'));
    }
}

```

```

        // 调用API获取批改结果
        // final response = await PadApiService.instance.getHomeworkResult(
        //     event.homeworkId,
        // );

        // _emit(GradeResultState(...));
    } catch (e) {
        _emit(HomeworkErrorState(
            message: '加载批改结果失败: $e',
            actionType: 'view_grade',
        ));
    }
}

/// 释放资源
void dispose() {
    _stateController.close();
    _cachedHomeworks.clear();
    _answerCache.clear();
}
}

```

eye_care/

eye_care/eye_care_manager.dart

```

/// 自然写互动课堂平板端应用软件 V1.0
/// 护眼管理器 - 色温调节、使用时长监控、距离检测
///
/// 功能说明：
/// 1. 色温调节（暖色滤镜，减少蓝光对眼睛的刺激）
/// 2. 使用时长监控（按应用/科目统计，超时提醒休息）
/// 3. 距离检测（前置摄像头检测用眼距离，过近时提醒）
/// 4. 定时提醒（每30分钟提醒休息，远眺放松）
/// 5. 家长远程管控（接收家长设置的时段/时长限制）
/// 6. 护眼数据统计（每日使用时长报告）

import 'dart:async';

/// 护眼模式配置
class EyeCareConfig {
    /// 是否启用护眼模式
    bool enabled;

    /// 色温强度（0.0=关闭，1.0=最暖）
    double colorTemperature;

    /// 连续使用提醒间隔（分钟）
    int reminderIntervalMinutes;

    /// 每日使用时长上限（分钟，0=不限制）
    int dailyLimitMinutes;
}

```

```

    /// 允许使用的时段（开始小时，结束小时）
    int allowedStartHour;
    int allowedEndHour;

    /// 是否启用距离检测
    bool distanceDetectionEnabled;

    /// 安全用眼距离（厘米）
    int safeDistanceCm;

    /// 夜间模式自动开启时间（小时）
    int nightModeStartHour;
    int nightModeEndHour;

    EyeCareConfig({
        this.enabled = true,
        this.colorTemperature = 0.3,
        this.reminderIntervalMinutes = 30,
        this.dailyLimitMinutes = 120,
        this.allowedStartHour = 7,
        this.allowedEndHour = 21,
        this.distanceDetectionEnabled = false,
        this.safeDistanceCm = 30,
        this.nightModeStartHour = 20,
        this.nightModeEndHour = 7,
    });

    Map<String, dynamic> toJson() => {
        'enabled': enabled,
        'color_temperature': colorTemperature,
        'reminder_interval': reminderIntervalMinutes,
        'daily_limit': dailyLimitMinutes,
        'allowed_start': allowedStartHour,
        'allowed_end': allowedEndHour,
        'distance_enabled': distanceDetectionEnabled,
        'safe_distance': safeDistanceCm,
        'night_start': nightModeStartHour,
        'night_end': nightModeEndHour,
    };

    factory EyeCareConfig.fromJson(Map<String, dynamic> json) {
        return EyeCareConfig(
            enabled: json['enabled'] ?? true,
            colorTemperature: (json['color_temperature'] ?? 0.3).toDouble(),
            reminderIntervalMinutes: json['reminder_interval'] ?? 30,
            dailyLimitMinutes: json['daily_limit'] ?? 120,
            allowedStartHour: json['allowed_start'] ?? 7,
            allowedEndHour: json['allowed_end'] ?? 21,
            distanceDetectionEnabled: json['distance_enabled'] ?? false,
            safeDistanceCm: json['safe_distance'] ?? 30,
            nightModeStartHour: json['night_start'] ?? 20,
            nightModeEndHour: json['night_end'] ?? 7,
        );
    }
}

/// 使用时长记录

```

```

class UsageRecord {
    final String date;          // 日期 (yyyy-MM-dd)
    final String category;      // 分类 (homework/practice/reading)
    final int durationMinutes;  // 使用时长 (分钟)
    final int sessionCount;     // 使用次数

    UsageRecord({
        required this.date,
        required this.category,
        required this.durationMinutes,
        required this.sessionCount,
    });

    Map<String, dynamic> toJson() => {
        'date': date, 'category': category,
        'duration': durationMinutes, 'sessions': sessionCount,
    };
}

/// 护眼事件类型
enum EyeCareEvent {
    restReminder,           // 休息提醒
    dailyLimitReached,      // 每日时长上限
    outsideAllowedTime,     // 超出允许使用时段
    tooCloseWarning,        // 用眼距离过近
    nightModeOn,            // 夜间模式开启
    nightModeOff,           // 夜间模式关闭
}

/// 护眼事件回调
typedef EyeCareEventCallback = void Function(EyeCareEvent event, Map<String, dynamic> data);

/// 护眼管理器
class EyeCareManager {
    /// 护眼配置
    EyeCareConfig _config = EyeCareConfig();

    /// 事件回调列表
    final List<EyeCareEventCallback> _callbacks = [];

    /// 当前会话开始时间
    DateTime? _sessionStartTime;

    /// 今日累计使用时长 (秒)
    int _todayUsageSeconds = 0;

    /// 当前连续使用时长 (秒)
    int _continuousUsageSeconds = 0;

    /// 今日使用记录
    final Map<String, int> _categoryUsage = {};

    /// 计时器 (每秒更新使用时长)
    Timer? _usageTimer;

    /// 距离检测计时器

```



```
Timer? _distanceTimer;

/// 夜间模式检查计时器
Timer? _nightModeTimer;

/// 当前是否在夜间模式
bool _isNightMode = false;

/// 当前色温值（供外部读取）
double get currentColorTemperature {
    if (!_config.enabled) return 0.0;
    if (_isNightMode) return _config.colorTemperature * 1.5; // 夜间加强
    return _config.colorTemperature;
}

/// 今日总使用时长（分钟）
int get todayUsageMinutes => _todayUsageSeconds ~/ 60;

/// 剩余可用时长（分钟，-1表示不限制）
int get remainingMinutes {
    if (_config.dailyLimitMinutes <= 0) return -1;
    return _config.dailyLimitMinutes - todayUsageMinutes;
}

/// 注册事件回调
void addCallback(EyeCareEventCallback callback) {
    _callbacks.add(callback);
}

/// 移除事件回调
void removeCallback(EyeCareEventCallback callback) {
    _callbacks.remove(callback);
}

/// 更新配置（家长远程设置后调用）
void updateConfig(EyeCareConfig newConfig) {
    _config = newConfig;
    if (_config.enabled) {
        _startMonitoring();
    } else {
        _stopMonitoring();
    }
}

/// 开始使用（进入学习功能时调用）
void startSession({String category = 'default'}) {
    _sessionStartTime = DateTime.now();
    _continuousUsageSeconds = 0;

    // 检查是否在允许时段内
    final now = DateTime.now();
    if (_config.enabled && !_isWithinAllowedTime(now)) {
        _notifyEvent(EyeCareEvent.outsideAllowedTime, {
            'allowed_start': _config.allowedStartHour,
            'allowed_end': _config.allowedEndHour,
        });
    }
}
```

```

// 启动使用时长计时器
_usageTimer?.cancel();
_usageTimer = Timer.periodic(const Duration(seconds: 1), (_) {
    _todayUsageSeconds++;
    _continuousUsageSeconds++;

    // 检查连续使用时长提醒
    if (_config.reminderIntervalMinutes > 0 &&
        _continuousUsageSeconds > 0 &&
        _continuousUsageSeconds % (_config.reminderIntervalMinutes * 60) == 0) {
        _notifyEvent(EyeCareEvent.restReminder, {
            'continuous_minutes': _continuousUsageSeconds ~/ 60,
            'total_minutes': todayUsageMinutes,
        });
    }

    // 检查每日使用上限
    if (_config.dailyLimitMinutes > 0 &&
        todayUsageMinutes >= _config.dailyLimitMinutes) {
        _notifyEvent(EyeCareEvent.dailyLimitReached, {
            'limit_minutes': _config.dailyLimitMinutes,
            'used_minutes': todayUsageMinutes,
        });
    }
});

// 启动距离检测
if (_config.distanceDetectionEnabled) {
    _startDistanceDetection();
}

// 启动夜间模式检查
_startNightModeCheck();
}

/// 结束使用（退出学习功能时调用）
void endSession({String category = 'default'}) {
    _usageTimer?.cancel();
    _usageTimer = null;

    if (_sessionStartTime != null) {
        final duration = DateTime.now().difference(_sessionStartTime!).inMinutes;
        _categoryUsage[category] = (_categoryUsage[category] ?? 0) + duration;
    }

    _sessionStartTime = null;
    _continuousUsageSeconds = 0;

    _distanceTimer?.cancel();
    _distanceTimer = null;
}

/// 用户休息后重置连续使用计时
void acknowledgeRest() {
    _continuousUsageSeconds = 0;
}

```

```

/// 检查当前时间是否在允许使用时段内
bool _isWithinAllowedTime(DateTime time) {
    final hour = time.hour;
    if (_config.allowedStartHour <= _config.allowedEndHour) {
        return hour >= _config.allowedStartHour && hour < _config.allowedEndHour;
    } else {
        // 跨午夜的情况
        return hour >= _config.allowedStartHour || hour < _config.allowedEndHour;
    }
}

/// 启动监控
void _startMonitoring() {
    _startNightModeCheck();
}

/// 停止监控
void _stopMonitoring() {
    _usageTimer?.cancel();
    _distanceTimer?.cancel();
    _nightModeTimer?.cancel();
}

/// 启动距离检测（通过前置摄像头估算用眼距离）
void _startDistanceDetection() {
    _distanceTimer?.cancel();
    _distanceTimer = Timer.periodic(const Duration(seconds: 10), (_) {
        // 调用前置摄像头进行人脸检测
        // 通过人脸框大小估算距离（人脸越大=距离越近）
        _checkEyeDistance();
    });
}

/// 检查用眼距离（基于前置摄像头人脸检测）
void _checkEyeDistance() {
    // 实际实现：
    // 1. 使用CameraController获取前置摄像头预览帧
    // 2. 使用MLKit/TFLite进行人脸检测
    // 3. 根据人脸框宽度估算距离：distance = (focal_length * real_face_width) /
    face_width_in_pixels
    // 4. 本地处理，不上传图像数据（隐私保护）

    // 模拟距离检测结果
    final estimatedDistanceCm = 35; // 实际从摄像头计算

    if (estimatedDistanceCm < _config.safeDistanceCm) {
        _notifyEvent(EyeCareEvent.tooCloseWarning, {
            'current_distance': estimatedDistanceCm,
            'safe_distance': _config.safeDistanceCm,
        });
    }
}

/// 启动夜间模式检查
void _startNightModeCheck() {
    _nightModeTimer?.cancel();
}

```

```

        _nightModeTimer = Timer.periodic(const Duration(minutes: 1), (_) {
            final hour = DateTime.now().hour;
            final shouldBeNightMode = _isNightTimeHour(hour);

            if (shouldBeNightMode && !_isNightMode) {
                _isNightMode = true;
                _notifyEvent(EyeCareEvent.nightModeOn, {});
            } else if (!shouldBeNightMode && _isNightMode) {
                _isNightMode = false;
                _notifyEvent(EyeCareEvent.nightModeOff, {});
            }
        });

        // 立即检查一次
        final hour = DateTime.now().hour;
        _isNightMode = _isNightTimeHour(hour);
    }

    /// 判断是否为夜间时段
    bool _isNightTimeHour(int hour) {
        if (_config.nightModeStartHour <= _config.nightModeEndHour) {
            return hour >= _config.nightModeStartHour && hour < _config.nightModeEndHour;
        } else {
            return hour >= _config.nightModeStartHour || hour < _config.nightModeEndHour;
        }
    }

    /// 获取今日使用统计
    List<UsageRecord> getTodayUsageRecords() {
        final today = DateTime.now().toString().substring(0, 10);
        return _categoryUsage.entries.map((e) => UsageRecord(
            date: today,
            category: e.key,
            durationMinutes: e.value,
            sessionCount: 1,
        )).toList();
    }

    /// 通知事件到所有回调
    void _notifyEvent(EyeCareEvent event, Map<String, dynamic> data) {
        for (final callback in _callbacks) {
            try {
                callback(event, data);
            } catch (e) {
                // 忽略回调异常
            }
        }
    }

    /// 释放资源
    void dispose() {
        _usageTimer?.cancel();
        _distanceTimer?.cancel();
        _nightModeTimer?.cancel();
        _callbacks.clear();
    }

```

```
}  
}
```

renderer/

renderer/strokePainter.dart

```
/// 自然写互动课堂平板端应用软件 V1.0  
/// Skia笔迹渲染器 - CustomPainter实现触屏直写与点阵笔笔迹渲染  
///  
/// 功能说明:  
/// 1. CustomPainter高性能笔迹绘制 (Skia引擎)  
/// 2. 触屏直写支持 (手指/触控笔Pointer事件处理)  
/// 3. 点阵笔BLE数据渲染 (从BLE服务接收坐标数据)  
/// 4. 压力感应笔锋效果 (触控笔ActiveStylus压力数据)  
/// 5. 贝塞尔曲线平滑算法  
/// 6. 字帖练习辅助线 (田字格/米字格/四线三格)  
/// 7. 撤销/重做操作栈  
/// 8. 笔迹导出 (SVG/PNG格式)  
  
import 'dart:math';  
import 'dart:ui' as ui;  
import 'package:flutter/material.dart';  
import 'package:flutter/gestures.dart';  
  
/* ===== 数据模型 ===== */  
  
/// 笔迹点  
class PadStrokePoint {  
  final double x;  
  final double y;  
  final double pressure;  
  final int timestamp;  
  
  const PadStrokePoint({  
    required this.x,  
    required this.y,  
    this.pressure = 0.5,  
    required this.timestamp,  
  });  
  
  Map<String, dynamic> toJson() => {  
    'x': x, 'y': y, 'pressure': pressure, 'timestamp': timestamp,  
  };  
}  
  
/// 笔画  
class PadStroke {  
  final List<PadStrokePoint> points;  
  final Color color;  
  final double baseWidth;  
  final String source; // 'touch'=触屏, 'ble'=点阵笔
```

```

    PadStroke({
        List<PadStrokePoint>? points,
        this.color = Colors.black,
        this.baseWidth = 2.5,
        this.source = 'touch',
    }) : points = points ?? [];

    void addPoint(PadStrokePoint point) => points.add(point);
}

/// 辅助线类型
enum GuideLineType {
    none,          // 无辅助线
    tianZiGe,      // 田字格
    miZiGe,        // 米字格
    siXianSanGe,   // 四线三格 (英文/拼音)
}

/// 撤销/重做操作
sealed class CanvasAction {}
class AddStrokeAction extends CanvasAction {
    final PadStroke stroke;
    AddStrokeAction(this.stroke);
}
class ClearAction extends CanvasAction {
    final List<PadStroke> clearedStrokes;
    ClearAction(this.clearedStrokes);
}

/* ===== 笔迹画布Widget ===== */

/// 平板端笔迹渲染画布
/// 支持触屏直写和BLE点阵笔两种输入方式
class PadStrokeCanvas extends StatefulWidget {
    /// 初始笔画数据 (如加载已有作业笔迹)
    final List<PadStroke>? initialStrokes;

    /// 辅助线类型
    final GuideLineType guideLineType;

    /// 是否只读模式 (查看已提交的作业)
    final bool readOnly;

    /// 笔迹颜色
    final Color strokeColor;

    /// 笔画宽度
    final double strokeWidth;

    /// 笔迹变化回调
    final Function(List<PadStroke>)? onStrokesChanged;

    const PadStrokeCanvas({
        super.key,
        this.initialStrokes,
        this.guideLineType = GuideLineType.none,
        this.readOnly = false,
    });
}

```

```

        this.strokeColor = Colors.black,
        this.strokeWidth = 2.5,
        this.onStrokesChanged,
    });

    @override
    State<PadStrokeCanvas> createState() => _PadStrokeCanvasState();
}

class _PadStrokeCanvasState extends State<PadStrokeCanvas> {
    /// 已完成的笔画列表
    final List<PadStroke> _strokes = [];

    /// 当前正在绘制的笔画
    PadStroke? _currentStroke;

    /// 撤销栈
    final List<CanvasAction> _undoStack = [];

    /// 重做栈
    final List<CanvasAction> _redoStack = [];

    /// 最大撤销步数
    static const int maxUndoSteps = 50;

    @override
    void initState() {
        super.initState();
        if (widget.initialStrokes != null) {
            _strokes.addAll(widget.initialStrokes!);
        }
    }

    /// 撤销最后一个操作
    void undo() {
        if (_undoStack.isEmpty) return;
        final action = _undoStack.removeLast();
        if (action is AddStrokeAction) {
            _strokes.remove(action.stroke);
            _redoStack.add(action);
        } else if (action is ClearAction) {
            _strokes.addAll(action.clearedStrokes);
            _redoStack.add(action);
        }
        setState(() {});
        widget.onStrokesChanged?.call(_strokes);
    }

    /// 重做上一个撤销的操作
    void redo() {
        if (_redoStack.isEmpty) return;
        final action = _redoStack.removeLast();
        if (action is AddStrokeAction) {
            _strokes.add(action.stroke);
            _undoStack.add(action);
        } else if (action is ClearAction) {
            _strokes.clear();

```

```

        _undoStack.add(action);
    }
    setState(() {});
    widget.onStrokesChanged?.call(_strokes);
}

/// 清除所有笔迹
void clearAll() {
    if (_strokes.isEmpty) return;
    final cleared = List<PadStroke>.from(_strokes);
    _undoStack.add(ClearAction(cleared));
    _strokes.clear();
    _redoStack.clear();
    setState(() {});
    widget.onStrokesChanged?.call(_strokes);
}

/// 从BLE点阵笔添加笔画（外部调用）
void addBleStroke(PadStroke stroke) {
    _strokes.add(stroke);
    _undoStack.add(AddStrokeAction(stroke));
    _redoStack.clear();
    setState(() {});
    widget.onStrokesChanged?.call(_strokes);
}

/// 获取所有笔画数据（用于提交）
List<PadStroke> getStrokes() => List.unmodifiable(_strokes);

@override
Widget build(BuildContext context) {
    return Listener(
        // 使用Listener而非GestureDetector, 以获取精确的Pointer事件
        onPointerDown: widget.readOnly ? null : _onPointerDown,
        onPointerMove: widget.readOnly ? null : _onPointerMove,
        onPointerUp: widget.readOnly ? null : _onPointerUp,
        child: ClipRect(
            child: CustomPaint(
                painter: _PadStrokePainter(
                    strokes: _strokes,
                    currentStroke: _currentStroke,
                    guidelineType: widget.guidelineType,
                ),
                size: Size.infinite,
            ),
        ),
    );
}

/// 触屏落笔
void _onPointerDown(PointerDownEvent event) {
    final pressure = event.pressure > 0 ? event.pressure : 0.5;
    _currentStroke = PadStroke(
        color: widget.strokeColor,
        baseWidth: widget.strokeWidth,
        source: event.kind == PointerDeviceKind.stylus ? 'stylus' : 'touch',
    );
}

```



```

        _currentStroke!.addPoint(PadStrokePoint(
            x: event.localPosition.dx,
            y: event.localPosition.dy,
            pressure: pressure,
            timestamp: DateTime.now().millisecondsSinceEpoch,
        ));
        setState(() {});
    }

    /// 触屏移动
    void _onPointerMove(PointerMoveEvent event) {
        if (_currentStroke == null) return;
        final pressure = event.pressure > 0 ? event.pressure : 0.5;
        _currentStroke!.addPoint(PadStrokePoint(
            x: event.localPosition.dx,
            y: event.localPosition.dy,
            pressure: pressure,
            timestamp: DateTime.now().millisecondsSinceEpoch,
        ));
        setState(() {});
    }

    /// 触屏抬笔
    void _onPointerUp(PointerUpEvent event) {
        if (_currentStroke == null) return;
        if (_currentStroke!.points.length >= 2) {
            _strokes.add(_currentStroke!);
            _undoStack.add(AddStrokeAction(_currentStroke!));
            _redoStack.clear();
            // 限制撤销栈大小
            if (_undoStack.length > maxUndoSteps) {
                _undoStack.removeAt(0);
            }
            widget.onStrokesChanged?.call(_strokes);
        }
        _currentStroke = null;
        setState(() {});
    }
}

/* ===== Painter实现 ===== */

/// 笔迹绘制Painter
class _PadStrokePainter extends CustomPainter {
    final List<PadStroke> strokes;
    final PadStroke? currentStroke;
    final GuideLineType guideLineType;

    _PadStrokePainter({
        required this.strokes,
        this.currentStroke,
        this.guideLineType = GuideLineType.none,
    });

    @override
    void paint(Canvas canvas, Size size) {
        // 绘制背景

```

```

canvas.drawRect(
    Rect.fromLTWH(0, 0, size.width, size.height),
    Paint()..color = Colors.white,
);

// 绘制辅助线
if (guideLineType != GuideLineType.none) {
    _drawGuideLines(canvas, size);
}

// 绘制已完成的笔画
for (final stroke in strokes) {
    _drawStroke(canvas, stroke);
}

// 绘制当前活跃笔画
if (currentStroke != null) {
    _drawStroke(canvas, currentStroke!);
}
}

/// 绘制辅助线
void _drawGuideLines(Canvas canvas, Size size) {
    final paint = Paint()
        ..style = PaintingStyle.stroke
        ..strokeWidth = 0.5;

    switch (guideLineType) {
        case GuideLineType.tianZiGe:
            _drawTianZiGe(canvas, size, paint);
            break;
        case GuideLineType.miZiGe:
            _drawMiZiGe(canvas, size, paint);
            break;
        case GuideLineType.siXianSanGe:
            _drawSiXianSanGe(canvas, size, paint);
            break;
        default:
            break;
    }
}

/// 绘制田字格
void _drawTianZiGe(Canvas canvas, Size size, Paint paint) {
    const cellSize = 80.0;
    paint.color = Colors.red.withValues(alpha: 0.3);

    // 外框 (实线)
    for (double x = 0; x <= size.width; x += cellSize) {
        canvas.drawLine(Offset(x, 0), Offset(x, size.height), paint);
    }
    for (double y = 0; y <= size.height; y += cellSize) {
        canvas.drawLine(Offset(0, y), Offset(size.width, y), paint);
    }

    // 中心十字线 (虚线效果用半透明)
    paint.color = Colors.red.withValues(alpha: 0.15);

```

```

    final halfCell = cellSize / 2;
    for (double x = halfCell; x < size.width; x += cellSize) {
        canvas.drawLine(Offset(x, 0), Offset(x, size.height), paint);
    }
    for (double y = halfCell; y < size.height; y += cellSize) {
        canvas.drawLine(Offset(0, y), Offset(size.width, y), paint);
    }
}

/// 绘制米字格
void _drawMiZiGe(Canvas canvas, Size size, Paint paint) {
    const cellSize = 80.0;
    paint.color = Colors.red.withValues(alpha: 0.3);

    for (double x = 0; x <= size.width; x += cellSize) {
        canvas.drawLine(Offset(x, 0), Offset(x, size.height), paint);
    }
    for (double y = 0; y <= size.height; y += cellSize) {
        canvas.drawLine(Offset(0, y), Offset(size.width, y), paint);
    }

    // 对角线 + 十字线
    paint.color = Colors.red.withValues(alpha: 0.15);
    for (double x = 0; x < size.width; x += cellSize) {
        for (double y = 0; y < size.height; y += cellSize) {
            // 对角线
            canvas.drawLine(Offset(x, y), Offset(x + cellSize, y + cellSize), paint);
            canvas.drawLine(Offset(x + cellSize, y), Offset(x, y + cellSize), paint);
            // 十字线
            canvas.drawLine(Offset(x + cellSize / 2, y), Offset(x + cellSize / 2, y +
cellSize), paint);
            canvas.drawLine(Offset(x, y + cellSize / 2), Offset(x + cellSize, y + cellSize /
2), paint);
        }
    }
}

/// 绘制四线三格（拼音/英文）
void _drawSiXianSanGe(Canvas canvas, Size size, Paint paint) {
    const lineSpacing = 15.0;
    const groupHeight = lineSpacing * 3;
    const groupGap = 20.0;

    paint.color = Colors.green.withValues(alpha: 0.3);

    double y = 20;
    while (y < size.height - groupHeight) {
        // 四条横线
        for (int i = 0; i < 4; i++) {
            final lineY = y + i * lineSpacing;
            // 第二条线（中线）用虚线表示
            if (i == 1 || i == 2) {
                paint.color = Colors.green.withValues(alpha: 0.15);
            } else {
                paint.color = Colors.green.withValues(alpha: 0.3);
            }
            canvas.drawLine(Offset(0, lineY), Offset(size.width, lineY), paint);
        }
        y += groupHeight + groupGap;
    }
}

```

```

    }
    y += groupHeight + groupGap;
}
}

/// 绘制单个笔画 (贝塞尔平滑 + 压力笔锋)
void _drawStroke(Canvas canvas, PadStroke stroke) {
    if (stroke.points.length < 2) return;

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

    for (int i = 1; i < stroke.points.length; i++) {
        final prev = stroke.points[i - 1];
        final curr = stroke.points[i];

        // 压力笔锋宽度计算
        final avgPressure = (prev.pressure + curr.pressure) / 2.0;
        var width = stroke.baseWidth * (0.3 + avgPressure * 1.7);

        // 落笔过渡
        if (i < 5) width *= (i / 5.0);
        // 抬笔过渡
        final remaining = stroke.points.length - i;
        if (remaining < 5) width *= (remaining / 5.0);
        width = max(width, 0.5);

        paint.strokeWidth = width;

        if (i >= 2) {
            // 贝塞尔曲线平滑
            final pp = stroke.points[i - 2];
            final cp1x = prev.x + (curr.x - pp.x) * 0.2;
            final cp1y = prev.y + (curr.y - pp.y) * 0.2;
            final cp2x = curr.x - (curr.x - prev.x) * 0.2;
            final cp2y = curr.y - (curr.y - prev.y) * 0.2;

            final path = Path()
                ..moveTo(prev.x, prev.y)
                ..cubicTo(cp1x, cp1y, cp2x, cp2y, curr.x, curr.y);
            canvas.drawPath(path, paint);
        } else {
            canvas.drawLine(Offset(prev.x, prev.y), Offset(curr.x, curr.y), paint);
        }
    }
}

@override
bool shouldRepaint(covariant _PadStrokePainter oldDelegate) {
    return oldDelegate.strokes.length != strokes.length ||
        oldDelegate.currentStroke != currentStroke;
}

```

```
}  
}
```

repository/

repository/local_repository.dart

```
// 自然写互动课堂平板端应用软件 V1.0  
// repository/local_repository.dart - SQLite + Hive本地数据存储  
  
import 'dart:async';  
import 'dart:convert';  
  
/// 数据库表名常量  
class PadDbTables {  
  static const String homework = 'pad_homework';  
  static const String homeworkQuestion = 'pad_homework_question';  
  static const String answerProgress = 'pad_answer_progress';  
  static const String errorBook = 'pad_error_book';  
  static const String studyPlan = 'pad_study_plan';  
  static const String studyTask = 'pad_study_task';  
  static const String practiceRecord = 'pad_practice_record';  
  static const String strokeCache = 'pad_stroke_cache';  
  static const String offlineAction = 'pad_offline_action';  
  static const String usageRecord = 'pad_usage_record';  
}  
  
/// 数据库版本  
const int padDbVersion = 4;  
  
/// 作业缓存模型  
class CachedHomework {  
  final String id;  
  final String title;  
  final String subject;  
  final String teacherName;  
  final String status;  
  final String? deadline;  
  final String? content;  
  final int totalQuestions;  
  final int answeredQuestions;  
  final DateTime cachedAt;  
  
  CachedHomework({  
    required this.id,  
    required this.title,  
    required this.subject,  
    required this.teacherName,  
    required this.status,  
    this.deadline,  
    this.content,  
    this.totalQuestions = 0,  
    this.answeredQuestions = 0,  
  })
```

```

        required this.cachedAt,
    });

    Map<String, dynamic> toMap() => {
        'id': id,
        'title': title,
        'subject': subject,
        'teacher_name': teacherName,
        'status': status,
        'deadline': deadline,
        'content': content,
        'total_questions': totalQuestions,
        'answered_questions': answeredQuestions,
        'cached_at': cachedAt.toIso8601String(),
    };

    factory CachedHomework.fromMap(Map<String, dynamic> map) {
        return CachedHomework(
            id: map['id'],
            title: map['title'] ?? '',
            subject: map['subject'] ?? '',
            teacherName: map['teacher_name'] ?? '',
            status: map['status'] ?? 'pending',
            deadline: map['deadline'],
            content: map['content'],
            totalQuestions: map['total_questions'] ?? 0,
            answeredQuestions: map['answered_questions'] ?? 0,
            cachedAt: DateTime.parse(map['cached_at']),
        );
    }
}

/// 错题记录模型
class ErrorBookEntry {
    final String id;
    final String homeworkId;
    final String questionId;
    final String subject;
    final String? knowledgePoint;
    final String questionContent;
    final String? questionImageUrl;
    final String? studentAnswer;
    final String? correctAnswer;
    final String? errorReason;
    final int reviewCount;
    final DateTime createdAt;
    final DateTime? lastReviewAt;

    ErrorBookEntry({
        required this.id,
        required this.homeworkId,
        required this.questionId,
        required this.subject,
        this.knowledgePoint,
        required this.questionContent,
        this.questionImageUrl,
        this.studentAnswer,
    })

```

```

        this.correctAnswer,
        this.errorReason,
        this.reviewCount = 0,
        required this.createdAt,
        this.lastReviewAt,
    });

    Map<String, dynamic> toMap() => {
        'id': id,
        'homework_id': homeworkId,
        'question_id': questionId,
        'subject': subject,
        'knowledge_point': knowledgePoint,
        'question_content': questionContent,
        'question_image_url': questionImageUrl,
        'student_answer': studentAnswer,
        'correct_answer': correctAnswer,
        'error_reason': errorReason,
        'review_count': reviewCount,
        'created_at': createdAt.toIso8601String(),
        'last_review_at': lastReviewAt?.toIso8601String(),
    };

    factory ErrorBookEntry.fromMap(Map<String, dynamic> map) {
        return ErrorBookEntry(
            id: map['id'],
            homeworkId: map['homework_id'] ?? '',
            questionId: map['question_id'] ?? '',
            subject: map['subject'] ?? '',
            knowledgePoint: map['knowledge_point'],
            questionContent: map['question_content'] ?? '',
            questionImageUrl: map['question_image_url'],
            studentAnswer: map['student_answer'],
            correctAnswer: map['correct_answer'],
            errorReason: map['error_reason'],
            reviewCount: map['review_count'] ?? 0,
            createdAt: DateTime.parse(map['created_at']),
            lastReviewAt: map['last_review_at'] != null
                ? DateTime.parse(map['last_review_at'])
                : null,
        );
    }
}

/// 学习计划模型
class StudyPlanEntry {
    final String id;
    final String title;
    final String type;
    final String? subject;
    final DateTime startDate;
    final DateTime endDate;
    final double progress;
    final int totalTasks;
    final int completedTasks;
    final bool isActive;
}

```

```

StudyPlanEntry({
    required this.id,
    required this.title,
    required this.type,
    this.subject,
    required this.startDate,
    required this.endDate,
    this.progress = 0.0,
    this.totalTasks = 0,
    this.completedTasks = 0,
    this.isActive = true,
});

Map<String, dynamic> toMap() => {
    'id': id,
    'title': title,
    'type': type,
    'subject': subject,
    'start_date': startDate.toIso8601String(),
    'end_date': endDate.toIso8601String(),
    'progress': progress,
    'total_tasks': totalTasks,
    'completed_tasks': completedTasks,
    'is_active': isActive ? 1 : 0,
};
}

```

/// 练字练习记录模型

```

class PracticeRecord {
    final String id;
    final String templateId;
    final String character;
    final int strokeScore;
    final int structureScore;
    final int overallScore;
    final String? strokeDataJson;
    final DateTime practiceAt;

    PracticeRecord({
        required this.id,
        required this.templateId,
        required this.character,
        this.strokeScore = 0,
        this.structureScore = 0,
        this.overallScore = 0,
        this.strokeDataJson,
        required this.practiceAt,
    });

    Map<String, dynamic> toMap() => {
        'id': id,
        'template_id': templateId,
        'character': character,
        'stroke_score': strokeScore,
        'structure_score': structureScore,
        'overall_score': overallScore,
        'stroke_data_json': strokeDataJson,
    }
}

```



```

        'practice_at': practiceAt.toIso8601String(),
    };
}

/// 平板端本地数据存储仓库
/// 使用SQLite持久化存储 + Hive内存级KV缓存
/// 支持：作业缓存、错题本、学习计划、练字记录、离线操作队列、使用时长记录
class PadLocalRepository {
    /// 数据库实例（实际使用sqflite库）
    // late final Database _db;

    /// 单例
    static PadLocalRepository? _instance;
    static PadLocalRepository get instance {
        _instance ??= PadLocalRepository._internal();
        return _instance!;
    }

    PadLocalRepository._internal();

    /// 初始化数据库，创建表结构并执行版本迁移
    Future<void> initialize() async {
        // 实际调用：openDatabase(path, version: padDbVersion, ...)
        // 以下为建表SQL

        // V1: 基础表
        await _createTablesV1();

        // V2: 增加学习计划表
        await _createTablesV2();

        // V3: 增加使用时长记录表
        await _createTablesV3();

        // V4: 增加练字记录表和索引优化
        await _createTablesV4();
    }

    /// V1建表：作业缓存、作答进度、错题本、离线操作队列
    Future<void> _createTablesV1() async {
        // 作业缓存表
        const createHomework = '''
            CREATE TABLE IF NOT EXISTS ${PadDbTables.homework} (
                id TEXT PRIMARY KEY,
                title TEXT NOT NULL,
                subject TEXT NOT NULL,
                teacher_name TEXT,
                status TEXT DEFAULT 'pending',
                deadline TEXT,
                content TEXT,
                total_questions INTEGER DEFAULT 0,
                answered_questions INTEGER DEFAULT 0,
                cached_at TEXT NOT NULL
            )
        ''';

        // 作业题目缓存表

```

```

const createQuestion = '''
CREATE TABLE IF NOT EXISTS ${PadDbTables.homeworkQuestion} (
  id TEXT PRIMARY KEY,
  homework_id TEXT NOT NULL,
  question_index INTEGER,
  type TEXT DEFAULT 'write',
  content TEXT,
  image_url TEXT,
  options TEXT,
  correct_answer TEXT,
  FOREIGN KEY (homework_id) REFERENCES ${PadDbTables.homework}(id)
)
''';

// 作答进度暂存表
const createProgress = '''
CREATE TABLE IF NOT EXISTS ${PadDbTables.answerProgress} (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  homework_id TEXT NOT NULL,
  question_id TEXT NOT NULL,
  text_answer TEXT,
  stroke_data TEXT,
  saved_at TEXT NOT NULL,
  UNIQUE(homework_id, question_id)
)
''';

// 错题本表
const createErrorBook = '''
CREATE TABLE IF NOT EXISTS ${PadDbTables.errorBook} (
  id TEXT PRIMARY KEY,
  homework_id TEXT,
  question_id TEXT,
  subject TEXT NOT NULL,
  knowledge_point TEXT,
  question_content TEXT NOT NULL,
  question_image_url TEXT,
  student_answer TEXT,
  correct_answer TEXT,
  error_reason TEXT,
  review_count INTEGER DEFAULT 0,
  created_at TEXT NOT NULL,
  last_review_at TEXT
)
''';

// 离线操作队列表
const createOffline = '''
CREATE TABLE IF NOT EXISTS ${PadDbTables.offlineAction} (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  action_type TEXT NOT NULL,
  payload TEXT NOT NULL,
  retry_count INTEGER DEFAULT 0,
  created_at TEXT NOT NULL,
  status TEXT DEFAULT 'pending'
)
''';

```

```

// 笔迹暂存表
const createStroke = '''
    CREATE TABLE IF NOT EXISTS ${PadDbTables.strokeCache} (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        homework_id TEXT,
        question_id TEXT,
        page_id TEXT,
        stroke_json TEXT NOT NULL,
        pen_mac TEXT,
        created_at TEXT NOT NULL
    )
''';

// 实际执行建表SQL
// await _db.execute(createHomework);
// await _db.execute(createQuestion);
// await _db.execute(createProgress);
// await _db.execute(createErrorBook);
// await _db.execute(createOffline);
// await _db.execute(createStroke);
}

/// V2建表: 学习计划与任务
Future<void> _createTablesV2() async {
    const createPlan = '''
        CREATE TABLE IF NOT EXISTS ${PadDbTables.studyPlan} (
            id TEXT PRIMARY KEY,
            title TEXT NOT NULL,
            type TEXT NOT NULL,
            subject TEXT,
            start_date TEXT NOT NULL,
            end_date TEXT NOT NULL,
            progress REAL DEFAULT 0.0,
            total_tasks INTEGER DEFAULT 0,
            completed_tasks INTEGER DEFAULT 0,
            is_active INTEGER DEFAULT 1
        )
    ''';

    const createTask = '''
        CREATE TABLE IF NOT EXISTS ${PadDbTables.studyTask} (
            id TEXT PRIMARY KEY,
            plan_id TEXT NOT NULL,
            title TEXT NOT NULL,
            description TEXT,
            target_date TEXT,
            is_completed INTEGER DEFAULT 0,
            completed_at TEXT,
            FOREIGN KEY (plan_id) REFERENCES ${PadDbTables.studyPlan}(id)
        )
    ''';

    // await _db.execute(createPlan);
    // await _db.execute(createTask);
}

```

```

/// V3建表: 使用时长记录
Future<void> _createTablesV3() async {
  const createUsage = '''
    CREATE TABLE IF NOT EXISTS ${PadDbTables.usageRecord} (
      id INTEGER PRIMARY KEY AUTOINCREMENT,
      date TEXT NOT NULL,
      app_name TEXT DEFAULT 'writech',
      subject TEXT,
      duration_seconds INTEGER DEFAULT 0,
      start_time TEXT NOT NULL,
      end_time TEXT
    )
  ''';

  // await _db.execute(createUsage);
}

/// V4建表: 练字记录 + 索引
Future<void> _createTablesV4() async {
  const createPractice = '''
    CREATE TABLE IF NOT EXISTS ${PadDbTables.practiceRecord} (
      id TEXT PRIMARY KEY,
      template_id TEXT NOT NULL,
      character TEXT NOT NULL,
      stroke_score INTEGER DEFAULT 0,
      structure_score INTEGER DEFAULT 0,
      overall_score INTEGER DEFAULT 0,
      stroke_data_json TEXT,
      practice_at TEXT NOT NULL
    )
  ''';

  // 索引优化
  const indexHomeworkStatus = '''
    CREATE INDEX IF NOT EXISTS idx_homework_status
    ON ${PadDbTables.homework}(status)
  ''';
  const indexErrorSubject = '''
    CREATE INDEX IF NOT EXISTS idx_error_subject
    ON ${PadDbTables.errorBook}(subject)
  ''';
  const indexPracticeChar = '''
    CREATE INDEX IF NOT EXISTS idx_practice_char
    ON ${PadDbTables.practiceRecord}(character)
  ''';

  // await _db.execute(createPractice);
  // await _db.execute(indexHomeworkStatus);
  // await _db.execute(indexErrorSubject);
  // await _db.execute(indexPracticeChar);
}

// =====
// 作业缓存 CRUD
// =====

/// 缓存作业到本地 (用于离线作答)

```

```

Future<void> cacheHomework(CachedHomework homework) async {
    // await _db.insert(
    //     PadDbTables.homework,
    //     homework.toMap(),
    //     conflictAlgorithm: ConflictAlgorithm.replace,
    // );
}

/// 获取本地缓存的作业列表
Future<List<CachedHomework>> getCachedHomeworks({
    String? status,
    int limit = 50,
}) async {
    // String where = '';
    // List<dynamic> whereArgs = [];
    // if (status != null) {
    //     where = 'status = ?';
    //     whereArgs = [status];
    // }
    // final maps = await _db.query(
    //     PadDbTables.homework,
    //     where: where.isNotEmpty ? where : null,
    //     whereArgs: whereArgs.isNotEmpty ? whereArgs : null,
    //     orderBy: 'cached_at DESC',
    //     limit: limit,
    // );
    // return maps.map((m) => CachedHomework.fromMap(m)).toList();
    return [];
}

/// 保存作答进度到本地
Future<void> saveAnswerProgress({
    required String homeworkId,
    required String questionId,
    String? textAnswer,
    String? strokeDataJson,
}) async {
    // await _db.insert(
    //     PadDbTables.answerProgress,
    //     {
    //         'homework_id': homeworkId,
    //         'question_id': questionId,
    //         'text_answer': textAnswer,
    //         'stroke_data': strokeDataJson,
    //         'saved_at': DateTime.now().toIso8601String(),
    //     },
    //     conflictAlgorithm: ConflictAlgorithm.replace,
    // );
}

/// 获取某作业的所有作答进度
Future<Map<String, Map<String, dynamic>>> getAnswerProgress(
    String homeworkId,
) async {
    // final maps = await _db.query(
    //     PadDbTables.answerProgress,
    //     where: 'homework_id = ?',

```

```

    //   whereArgs: [homeworkId],
    // );
    // final result = <String, Map<String, dynamic>>{};
    // for (final m in maps) {
    //   result[m['question_id'] as String] = m;
    // }
    // return result;
    return {};
}

// =====
// 错题本 CRUD
// =====

/// 添加错题记录
Future<void> addErrorEntry(ErrorBookEntry entry) async {
    // await _db.insert(
    //   PadDbTables.errorBook,
    //   entry.toMap(),
    //   conflictAlgorithm: ConflictAlgorithm.replace,
    // );
}

/// 获取错题列表（支持按科目/知识点筛选）
Future<List<ErrorBookEntry>> getErrorEntries({
    String? subject,
    String? knowledgePoint,
    int limit = 50,
    int offset = 0,
}) async {
    // final conditions = <String>[];
    // final args = <dynamic>[];
    // if (subject != null) {
    //   conditions.add('subject = ?');
    //   args.add(subject);
    // }
    // if (knowledgePoint != null) {
    //   conditions.add('knowledge_point = ?');
    //   args.add(knowledgePoint);
    // }
    // final maps = await _db.query(
    //   PadDbTables.errorBook,
    //   where: conditions.isNotEmpty ? conditions.join(' AND ') : null,
    //   whereArgs: args.isNotEmpty ? args : null,
    //   orderBy: 'created_at DESC',
    //   limit: limit,
    //   offset: offset,
    // );
    // return maps.map((m) => ErrorBookEntry.fromMap(m)).toList();
    return [];
}

/// 更新错题复习次数
Future<void> updateErrorReviewCount(String entryId) async {
    // await _db.rawUpdate(''
    //   UPDATE ${PadDbTables.errorBook}
    //   SET review_count = review_count + 1,

```

```

        //      last_review_at = ?
        //      WHERE id = ?
        //      '', [DateTime.now().toIso8601String(), entryId]);
    }

    /// 获取错题统计 (按科目分组计数)
    Future<Map<String, int>> getErrorStatsBySubject() async {
        // final maps = await _db.rawQuery('''
        //     SELECT subject, COUNT(*) as count
        //     FROM ${PadDbTables.errorBook}
        //     GROUP BY subject
        //     ''');
        // return {for (var m in maps) m['subject'] as String: m['count'] as int};
        return {};
    }

    // =====
    // 学习计划 CRUD
    // =====

    /// 保存学习计划
    Future<void> saveStudyPlan(StudyPlanEntry plan) async {
        // await _db.insert(
        //     PadDbTables.studyPlan,
        //     plan.toMap(),
        //     conflictAlgorithm: ConflictAlgorithm.replace,
        // );
    }

    /// 获取活跃的学习计划列表
    Future<List<Map<String, dynamic>>> getActiveStudyPlans() async {
        // return await _db.query(
        //     PadDbTables.studyPlan,
        //     where: 'is_active = 1',
        //     orderBy: 'start_date ASC',
        // );
        return [];
    }

    /// 更新学习计划进度
    Future<void> updatePlanProgress(
        String planId,
        double progress,
        int completedTasks,
    ) async {
        // await _db.update(
        //     PadDbTables.studyPlan,
        //     {'progress': progress, 'completed_tasks': completedTasks},
        //     where: 'id = ?',
        //     whereArgs: [planId],
        // );
    }

    // =====
    // 练字记录
    // =====

```

```

/// 保存练字记录
Future<void> savePracticeRecord(PracticeRecord record) async {
  // await _db.insert(
  //   PadDbTables.practiceRecord,
  //   record.toMap(),
  //   conflictAlgorithm: ConflictAlgorithm.replace,
  // );
}

/// 获取某字的练习历史（查看进步轨迹）
Future<List<Map<String, dynamic>>> getPracticeHistory(
  String character, {
    int limit = 20,
  }) async {
  // return await _db.query(
  //   PadDbTables.practiceRecord,
  //   where: 'character = ?',
  //   whereArgs: [character],
  //   orderBy: 'practice_at DESC',
  //   limit: limit,
  // );
  return [];
}

// =====
// 离线操作队列
// =====

/// 添加离线操作到队列
Future<void> enqueueOfflineAction(
  String actionType,
  Map<String, dynamic> payload,
) async {
  // await _db.insert(PadDbTables.offlineAction, {
  //   'action_type': actionType,
  //   'payload': jsonEncode(payload),
  //   'created_at': DateTime.now().toIso8601String(),
  //   'status': 'pending',
  // });
}

/// 获取待执行的离线操作
Future<List<Map<String, dynamic>>> getPendingOfflineActions() async {
  // return await _db.query(
  //   PadDbTables.offlineAction,
  //   where: 'status = ? AND retry_count < 5',
  //   whereArgs: ['pending'],
  //   orderBy: 'created_at ASC',
  // );
  return [];
}

/// 标记离线操作完成
Future<void> markOfflineActionDone(int actionId) async {
  // await _db.update(
  //   PadDbTables.offlineAction,
  //   {'status': 'done'},

```



```

    // where: 'id = ?',
    // whereArgs: [actionId],
    // );
}

// =====
// 使用时长记录
// =====

/// 记录使用时长
Future<void> recordUsage({
    required String date,
    required int durationSeconds,
    required String startTime,
    String? endTime,
    String? subject,
}) async {
    // await _db.insert(PadDbTables.usageRecord, {
    //     'date': date,
    //     'duration_seconds': durationSeconds,
    //     'start_time': startTime,
    //     'end_time': endTime,
    //     'subject': subject,
    // });
}

/// 获取某日使用总时长 (秒)
Future<int> getDailyUsage(String date) async {
    // final result = await _db.rawQuery('''
    //     SELECT COALESCE(SUM(duration_seconds), 0) as total
    //     FROM ${PadDbTables.usageRecord}
    //     WHERE date = ?
    //     ''', [date]);
    // return result.first['total'] as int? ?? 0;
    return 0;
}

// =====
// 数据库维护
// =====

/// 清理过期缓存数据 (30天前的作业缓存、90天前的笔迹暂存)
Future<void> cleanExpiredData() async {
    final thirtyDaysAgo = DateTime.now()
        .subtract(const Duration(days: 30))
        .toIso8601String();
    final ninetyDaysAgo = DateTime.now()
        .subtract(const Duration(days: 90))
        .toIso8601String();

    // await _db.delete(
    //     PadDbTables.homework,
    //     where: 'cached_at < ? AND status IN (?, ?)',
    //     whereArgs: [thirtyDaysAgo, 'graded', 'expired'],
    // );
    // await _db.delete(
    //     PadDbTables.strokeCache,

```

```

    // where: 'created_at < ?',
    // whereArgs: [ninetyDaysAgo],
    // );
    // await _db.delete(
    //   PadDbTables.offlineAction,
    //   where: 'status = ? AND created_at < ?',
    //   whereArgs: ['done', thirtyDaysAgo],
    // );
  }

  /// 获取本地数据库存储大小（字节）
  Future<int> getDatabaseSize() async {
    // final dbPath = await getDatabasesPath();
    // final file = File('$dbPath/writtech_pad.db');
    // return file.existsSync() ? file.lengthSync() : 0;
    return 0;
  }

  /// 关闭数据库
  Future<void> close() async {
    // await _db.close();
  }
}

```

service/

service/api_service.dart

```

// 自然写互动课堂平板端应用软件 V1.0
// service/api_service.dart - 云平台API服务（Dio HTTP客户端）

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

/// 云平台API基础路径配置
class ApiConfig {
  /// 生产环境API地址
  static const String productionBaseUrl = 'https://api.writtech.com/v1';

  /// 测试环境API地址
  static const String stagingBaseUrl = 'https://staging-api.writtech.com/v1';

  /// 连接超时时间（毫秒）
  static const int connectTimeout = 15000;

  /// 接收超时时间（毫秒）
  static const int receiveTimeout = 30000;

  /// Token刷新路径

```

```

static const String refreshTokenPath = '/auth/refresh';

/// 最大重试次数
static const int maxRetryCount = 3;

/// HMAC签名密钥标识
static const String hmacKeyId = 'writech-pad-v1';
}

/// API响应数据统一封装
class ApiResponse<T> {
    final int code;
    final String message;
    final T? data;
    final String? requestId;

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

    /// 判断请求是否成功
    bool get isSuccess => code == 0 || code == 200;

    /// 从JSON解析响应
    factory ApiResponse.fromJson(
        Map<String, dynamic> json,
        T Function(dynamic)? fromData,
    ) {
        return ApiResponse(
            code: json['code'] ?? -1,
            message: json['message'] ?? '未知错误',
            data: json['data'] != null && fromData != null
                ? fromData(json['data'])
                : json['data'] as T?,
            requestId: json['request_id'],
        );
    }
}

/// 离线请求队列项
class OfflineRequest {
    final String id;
    final String method;
    final String path;
    final Map<String, dynamic>? data;
    final DateTime createdAt;
    int retryCount;

    OfflineRequest({
        required this.id,
        required this.method,
        required this.path,
        this.data,
        required this.createdAt,
    });
}

```

```

        this.retryCount = 0,
    });

    /// 序列化为JSON用于本地持久化
    Map<String, dynamic> toJson() => {
      'id': id,
      'method': method,
      'path': path,
      'data': data,
      'created_at': createdAt.toIso8601String(),
      'retry_count': retryCount,
    };

    /// 从JSON反序列化
    factory OfflineRequest.fromJson(Map<String, dynamic> json) {
      return OfflineRequest(
        id: json['id'],
        method: json['method'],
        path: json['path'],
        data: json['data'],
        createdAt: DateTime.parse(json['created_at']),
        retryCount: json['retry_count'] ?? 0,
      );
    }
  }
}

/// 平板端云平台API服务
/// 负责与云平台的所有HTTP通信，包括：
/// - JWT双令牌认证与自动刷新
/// - HMAC-SHA256请求签名
/// - 离线请求队列暂存
/// - 学生简化登录（班级+姓名/学号）
class PadApiService {
  late final Dio _dio;
  final FlutterSecureStorage _secureStorage = const FlutterSecureStorage();

  /// 当前访问令牌
  String? _accessToken;

  /// 刷新令牌
  String? _refreshToken;

  /// Token刷新锁，防止并发刷新
  Completer<bool>? _refreshCompleter;

  /// 离线请求队列
  final List<OfflineRequest> _offlineQueue = [];

  /// 网络状态标志
  bool _isOnline = true;

  /// API事件流控制器（登录状态变化等）
  final StreamController<String> _eventController =
    StreamController<String>.broadcast();

  /// API事件流
  Stream<String> get eventStream => _eventController.stream;

```

```

/// 单例实例
static PadApiService? _instance;

/// 获取单例
static PadApiService get instance {
  _instance ??= PadApiService._internal();
  return _instance!;
}

/// 私有构造函数, 初始化Dio客户端
PadApiService._internal() {
  _dio = Dio(BaseOptions(
    baseUrl: ApiConfig.productionBaseUrl,
    connectTimeout: Duration(milliseconds: ApiConfig.connectTimeout),
    receiveTimeout: Duration(milliseconds: ApiConfig.receiveTimeout),
    headers: {
      'Content-Type': 'application/json',
      'X-Client-Platform': 'pad',
      'X-Client-Version': '1.0.0',
    },
  ));

  // 添加请求拦截器: 自动附加Token和HMAC签名
  _dio.interceptors.add(InterceptorsWrapper(
    onRequest: _onRequest,
    onResponse: _onResponse,
    onError: _onError,
  ));

  // 从安全存储恢复令牌
  _restoreTokens();
}

/// 从安全存储恢复上次保存的令牌
Future<void> _restoreTokens() async {
  _accessToken = await _secureStorage.read(key: 'access_token');
  _refreshToken = await _secureStorage.read(key: 'refresh_token');
}

/// 请求拦截器: 附加Authorization头和HMAC签名
void _onRequest(
  RequestOptions options,
  RequestInterceptorHandler handler,
) {
  // 附加JWT访问令牌
  if (_accessToken != null) {
    options.headers['Authorization'] = 'Bearer $_accessToken';
  }

  // 生成HMAC-SHA256请求签名
  final timestamp = DateTime.now().millisecondsSinceEpoch.toString();
  options.headers['X-Timestamp'] = timestamp;
  final signature = _generateSignature(
    options.method,
    options.path,
    timestamp,

```

```

        options.data,
    );
    options.headers['X-Signature'] = signature;

    handler.next(options);
}

/// 响应拦截器: 统一处理响应
void _onResponse(
    Response response,
    ResponseInterceptorHandler handler,
) {
    handler.next(response);
}

/// 错误拦截器: 处理401自动刷新Token、离线暂存等
Future<void> _onError(
    DioException error,
    ErrorInterceptorHandler handler,
) async {
    // 网络不可用时, 将请求加入离线队列
    if (error.type == DioExceptionType.connectionError ||
        error.type == DioExceptionType.connectionTimeout) {
        _isOnline = false;
        _enqueueOfflineRequest(error.requestOptions);
        handler.reject(error);
        return;
    }

    // 401未授权: 尝试刷新Token后重试
    if (error.response?.statusCode == 401) {
        final refreshSuccess = await _refreshAccessToken();
        if (refreshSuccess) {
            // Token刷新成功, 使用新Token重试原请求
            final retryOptions = error.requestOptions;
            retryOptions.headers['Authorization'] = 'Bearer $_accessToken';
            try {
                final response = await _dio.fetch(retryOptions);
                handler.resolve(response);
                return;
            } catch (retryError) {
                // 重试也失败了
            }
        } else {
            // Token刷新失败, 通知登出
            _eventController.add('token_expired');
        }
    }

    handler.reject(error);
}

/// 生成HMAC-SHA256请求签名
/// 签名内容: METHOD\nPATH\nTIMESTAMP\nBODY_SHA256
String _generateSignature(
    String method,
    String path,

```

```

    String timestamp,
    dynamic body,
  ) {
    // 计算请求体SHA256哈希
    String bodyHash = '';
    if (body != null) {
      final bodyStr = body is String ? body : jsonEncode(body);
      bodyHash = sha256.convert(utf8.encode(bodyStr)).toString();
    }

    // 拼接签名原文
    final signContent = '$method\n$path\n$timestamp\n$bodyHash';
    final hmacKey = utf8.encode(ApiConfig.hmacKeyId);
    final hmac = Hmac(sha256, hmacKey);
    final digest = hmac.convert(utf8.encode(signContent));

    return digest.toString();
  }

  /// 刷新访问令牌
  /// 使用Completer防止并发多次刷新
  Future<bool> _refreshAccessToken() async {
    // 如果已经在刷新中，等待结果
    if (_refreshCompleter != null) {
      return _refreshCompleter!.future;
    }

    _refreshCompleter = Completer<bool>();

    try {
      if (_refreshToken == null) {
        _refreshCompleter!.complete(false);
        return false;
      }

      // 发送刷新请求（不经过拦截器避免死循环）
      final response = await Dio().post(
        '${ApiConfig.productionBaseUrl}${ApiConfig.refreshTokenPath}',
        data: {'refresh_token': _refreshToken},
      );

      if (response.statusCode == 200 && response.data['code'] == 0) {
        _accessToken = response.data['data']['access_token'];
        _refreshToken = response.data['data']['refresh_token'];

        // 持久化新令牌到安全存储
        await _secureStorage.write(
          key: 'access_token',
          value: _accessToken,
        );
        await _secureStorage.write(
          key: 'refresh_token',
          value: _refreshToken,
        );

        _refreshCompleter!.complete(true);
        return true;
      }
    }
  }

```

```

    }

    _refreshCompleter!.complete(false);
    return false;
} catch (e) {
    _refreshCompleter!.complete(false);
    return false;
} finally {
    _refreshCompleter = null;
}
}

/// 将失败的请求加入离线队列
void _enqueueOfflineRequest(RequestOptions options) {
    final offlineReq = OfflineRequest(
        id: DateTime.now().microsecondsSinceEpoch.toString(),
        method: options.method,
        path: options.path,
        data: options.data is Map ? options.data : null,
        createdAt: DateTime.now(),
    );
    _offlineQueue.add(offlineReq);
}

/// 网络恢复后, 批量重发离线队列中的请求
Future<void> flushOfflineQueue() async {
    if (_offlineQueue.isEmpty) return;

    _isOnline = true;
    final pendingRequests = List<OfflineRequest>.from(_offlineQueue);
    _offlineQueue.clear();

    for (final req in pendingRequests) {
        try {
            if (req.retryCount >= ApiConfig.maxRetryCount) continue;
            req.retryCount++;

            switch (req.method.toUpperCase()) {
                case 'POST':
                    await _dio.post(req.path, data: req.data);
                    break;
                case 'PUT':
                    await _dio.put(req.path, data: req.data);
                    break;
                case 'DELETE':
                    await _dio.delete(req.path);
                    break;
                default:
                    await _dio.get(req.path);
            }
        } catch (e) {
            // 重发失败的请求重新加入队列
            if (req.retryCount < ApiConfig.maxRetryCount) {
                _offlineQueue.add(req);
            }
        }
    }
}

```



```

}

// =====
// 学生登录接口（简化登录，班级+姓名/学号）
// =====

/// 学生简化登录（无需手机号，仅班级+姓名或学号）
Future<ApiResponse<Map<String, dynamic>>> studentLogin({
  required String schoolCode,
  required String classId,
  required String studentName,
  String? studentNo,
}) async {
  try {
    final response = await _dio.post('/auth/student/login', data: {
      'school_code': schoolCode,
      'class_id': classId,
      'student_name': studentName,
      'student_no': studentNo,
      'device_type': 'pad',
    });

    final result = ApiResponse<Map<String, dynamic>>.fromJson(
      response.data,
      (data) => data as Map<String, dynamic>,
    );

    // 保存登录令牌
    if (result.isSuccess && result.data != null) {
      _accessToken = result.data!['access_token'];
      _refreshToken = result.data!['refresh_token'];
      await _secureStorage.write(
        key: 'access_token',
        value: _accessToken,
      );
      await _secureStorage.write(
        key: 'refresh_token',
        value: _refreshToken,
      );
    }

    return result;
  } on DioException catch (e) {
    return ApiResponse(code: -1, message: e.message ?? '网络请求失败');
  }
}

/// 教师登录（手机号+验证码）
Future<ApiResponse<Map<String, dynamic>>> teacherLogin({
  required String phone,
  required String verifyCode,
}) async {
  try {
    final response = await _dio.post('/auth/teacher/login', data: {
      'phone': phone,
      'verify_code': verifyCode,
      'device_type': 'pad',
    });

```

```

});

final result = ApiResponse<Map<String, dynamic>>.fromJson(
  response.data,
  (data) => data as Map<String, dynamic>,
);

if (result.isSuccess && result.data != null) {
  _accessToken = result.data!['access_token'];
  _refreshToken = result.data!['refresh_token'];
  await _secureStorage.write(
    key: 'access_token',
    value: _accessToken,
  );
  await _secureStorage.write(
    key: 'refresh_token',
    value: _refreshToken,
  );
}

return result;
} on DioException catch (e) {
  return ApiResponse(code: -1, message: e.message ?? '网络请求失败');
}
}

// =====
// 作业相关接口
// =====

/// 获取学生待完成作业列表
Future<ApiResponse<List<dynamic>>> getHomeworkList({
  int page = 1,
  int pageSize = 20,
  String? status,
}) async {
  try {
    final response = await _dio.get('/homework/list', queryParameters: {
      'page': page,
      'page_size': pageSize,
      if (status != null) 'status': status,
    });
    return ApiResponse<List<dynamic>>.fromJson(
      response.data,
      (data) => data as List<dynamic>,
    );
  } on DioException catch (e) {
    return ApiResponse(code: -1, message: e.message ?? '获取作业列表失败');
  }
}

/// 下载作业详情 (含题目内容, 支持离线作答)
Future<ApiResponse<Map<String, dynamic>>> downloadHomework(
  String homeworkId,
) async {
  try {
    final response = await _dio.get('/homework/detail/$homeworkId');

```

```

        return ApiResponse<Map<String, dynamic>>.fromJson(
            response.data,
            (data) => data as Map<String, dynamic>,
        );
    } on DioException catch (e) {
        return ApiResponse(code: -1, message: e.message ?? '下载作业失败');
    }
}

/// 提交作业 (含笔迹数据)
Future<ApiResponse<Map<String, dynamic>>> submitHomework({
    required String homeworkId,
    required List<Map<String, dynamic>> strokePages,
    int? timeCostSeconds,
}) async {
    try {
        final response = await _dio.post('/homework/submit', data: {
            'homework_id': homeworkId,
            'stroke_pages': strokePages,
            'time_cost': timeCostSeconds,
            'submit_time': DateTime.now().toIso8601String(),
        });
        return ApiResponse<Map<String, dynamic>>.fromJson(
            response.data,
            (data) => data as Map<String, dynamic>,
        );
    } on DioException catch (e) {
        // 离线时暂存提交请求
        if (!_isOnline) {
            _enqueueOfflineRequest(e.requestOptions);
        }
        return ApiResponse(code: -1, message: e.message ?? '提交作业失败');
    }
}

/// 获取作业批改结果
Future<ApiResponse<Map<String, dynamic>>> getHomeworkResult(
    String homeworkId,
) async {
    try {
        final response = await _dio.get('/homework/result/$homeworkId');
        return ApiResponse<Map<String, dynamic>>.fromJson(
            response.data,
            (data) => data as Map<String, dynamic>,
        );
    } on DioException catch (e) {
        return ApiResponse(code: -1, message: e.message ?? '获取批改结果失败');
    }
}

// =====
// 字帖练习接口
// =====

/// 获取字帖模板列表 (按年级/学科分类)
Future<ApiResponse<List<dynamic>>> getCopybookTemplates({
    required String grade,

```

```

    String? subject,
    int page = 1,
  }) async {
    try {
      final response = await _dio.get('/copybook/templates', queryParameters: {
        'grade': grade,
        'subject': subject,
        'page': page,
      });
      return ApiResponse<List<dynamic>>.fromJson(
        response.data,
        (data) => data as List<dynamic>,
      );
    } on DioException catch (e) {
      return ApiResponse(code: -1, message: e.message ?? '获取字帖失败');
    }
  }
}

```

/// 上传练字笔迹评分

```

Future<ApiResponse<Map<String, dynamic>>> submitPracticeStroke({
  required String templateId,
  required String character,
  required List<Map<String, dynamic>> strokes,
}) async {
  try {
    final response = await _dio.post('/copybook/evaluate', data: {
      'template_id': templateId,
      'character': character,
      'strokes': strokes,
    });
    return ApiResponse<Map<String, dynamic>>.fromJson(
      response.data,
      (data) => data as Map<String, dynamic>,
    );
  } on DioException catch (e) {
    return ApiResponse(code: -1, message: e.message ?? '提交练字评分失败');
  }
}

```

```

// =====
// 错题本接口
// =====

```

/// 获取错题列表（按知识点/科目分类）

```

Future<ApiResponse<List<dynamic>>> getErrorBookList({
  String? subject,
  String? knowledgePoint,
  int page = 1,
  int pageSize = 20,
}) async {
  try {
    final response = await _dio.get('/error-book/list', queryParameters: {
      if (subject != null) 'subject': subject,
      if (knowledgePoint != null) 'knowledge_point': knowledgePoint,
      'page': page,
      'page_size': pageSize,
    });
  }
}

```

```

        return ApiResponse<List<dynamic>>.fromJson(
            response.data,
            (data) => data as List<dynamic>,
        );
    } on DioException catch (e) {
        return ApiResponse(code: -1, message: e.message ?? '获取错题本失败');
    }
}

// =====
// 学情与学习计划接口
// =====

/// 获取学生个人学情概览
Future<ApiResponse<Map<String, dynamic>>> getStudentProfile() async {
    try {
        final response = await _dio.get('/profile/student/overview');
        return ApiResponse<Map<String, dynamic>>.fromJson(
            response.data,
            (data) => data as Map<String, dynamic>,
        );
    } on DioException catch (e) {
        return ApiResponse(code: -1, message: e.message ?? '获取学情失败');
    }
}

/// 获取学习计划列表
Future<ApiResponse<List<dynamic>>> getStudyPlans() async {
    try {
        final response = await _dio.get('/study-plan/list');
        return ApiResponse<List<dynamic>>.fromJson(
            response.data,
            (data) => data as List<dynamic>,
        );
    } on DioException catch (e) {
        return ApiResponse(code: -1, message: e.message ?? '获取学习计划失败');
    }
}

/// 更新学习计划进度
Future<ApiResponse<void>> updateStudyPlanProgress({
    required String planId,
    required String taskId,
    required double progress,
}) async {
    try {
        final response = await _dio.put('/study-plan/progress', data: {
            'plan_id': planId,
            'task_id': taskId,
            'progress': progress,
            'update_time': DateTime.now().toIso8601String(),
        });
        return ApiResponse<void>.fromJson(response.data, null);
    } on DioException catch (e) {
        return ApiResponse(code: -1, message: e.message ?? '更新进度失败');
    }
}

```

```

/// 登出，清除本地令牌
Future<void> logout() async {
  try {
    await _dio.post('/auth/logout');
  } catch (_) {
    // 忽略登出请求失败
  }
  _accessToken = null;
  _refreshToken = null;
  await _secureStorage.delete(key: 'access_token');
  await _secureStorage.delete(key: 'refresh_token');
  _eventController.add('logged_out');
}

/// 释放资源
void dispose() {
  _eventController.close();
  _dio.close();
}
}

```

service/ble_service.dart

```

// 自然写互动课堂平板端应用软件 V1.0
// service/ble_service.dart - BLE蓝牙点阵笔连接服务

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

/// BLE服务UUID常量定义
/// 基于自然写点阵笔自定义GATT Service规范
class PadBleConstants {
  /// 点阵笔主服务UUID
  static const String penServiceUuid = '0000ffe0-0000-1000-8000-00805f9b34fb';

  /// 笔迹坐标数据特征值UUID (Notify)
  static const String strokeCharUuid = '0000ffe1-0000-1000-8000-00805f9b34fb';

  /// 笔控制指令特征值UUID (Write)
  static const String controlCharUuid = '0000ffe2-0000-1000-8000-00805f9b34fb';

  /// 电量信息特征值UUID (Read/Notify)
  static const String batteryCharUuid = '0000ffe3-0000-1000-8000-00805f9b34fb';

  /// 设备信息服务UUID
  static const String deviceInfoUuid = '0000180a-0000-1000-8000-00805f9b34fb';

  /// 扫描超时时间 (秒)
  static const int scanTimeoutSeconds = 15;

  /// 自动重连延迟 (秒)
  static const int reconnectDelaySeconds = 3;
}

```

```

    /// 最大重连次数
    static const int maxReconnectAttempts = 10;

    /// MTU协商大小
    static const int requestedMtu = 247;

    /// 笔迹数据缓冲批量回调阈值
    static const int strokeBatchSize = 8;

    /// 电量读取间隔（秒）
    static const int batteryReadInterval = 60;
}

/// 单个笔迹坐标点数据
class PadPenPoint {
    /// X坐标（0.01mm精度，16位无符号）
    final double x;

    /// Y坐标（0.01mm精度，16位无符号）
    final double y;

    /// 压力值（0-255，8位无符号）
    final int pressure;

    /// 时间戳（相对值，16位无符号，单位ms）
    final int timestamp;

    /// 是否为落笔点
    final bool isPenDown;

    PadPenPoint({
        required this.x,
        required this.y,
        required this.pressure,
        required this.timestamp,
        this.isPenDown = false,
    });

    @override
    String toString() =>
        'PadPenPoint(x: ${x.toStringAsFixed(2)}, y: ${y.toStringAsFixed(2)}, '
        'p: $pressure, t: $timestamp)';
}

/// 点阵笔设备信息
class PadPenDevice {
    /// 设备蓝牙MAC地址
    final String macAddress;

    /// 设备名称
    final String name;

    /// 信号强度（RSSI）
    int rssi;

    /// 当前连接状态
    PenConnectionState connectionState;

```

```

    /// 电量百分比 (0-100)
    int batteryLevel;

    /// 固件版本号
    String? firmwareVersion;

    /// 当前所在点阵码页面ID
    String? currentPageId;

    PadPenDevice({
        required this.macAddress,
        required this.name,
        this.rssi = -100,
        this.connectionState = PenConnectionState.disconnected,
        this.batteryLevel = -1,
        this.firmwareVersion,
        this.currentPageId,
    });
}

/// 笔连接状态枚举
enum PenConnectionState {
    /// 未连接
    disconnected,

    /// 正在扫描
    scanning,

    /// 正在连接
    connecting,

    /// 已连接
    connected,

    /// 正在断开
    disconnecting,

    /// 自动重连中
    reconnecting,
}

/// 笔迹数据事件 (批量坐标点回调)
class PenStrokeEvent {
    /// 来源笔的MAC地址
    final String penMac;

    /// 坐标点列表
    final List<PadPenPoint> points;

    /// 所在页面ID (点阵码识别)
    final String? pageId;

    PenStrokeEvent({
        required this.penMac,
        required this.points,
        this.pageId,
    });
}

```



```

});
}

/// BLE蓝牙点阵笔连接服务
/// 负责扫描、连接、数据接收、电量监控、自动重连等功能
/// 平板端支持同时连接1支笔（学生个人使用场景）
class PadBleService {
    /// 已发现的设备列表
    final List<PadPenDevice> _discoveredDevices = [];

    /// 当前已连接的笔
    PadPenDevice? _connectedPen;

    /// 笔迹数据缓冲区（累积到阈值后批量回调）
    final List<PadPenPoint> _strokeBuffer = [];

    /// 扫描结果流
    final StreamController<List<PadPenDevice>> _scanController =
        StreamController<List<PadPenDevice>>.broadcast();

    /// 笔迹数据事件流
    final StreamController<PenStrokeEvent> _strokeController =
        StreamController<PenStrokeEvent>.broadcast();

    /// 连接状态变化流
    final StreamController<PenConnectionState> _connectionController =
        StreamController<PenConnectionState>.broadcast();

    /// 电量变化流
    final StreamController<int> _batteryController =
        StreamController<int>.broadcast();

    /// 自动重连计数器
    int _reconnectAttempts = 0;

    /// 重连定时器
    Timer? _reconnectTimer;

    /// 电量读取定时器
    Timer? _batteryTimer;

    /// 是否正在扫描
    bool _isScanning = false;

    /// 公开的流
    Stream<List<PadPenDevice>> get scanStream => _scanController.stream;
    Stream<PenStrokeEvent> get strokeStream => _strokeController.stream;
    Stream<PenConnectionState> get connectionStream =>
        _connectionController.stream;
    Stream<int> get batteryStream => _batteryController.stream;

    /// 获取当前连接的笔
    PadPenDevice? get connectedPen => _connectedPen;

    /// 开始扫描附近的点阵笔设备
    /// 按服务UUID过滤，仅发现自然写点阵笔
    Future<void> startScan() async {

```

```

    if (!_isScanning) return;
    _isScanning = true;
    _discoveredDevices.clear();

    // 通知扫描状态
    _connectionController.add(PenConnectionState.scanning);

    // 模拟BLE扫描（实际使用flutter_blue_plus库）
    // 过滤条件：仅发现包含pen_service_uuid的设备
    // scanFilters: [ScanFilter(serviceUuid: PadBleConstants.penServiceUuid)]

    // 设置扫描超时
    Timer(Duration(seconds: PadBleConstants.scanTimeoutSeconds), () {
      stopScan();
    });
  }

  /// 停止扫描
  Future<void> stopScan() async {
    _isScanning = false;
    // 实际调用：FlutterBluePlus.stopScan()
  }

  /// 处理扫描结果回调
  void _onScanResult(String mac, String name, int rssi) {
    // 检查是否已发现过
    final existingIndex = _discoveredDevices.indexWhere(
      (d) => d.macAddress == mac,
    );

    if (existingIndex >= 0) {
      // 更新已有设备的RSSI
      _discoveredDevices[existingIndex].rssi = rssi;
    } else {
      // 添加新发现的设备
      _discoveredDevices.add(PadPenDevice(
        macAddress: mac,
        name: name,
        rssi: rssi,
      ));
    }

    // 按信号强度降序排列
    _discoveredDevices.sort((a, b) => b.rssi.compareTo(a.rssi));
    _scanController.add(List.from(_discoveredDevices));
  }

  /// 连接指定的点阵笔
  /// [device] 要连接的笔设备信息
  Future<bool> connectPen(PadPenDevice device) async {
    // 先断开已有连接
    if (_connectedPen != null) {
      await disconnectPen();
    }

    device.connectionState = PenConnectionState.connecting;
    _connectionController.add(PenConnectionState.connecting);
  }

```

```

try {
    // 停止扫描
    await stopScan();

    // 执行BLE连接
    // 实际调用: device.connect(timeout: Duration(seconds: 10))
    // 协商MTU
    // await device.requestMtu(PadBleConstants.requestedMtu);

    // 发现服务和特征值
    // final services = await device.discoverServices();
    // 查找笔迹数据特征值并订阅Notify

    // 设置连接成功状态
    device.connectionState = PenConnectionState.connected;
    _connectedPen = device;
    _reconnectAttempts = 0;
    _connectionController.add(PenConnectionState.connected);

    // 启动电量定时读取
    _startBatteryMonitor();

    // 订阅笔迹数据特征值
    _subscribeStrokeData();

    return true;
} catch (e) {
    device.connectionState = PenConnectionState.disconnected;
    _connectionController.add(PenConnectionState.disconnected);
    return false;
}
}

/// 订阅笔迹坐标数据Notify特征值
void _subscribeStrokeData() {
    // 实际调用:
    // characteristic.setNotifyValue(true);
    // characteristic.onValueReceived.listen(_onStrokeDataReceived);
}

/// 处理接收到的笔迹原始数据 (7字节紧凑编码)
/// 数据格式: [X_H, X_L, Y_H, Y_L, Pressure, TS_H, TS_L]
/// X: 16位无符号 (0.01mm精度)
/// Y: 16位无符号 (0.01mm精度)
/// Pressure: 8位无符号 (0-255)
/// Timestamp: 16位无符号 (相对毫秒)
void _onStrokeDataReceived(Uint8List rawData) {
    if (rawData.length < 7) return;

    // 可能包含多个坐标点 (每7字节一个)
    int offset = 0;
    while (offset + 7 <= rawData.length) {
        // 解码X坐标 (大端序16位)
        final int rawX = (rawData[offset] << 8) | rawData[offset + 1];
        final double x = rawX * 0.01; // 转换为毫米
    }
}

```

```

// 解码Y坐标
final int rawY = (rawData[offset + 2] << 8) | rawData[offset + 3];
final double y = rawY * 0.01;

// 解码压力值
final int pressure = rawData[offset + 4];

// 解码时间戳
final int timestamp =
    (rawData[offset + 5] << 8) | rawData[offset + 6];

// 判断落笔/抬笔（压力值>0为落笔）
final bool isPenDown = pressure > 0;

final point = PadPenPoint(
    x: x,
    y: y,
    pressure: pressure,
    timestamp: timestamp,
    isPenDown: isPenDown,
);

_strokeBuffer.add(point);
offset += 7;
}

// CRC-16 CCITT校验（如果数据包尾部有2字节CRC）
if (rawData.length > offset + 1) {
    final int receivedCrc = (rawData[offset] << 8) | rawData[offset + 1];
    final int calculatedCrc = _calculateCrc16(
        rawData.sublist(0, offset),
    );
    if (receivedCrc != calculatedCrc) {
        // CRC校验失败，丢弃本批数据
        _strokeBuffer.clear();
        return;
    }
}

// 达到批量阈值后回调
if (_strokeBuffer.length >= PadBleConstants.strokeBatchSize) {
    _flushStrokeBuffer();
}

}

/// 将缓冲区中的笔迹数据批量回调
void _flushStrokeBuffer() {
    if (_strokeBuffer.isEmpty || _connectedPen == null) return;

    final event = PenStrokeEvent(
        penMac: _connectedPen!.macAddress,
        points: List.from(_strokeBuffer),
        pageId: _connectedPen!.currentPageId,
    );

    _strokeController.add(event);
    _strokeBuffer.clear();
}

```

```

}

/// CRC-16 CCITT校验算法
/// 多项式: 0x1021, 初始值: 0xFFFF
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;
}

/// 启动电量定时读取
void _startBatteryMonitor() {
    _batteryTimer?.cancel();
    _batteryTimer = Timer.periodic(
        Duration(seconds: PadBleConstants.batteryReadInterval),
        (_) => _readBatteryLevel(),
    );
    // 立即读取一次
    _readBatteryLevel();
}

/// 读取笔电量
Future<void> _readBatteryLevel() async {
    if (_connectedPen == null) return;

    try {
        // 实际调用: 读取battery特征值
        // final value = await batteryCharacteristic.read();
        // _connectedPen!.batteryLevel = value[0];
        // _batteryController.add(_connectedPen!.batteryLevel);
    } catch (e) {
        // 读取失败, 忽略
    }
}

/// 向笔发送控制指令
/// [command] 指令类型 (如: LED闪烁、蜂鸣提示、固件信息查询)
Future<void> sendCommand(int command, [Uint8List? payload]) async {
    if (_connectedPen == null) return;

    // 构建指令包: [CMD, LEN, PAYLOAD..., CRC_H, CRC_L]
    final List<int> packet = [command];
    if (payload != null) {
        packet.add(payload.length);
        packet.addAll(payload);
    } else {
        packet.add(0);
    }
}

```

```

// 追加CRC校验
final crc = _calculateCrc16(Uint8List.fromList(packet));
packet.add((crc >> 8) & 0xFF);
packet.add(crc & 0xFF);

// 实际调用: controlCharacteristic.write(Uint8List.fromList(packet));
}

/// 断开当前笔连接
Future<void> disconnectPen() async {
  _batteryTimer?.cancel();
  _reconnectTimer?.cancel();

  if (_connectedPen != null) {
    _connectedPen!.connectionState = PenConnectionState.disconnecting;
    _connectionController.add(PenConnectionState.disconnecting);

    // 实际调用: device.disconnect();
    _connectedPen!.connectionState = PenConnectionState.disconnected;
    _connectedPen = null;
    _connectionController.add(PenConnectionState.disconnected);
  }

  // 清空缓冲区
  _flushStrokeBuffer();
}

/// 处理连接意外断开, 启动自动重连
void _onDisconnected(PadPenDevice device) {
  if (_reconnectAttempts >= PadBleConstants.maxReconnectAttempts) {
    // 超过最大重连次数, 放弃重连
    _connectionController.add(PenConnectionState.disconnected);
    return;
  }

  _connectionController.add(PenConnectionState.reconnecting);
  _reconnectAttempts++;

  // 指数退避延迟重连
  final delay = PadBleConstants.reconnectDelaySeconds * _reconnectAttempts;
  final clampedDelay = delay > 30 ? 30 : delay;

  _reconnectTimer = Timer(Duration(seconds: clampedDelay), () async {
    final success = await connectPen(device);
    if (!success) {
      _onDisconnected(device);
    }
  });
}

/// 释放所有资源
void dispose() {
  _batteryTimer?.cancel();
  _reconnectTimer?.cancel();
  _scanController.close();
  _strokeController.close();
}

```

```
        _connectionController.close();  
        _batteryController.close();  
        _strokeBuffer.clear();  
    }  
}
```