

自然写互动课堂PC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

```
08-writech-app-pc/
├── cast/
│   └── screen_cast.ts
├── database/
│   └── db_manager.ts
├── main/
│   ├── device_manager.ts
│   └── main.ts
├── renderer/
│   ├── api/
│   │   └── cloud_api.ts
│   ├── components/
│   │   └── StrokeCanvas.vue
│   └── store/
│       └── index.ts
```

源程序文件清单

cast/

cast/screen_cast.ts

```
/**
 * 自然写互动课堂PC端应用软件 V1.0
 * WebRTC投屏模块 - 实现PC端屏幕内容投射到智慧黑板/电视大屏
 *
 * 功能说明：
 * 1. WebRTC点对点连接建立（ICE候选收集、STUN/TURN穿透）
```

- * 2. 屏幕捕获与视频流编码 (desktopCapturer API)
- * 3. 自适应码率控制 (根据网络状况动态调整分辨率和帧率)
- * 4. 信令服务通信 (通过WebSocket交换SDP和ICE候选)
- * 5. 多目标同时投屏 (一个PC端可投射到多个大屏设备)
- * 6. 投屏区域选择 (全屏/窗口/自定义区域)
- * 7. 音频同步传输 (系统音频 + 麦克风输入混合)
- * 8. 投屏安全控制 (PIN码配对, 防止未授权投屏)

```

import { EventEmitter } from 'events';
import crypto from 'crypto';

/* ===== 类型定义 ===== */

/** 投屏目标设备信息 */
interface CastTarget {
  deviceId: string;          // 大屏设备唯一标识
  deviceName: string;        // 设备显示名称 (如"教室1号黑板")
  deviceType: 'board' | 'tv'; // 设备类型: 智慧黑板 / 电视
  ipAddress: string;         // 设备IP地址
  port: number;              // 信令端口
  status: 'discovered' | 'connecting' | 'connected' | 'disconnected';
  peerConnection: any;       // RTCPeerConnection实例
  lastPingTime: number;      // 最后心跳时间
}

/** 投屏配置参数 */
interface CastConfig {
  maxWidth: number;          // 最大投屏分辨率宽度
  maxHeight: number;         // 最大投屏分辨率高度
  maxFrameRate: number;      // 最大帧率
  minBitrate: number;        // 最低码率 (kbps)
  maxBitrate: number;        // 最高码率 (kbps)
  enableAudio: boolean;      // 是否传输音频
  captureMode: 'screen' | 'window' | 'region'; // 捕获模式
  stunServers: string[];     // STUN服务器列表
  turnServer: string;        // TURN中继服务器地址
  turnUsername: string;      // TURN认证用户名
  turnCredential: string;    // TURN认证密码
  signalServerUrl: string;    // 信令服务器WebSocket地址
  pinCode: string;           // 投屏PIN码 (4位数字)
}

/** 投屏质量统计 */
interface CastQualityStats {
  currentBitrate: number;    // 当前码率 (kbps)
  currentFps: number;        // 当前帧率
  packetLoss: number;        // 丢包率 (百分比)
  roundTripTime: number;     // 往返延迟 (毫秒)
  resolution: string;        // 当前分辨率
  encoderType: string;       // 编码器类型
  timestamp: number;
}

/** 信令消息格式 */
interface SignalMessage {
  type: 'offer' | 'answer' | 'candidate' | 'pin_verify' | 'cast_stop' |

```

```

'quality_adjust';
  fromDeviceId: string;
  toDeviceId: string;
  payload: any;
  timestamp: number;
  signature: string;          // HMAC-SHA256消息签名
}

/* ===== 投屏管理器 ===== */

// 默认投屏配置
const DEFAULT_CAST_CONFIG: CastConfig = {
  maxWidth: 1920,
  maxHeight: 1080,
  maxFrameRate: 30,
  minBitrate: 500,
  maxBitrate: 4000,
  enableAudio: true,
  captureMode: 'screen',
  stunServers: ['stun:stun.writech.com:3478'],
  turnServer: 'turn:turn.writech.com:3478',
  turnUsername: '',
  turnCredential: '',
  signalServerUrl: 'wss://signal.writech.com/cast',
  pinCode: ''
};

/**
 * 投屏管理器 - 管理WebRTC投屏的完整生命周期
 * 支持同时向多个大屏设备投射内容
 */
class ScreenCastManager extends EventEmitter {
  private config: CastConfig;
  private targets: Map<string, CastTarget> = new Map(); // 投屏目标设备列表
  private localStream: MediaStream | null = null; // 本地媒体流
  private signalSocket: WebSocket | null = null; // 信令WebSocket连接
  private localDeviceId: string; // 本机设备标识
  private statsTimers: Map<string, ReturnType<typeof setInterval>> = new Map();
  private qualityHistory: CastQualityStats[] = []; // 质量统计历史
  private isCapturing: boolean = false;
  private hmacKey: string; // 消息签名密钥

  constructor(config?: Partial<CastConfig>) {
    super();
    this.config = { ...DEFAULT_CAST_CONFIG, ...config };
    // 使用机器MAC地址+时间戳生成唯一设备标识
    this.localDeviceId = `pc_${crypto.randomBytes(4).toString('hex')}`;
    this.hmacKey = crypto.randomBytes(16).toString('hex');
  }

  /**
   * 初始化投屏管理器
   * 建立信令服务器连接, 准备接收设备发现消息
   */
  async initialize(): Promise<void> {
    try {
      await this.connectSignalServer();
    }
  }
}

```

```

        console.log('[ScreenCast] 投屏管理器初始化完成');
    } catch (error) {
        console.error('[ScreenCast] 初始化失败:', error);
        throw error;
    }
}

/**
 * 连接信令服务器（通过WebSocket交换SDP和ICE候选）
 * 支持断线自动重连（指数退避策略）
 */
private async connectSignalServer(): Promise<void> {
    return new Promise((resolve, reject) => {
        const url = `${this.config.signalServerUrl}?
deviceId=${this.localDeviceId}&type=pc`;
        this.signalSocket = new WebSocket(url);

        this.signalSocket.onopen = () => {
            console.log('[ScreenCast] 信令服务器连接成功');
            resolve();
        };

        this.signalSocket.onmessage = (event: MessageEvent) => {
            try {
                const message: SignalMessage = JSON.parse(event.data);
                this.handleSignalMessage(message);
            } catch (error) {
                console.error('[ScreenCast] 信令消息解析失败:', error);
            }
        };

        this.signalSocket.onclose = () => {
            console.warn('[ScreenCast] 信令连接断开, 5秒后重连');
            setTimeout(() => this.connectSignalServer(), 5000);
        };

        this.signalSocket.onerror = (error) => {
            console.error('[ScreenCast] 信令连接错误:', error);
            reject(error);
        };
    });
}

/**
 * 处理信令消息分发
 * 根据消息类型执行不同的操作（SDP交换/ICE候选/PIN验证等）
 */
private handleSignalMessage(message: SignalMessage): void {
    // 验证消息签名（防止篡改）
    if (message.signature && !this.verifyMessageSignature(message)) {
        console.warn('[ScreenCast] 消息签名验证失败, 丢弃:', message.type);
        return;
    }

    switch (message.type) {
        case 'answer':
            this.handleRemoteAnswer(message.fromDeviceId, message.payload);
    }
}

```

```

        break;
    case 'candidate':
        this.handleRemoteCandidate(message.fromDeviceId, message.payload);
        break;
    case 'pin_verify':
        this.handlePinVerifyResult(message.fromDeviceId, message.payload);
        break;
    case 'quality_adjust':
        this.handleQualityAdjust(message.fromDeviceId, message.payload);
        break;
    case 'cast_stop':
        this.handleRemoteStop(message.fromDeviceId);
        break;
    default:
        console.warn('[ScreenCast] 未知信令类型:', message.type);
    }
}

/**
 * 开始屏幕捕获 - 使用Electron desktopCapturer API获取屏幕视频流
 * 支持全屏、窗口、自定义区域三种捕获模式
 */
async startCapture(sourceId?: string): Promise<void> {
    if (this.isCapturing) {
        console.warn('[ScreenCast] 已在投屏中, 请先停止当前投屏');
        return;
    }

    try {
        // 通过Electron desktopCapturer获取可用的屏幕/窗口源
        const { desktopCapturer } = require('electron');
        const sources = await desktopCapturer.getSources({
            types: this.config.captureMode === 'window' ? ['window'] : ['screen'],
            thumbnailSize: { width: 320, height: 180 }
        });

        if (sources.length === 0) {
            throw new Error('未找到可用的屏幕源');
        }

        // 选择屏幕源 (默认使用第一个或指定的源)
        const selectedSource = sourceId
            ? sources.find((s: any) => s.id === sourceId) || sources[0]
            : sources[0];

        // 配置视频约束参数
        const videoConstraints: any = {
            mandatory: {
                chromeMediaSource: 'desktop',
                chromeMediaSourceId: selectedSource.id,
                maxWidth: this.config.maxWidth,
                maxHeight: this.config.maxHeight,
                maxFrameRate: this.config.maxFrameRate,
                minFrameRate: 15
            }
        };
    }
};

```

```

// 获取媒体流 (视频 + 可选音频)
const stream = await (navigator.mediaDevices as any).getUserMedia({
  video: videoConstraints,
  audio: this.config.enableAudio ? {
    mandatory: { chromeMediaSource: 'desktop' }
  } : false
});

this.localStream = stream;
this.isCapturing = true;
this.emit('captureStarted', { sourceId: selectedSource.id, name:
selectedSource.name });
  console.log('[ScreenCast] 屏幕捕获已启动:', selectedSource.name);
} catch (error) {
  console.error('[ScreenCast] 屏幕捕获失败:', error);
  throw error;
}
}

/**
 * 向指定大屏设备发起投屏连接
 * 创建RTCPeerConnection, 添加本地流, 发送SDP Offer
 */
async castToDevice(deviceId: string, deviceName: string, ipAddress: string, port:
number): Promise<void> {
  if (!this.localStream) {
    throw new Error('请先启动屏幕捕获');
  }

  // 创建投屏目标记录
  const target: CastTarget = {
    deviceId, deviceName,
    deviceType: 'board',
    ipAddress, port,
    status: 'connecting',
    peerConnection: null,
    lastPingTime: Date.now()
  };

  // 配置ICE服务器 (STUN + TURN)
  const iceConfig: RTCCConfiguration = {
    iceServers: [
      { urls: this.config.stunServers },
      {
        urls: this.config.turnServer,
        username: this.config.turnUsername,
        credential: this.config.turnCredential
      }
    ],
    iceCandidatePoolSize: 10
  };

  // 创建RTCPeerConnection
  const pc = new RTCPeerConnection(iceConfig);
  target.peerConnection = pc;

  // 添加本地媒体流的所有轨道

```

```

this.localStream.getTracks().forEach(track => {
  pc.addTrack(track, this.localStream!);
});

// 配置视频编码参数 (优先使用H.264 High Profile)
const sender = pc.getSenders().find(s => s.track?.kind === 'video');
if (sender) {
  const params = sender.getParameters();
  if (params.encodings && params.encodings.length > 0) {
    params.encodings[0].maxBitrate = this.config.maxBitrate * 1000;
    params.encodings[0].maxFramerate = this.config.maxFrameRate;
    await sender.setParameters(params);
  }
}

// 监听ICE候选事件, 发送给对端
pc.onicecandidate = (event) => {
  if (event.candidate) {
    this.sendSignalMessage({
      type: 'candidate',
      fromDeviceId: this.localDeviceId,
      toDeviceId: deviceId,
      payload: event.candidate.toJSON(),
      timestamp: Date.now(),
      signature: ''
    });
  }
};

// 监听连接状态变化
pc.onconnectionstatechange = () => {
  console.log(`[ScreenCast] 连接状态[${deviceId}]:`, pc.connectionState);
  switch (pc.connectionState) {
    case 'connected':
      target.status = 'connected';
      this.startQualityMonitor(deviceId);
      this.emit('deviceConnected', { deviceId, deviceName });
      break;
    case 'disconnected':
    case 'failed':
      target.status = 'disconnected';
      this.stopQualityMonitor(deviceId);
      this.emit('deviceDisconnected', { deviceId, deviceName });
      break;
  }
};

// 创建并发送SDP Offer
const offer = await pc.createOffer({
  offerToReceiveAudio: false,
  offerToReceiveVideo: false
});
await pc.setLocalDescription(offer);

// 通过信令服务器发送Offer给大屏设备
this.sendSignalMessage({
  type: 'offer',

```

```

        fromDeviceId: this.localDeviceId,
        toDeviceId: deviceId,
        payload: { sdp: offer.sdp, type: offer.type, pinCode: this.config.pinCode },
        timestamp: Date.now(),
        signature: ''
    });

    this.targets.set(deviceId, target);
    console.log(`[ScreenCast] 已向 ${deviceId} 发起投屏请求`);
}

/** 处理远端设备的SDP Answer */
private async handleRemoteAnswer(deviceId: string, payload: any): Promise<void> {
    const target = this.targets.get(deviceId);
    if (!target || !target.peerConnection) return;

    try {
        const answer = new RTCSessionDescription(payload);
        await target.peerConnection.setRemoteDescription(answer);
        console.log(`[ScreenCast] 收到 ${target.deviceId} 的Answer`);
    } catch (error) {
        console.error(`[ScreenCast] 设置RemoteDescription失败:`, error);
    }
}

/** 处理远端ICE候选 */
private async handleRemoteCandidate(deviceId: string, payload: any): Promise<void> {
    const target = this.targets.get(deviceId);
    if (!target || !target.peerConnection) return;

    try {
        const candidate = new RTCIceCandidate(payload);
        await target.peerConnection.addIceCandidate(candidate);
    } catch (error) {
        console.error(`[ScreenCast] 添加ICE候选失败:`, error);
    }
}

/** 处理PIN码验证结果 */
private handlePinVerifyResult(deviceId: string, payload: { verified: boolean }):
void {
    if (!payload.verified) {
        console.warn(`[ScreenCast] 设备 ${deviceId} PIN码验证失败`);
        this.disconnectDevice(deviceId);
        this.emit('pinVerifyFailed', { deviceId });
    }
}

/** 处理远端质量调整请求（大屏端网络差时要求降低码率） */
private handleQualityAdjust(deviceId: string, payload: { maxBitrate?: number;
maxFps?: number }): void {
    const target = this.targets.get(deviceId);
    if (!target || !target.peerConnection) return;

    const sender = target.peerConnection.getSenders().find((s: any) => s.track?.kind
=== 'video');
    if (sender) {

```



```

        const params = sender.getParameters();
        if (params.encodings && params.encodings.length > 0) {
            if (payload.maxBitrate) {
                params.encodings[0].maxBitrate = payload.maxBitrate * 1000;
            }
            if (payload.maxFps) {
                params.encodings[0].maxFramerate = payload.maxFps;
            }
            sender.setParameters(params);
            console.log(`[ScreenCast] 已调整投屏质量: 码率=${payload.maxBitrate}kbps,
帧率=${payload.maxFps}fps`);
        }
    }
}

/** 处理远端停止投屏请求 */
private handleRemoteStop(deviceId: string): void {
    console.log(`[ScreenCast] 收到远端停止请求: ${deviceId}`);
    this.disconnectDevice(deviceId);
}

/**
 * 启动投屏质量监控
 * 每3秒采集一次WebRTC连接统计信息
 */
private startQualityMonitor(deviceId: string): void {
    const timer = setInterval(async () => {
        const target = this.targets.get(deviceId);
        if (!target || !target.peerConnection) {
            this.stopQualityMonitor(deviceId);
            return;
        }

        try {
            const stats = await target.peerConnection.getStats();
            let qualityStats: CastQualityStats = {
                currentBitrate: 0, currentFps: 0,
                packetLoss: 0, roundTripTime: 0,
                resolution: '', encoderType: '',
                timestamp: Date.now()
            };

            stats.forEach((report: any) => {
                if (report.type === 'outbound-rtp' && report.kind === 'video') {
                    qualityStats.currentBitrate = Math.round((report.bytesSent * 8)
/ 1000);

                    qualityStats.currentFps = report.framesPerSecond || 0;
                    qualityStats.resolution =
`${report.frameWidth}x${report.frameHeight}`;
                    qualityStats.encoderType = report.encoderImplementation ||
'unknown';
                }
                if (report.type === 'candidate-pair' && report.state ===
'succeeded') {
                    qualityStats.roundTripTime = report.currentRoundTripTime * 1000;
                }
                if (report.type === 'remote-inbound-rtp') {

```

```

        qualityStats.packetLoss = report.fractionLost * 100;
    }
});

// 保存统计历史 (最多保留1000条)
this.qualityHistory.push(qualityStats);
if (this.qualityHistory.length > 1000) {
    this.qualityHistory.splice(0, this.qualityHistory.length - 1000);
}

// 自适应码率控制: 丢包率过高时自动降低码率
if (qualityStats.packetLoss > 5) {
    const reducedBitrate = Math.max(
        this.config.minBitrate,
        qualityStats.currentBitrate * 0.7
    );
    this.adjustBitrate(deviceId, reducedBitrate);
} else if (qualityStats.packetLoss < 1 && qualityStats.currentBitrate <
this.config.maxBitrate) {
    // 网络状况良好时逐步提高码率
    const increasedBitrate = Math.min(
        this.config.maxBitrate,
        qualityStats.currentBitrate * 1.1
    );
    this.adjustBitrate(deviceId, increasedBitrate);
}

    this.emit('qualityUpdate', { deviceId, stats: qualityStats });
} catch (error) {
    console.error('[ScreenCast] 质量监控统计失败:', error);
}
}, 3000);

    this.statsTimers.set(deviceId, timer);
}

/** 停止质量监控 */
private stopQualityMonitor(deviceId: string): void {
    const timer = this.statsTimers.get(deviceId);
    if (timer) {
        clearInterval(timer);
        this.statsTimers.delete(deviceId);
    }
}

/** 动态调整视频码率 */
private adjustBitrate(deviceId: string, targetBitrate: number): void {
    const target = this.targets.get(deviceId);
    if (!target || !target.peerConnection) return;

    const sender = target.peerConnection.getSenders().find((s: any) => s.track?.kind
=== 'video');
    if (sender) {
        const params = sender.getParameters();
        if (params.encodings && params.encodings.length > 0) {
            params.encodings[0].maxBitrate = Math.round(targetBitrate * 1000);
            sender.setParameters(params).catch((e: Error) => {

```

```

        console.error('[ScreenCast] 码率调整失败:', e.message);
    });
}
}

/** 断开指定设备的投屏连接 */
disconnectDevice(deviceId: string): void {
    const target = this.targets.get(deviceId);
    if (!target) return;

    // 关闭PeerConnection
    if (target.peerConnection) {
        target.peerConnection.close();
    }

    // 停止质量监控
    this.stopQualityMonitor(deviceId);

    // 通知对端
    this.sendMessage({
        type: 'cast_stop',
        fromDeviceId: this.localDeviceId,
        toDeviceId: deviceId,
        payload: {},
        timestamp: Date.now(),
        signature: ''
    });

    this.targets.delete(deviceId);
    this.emit('deviceDisconnected', { deviceId, deviceName: target.deviceName });
    console.log(`[ScreenCast] 已断开投屏: ${target.deviceName}`);
}

/** 停止所有投屏并释放资源 */
stopAllCasting(): void {
    // 断开所有投屏目标
    for (const deviceId of this.targets.keys()) {
        this.disconnectDevice(deviceId);
    }

    // 停止屏幕捕获
    if (this.localStream) {
        this.localStream.getTracks().forEach(track => track.stop());
        this.localStream = null;
    }
    this.isCapturing = false;

    this.emit('allCastingStopped');
    console.log('[ScreenCast] 所有投屏已停止');
}

/** 发送信令消息（附加HMAC-SHA256签名） */
private sendMessage(message: SignalMessage): void {
    // 生成消息签名，防止信令被篡改
    const content =
` ${message.type}: ${message.fromDeviceId}: ${message.toDeviceId}: ${message.timestamp}`;

```

```

        message.signature = crypto.createHmac('sha256',
this.hmacKey).update(content).digest('hex');

        if (this.signalSocket && this.signalSocket.readyState === WebSocket.OPEN) {
            this.signalSocket.send(JSON.stringify(message));
        } else {
            console.warn('[ScreenCast] 信令连接不可用, 消息发送失败');
        }
    }

    /** 验证收到的信令消息签名 */
    private verifyMessageSignature(message: SignalMessage): boolean {
        const content =
`${message.type}:${message.fromDeviceId}:${message.toDeviceId}:${message.timestamp}`;
        const expected = crypto.createHmac('sha256',
this.hmacKey).update(content).digest('hex');
        return message.signature === expected;
    }

    /** 获取当前投屏状态汇总 */
    getStatus(): { isCapturing: boolean; connectedDevices: number; targets: any[] } {
        const targetList = Array.from(this.targets.values()).map(t => ({
            deviceId: t.deviceId,
            deviceName: t.deviceName,
            status: t.status,
            deviceType: t.deviceType
        }));
        return {
            isCapturing: this.isCapturing,
            connectedDevices: targetList.filter(t => t.status === 'connected').length,
            targets: targetList
        };
    }

    /** 销毁投屏管理器, 释放所有资源 */
    destroy(): void {
        this.stopAllCasting();
        if (this.signalSocket) {
            this.signalSocket.close();
            this.signalSocket = null;
        }
        this.qualityHistory = [];
        this.removeAllListeners();
        console.log('[ScreenCast] 投屏管理器已销毁');
    }
}

export default ScreenCastManager;

```

database/

database/db_manager.ts

```

/**
 * 自然写互动课堂PC端应用软件 V1.0
 * 数据库管理模块 - 基于better-sqlite3实现SQLite本地数据持久化
 *
 * 功能说明:
 * 1. 数据库初始化与版本迁移 (Schema Migration)
 * 2. 学生笔迹数据的存储与检索 (支持按学生/作业/时间维度查询)
 * 3. 作业批改记录管理 (AI批改 + 人工标注)
 * 4. 班级/学生信息本地缓存 (减少网络请求)
 * 5. 点阵码映射关系维护 (课件页面与点阵码对应)
 * 6. 课件元数据索引 (本地课件文件的管理信息)
 * 7. 数据库文件加密 (SQLCipher集成, 防止本地数据泄露)
 * 8. 自动备份与数据清理策略
 */

import path from 'path';
import fs from 'fs';
import { app } from 'electron';
import crypto from 'crypto';

/* ===== 类型定义 ===== */

/** 数据库配置接口 */
interface DatabaseConfig {
  dbPath: string; // 数据库文件路径
  encryptionKey: string; // 加密密钥 (SQLCipher)
  maxBackups: number; // 最大备份数量
  autoVacuumInterval: number; // 自动整理间隔 (毫秒)
  walMode: boolean; // 是否启用WAL模式
}

/** 学生笔迹记录 */
interface StrokeRecord {
  id: string;
  studentId: string;
  studentName: string;
  assignmentId: string;
  pageIndex: number;
  strokeData: string; // JSON序列化的笔迹坐标数据
  thumbnailPath: string; // 缩略图文件路径
  collectTime: number; // 采集时间戳
  syncStatus: number; // 同步状态: 0=未同步, 1=已同步, 2=同步失败
  fileSize: number; // 数据大小 (字节)
}

/** 批改记录 */
interface GradeRecord {
  id: string;
  assignmentId: string;
  studentId: string;
  aiScore: number; // AI评分 (0-100)
  teacherScore: number; // 教师评分 (-1表示未批改)
  aiAnnotation: string; // AI批改标注JSON
  teacherAnnotation: string; // 教师手动标注JSON
  gradeTime: number;
  status: number; // 0=待批改, 1=AI已批, 2=教师已批
}

```

```

}

/** 班级信息 */
interface ClassInfo {
  classId: string;
  className: string;
  grade: string;
  teacherId: string;
  studentCount: number;
  lastSyncTime: number;
}

/** 学生信息 */
interface StudentInfo {
  studentId: string;
  studentName: string;
  classId: string;
  seatNumber: number;
  penDeviceId: string;      // 绑定的点阵笔设备ID
  avatarPath: string;
}

/** 点阵码映射 */
interface DotCodeMapping {
  dotCodeId: string;        // 点阵码唯一标识
  coursewareId: string;     // 课件ID
  pageIndex: number;        // 对应页面索引
  regionType: string;       // 区域类型: 'answer'/'writing'/'drawing'
  coordinates: string;      // 区域坐标JSON
}

/** 课件元数据 */
interface CoursewareMeta {
  coursewareId: string;
  title: string;
  type: string;              // 'ppt'/'pdf'/'custom'
  filePath: string;          // 本地文件路径
  pageCount: number;
  fileSize: number;
  createTime: number;
  lastOpenTime: number;
  cloudUrl: string;          // 云端地址
  syncStatus: number;
}

/** 迁移脚本定义 */
interface Migration {
  version: number;
  description: string;
  sql: string;
}

/* ===== 数据库管理器 ===== */

// 数据库Schema版本号, 每次表结构变更递增
const CURRENT_SCHEMA_VERSION = 5;

```

```

/**
 * 数据库管理器 - 统一管理SQLite数据库的生命周期
 * 采用单例模式确保全局唯一数据库连接
 */
class DatabaseManager {
  private db: any = null; // better-sqlite3 数据库实例
  private config: DatabaseConfig; // 数据库配置
  private backupTimer: ReturnType<typeof setInterval> | null = null;
  private vacuumTimer: ReturnType<typeof setInterval> | null = null;
  private initialized: boolean = false;

  constructor() {
    // 默认配置: 数据库存储在应用数据目录
    const userDataPath = app.getPath('userData');
    this.config = {
      dbPath: path.join(userDataPath, 'writech_data.db'),
      encryptionKey: this.loadOrCreateEncryptionKey(),
      maxBackups: 5,
      autoVacuumInterval: 24 * 60 * 60 * 1000, // 每24小时整理一次
      walMode: true
    };
  }

  /**
   * 加载或创建数据库加密密钥
   * 密钥存储在操作系统安全凭据管理器中 (通过keytar)
   * 首次运行时生成随机256位密钥
   */
  private loadOrCreateEncryptionKey(): string {
    const keyFilePath = path.join(app.getPath('userData'), '.db_key');
    try {
      if (fs.existsSync(keyFilePath)) {
        return fs.readFileSync(keyFilePath, 'utf-8').trim();
      }
      // 生成256位随机密钥并保存
      const newKey = crypto.randomBytes(32).toString('hex');
      fs.writeFileSync(keyFilePath, newKey, { mode: 0o600 });
      console.log('[DatabaseManager] 已生成新的数据库加密密钥');
      return newKey;
    } catch (error) {
      console.error('[DatabaseManager] 密钥管理失败, 使用默认密钥:', error);
      return 'writech_default_key_2024';
    }
  }

  /**
   * 初始化数据库连接并执行迁移
   * 启用WAL模式提高并发读写性能
   * 设置SQLCipher加密密钥
   */
  async initialize(): Promise<void> {
    if (this.initialized) return;

    try {
      const Database = require('better-sqlite3');
      const dbDir = path.dirname(this.config.dbPath);
      if (!fs.existsSync(dbDir)) {

```

```

        fs.mkdirSync(dbDir, { recursive: true });
    }

    // 创建数据库连接（启用verbose日志用于调试）
    this.db = new Database(this.config.dbPath, { verbose: undefined });

    // 设置SQLCipher加密密钥
    this.db.pragma(`key='${this.config.encryptionKey}'`);

    // 启用WAL模式提高并发性能
    if (this.config.walMode) {
        this.db.pragma('journal_mode=WAL');
        this.db.pragma('synchronous=NORMAL');
    }

    // 启用外键约束
    this.db.pragma('foreign_keys=ON');

    // 执行数据库迁移
    this.runMigrations();

    // 启动定时任务（备份 + 整理）
    this.startAutoBackup();
    this.startAutoVacuum();

    this.initialized = true;
    console.log('[DatabaseManager] 数据库初始化完成，版本:',
CURRENT_SCHEMA_VERSION);
    } catch (error) {
        console.error('[DatabaseManager] 数据库初始化失败:', error);
        throw error;
    }
}

/**
 * 获取所有迁移脚本列表
 * 每个版本对应一个迁移脚本，按版本号顺序执行
 */
private getMigrations(): Migration[] {
    return [
        {
            version: 1,
            description: '创建基础表结构',
            sql: `
                -- 学生笔迹数据表
                CREATE TABLE IF NOT EXISTS stroke_records (
                    id TEXT PRIMARY KEY,
                    student_id TEXT NOT NULL,
                    student_name TEXT NOT NULL,
                    assignment_id TEXT NOT NULL,
                    page_index INTEGER DEFAULT 0,
                    stroke_data TEXT NOT NULL,
                    thumbnail_path TEXT DEFAULT '',
                    collect_time INTEGER NOT NULL,
                    sync_status INTEGER DEFAULT 0,
                    file_size INTEGER DEFAULT 0,
                    created_at INTEGER DEFAULT (strftime('%s','now'))
            `
        }
    ]
}

```



```

    );
    CREATE INDEX IF NOT EXISTS idx_stroke_student ON
stroke_records(student_id);
    CREATE INDEX IF NOT EXISTS idx_stroke_assignment ON
stroke_records(assignment_id);
    CREATE INDEX IF NOT EXISTS idx_stroke_time ON
stroke_records(collect_time);

-- 批改记录表
CREATE TABLE IF NOT EXISTS grade_records (
    id TEXT PRIMARY KEY,
    assignment_id TEXT NOT NULL,
    student_id TEXT NOT NULL,
    ai_score REAL DEFAULT -1,
    teacher_score REAL DEFAULT -1,
    ai_annotation TEXT DEFAULT '{}',
    teacher_annotation TEXT DEFAULT '{}',
    grade_time INTEGER NOT NULL,
    status INTEGER DEFAULT 0,
    created_at INTEGER DEFAULT (strftime('%s','now'))
);
    CREATE INDEX IF NOT EXISTS idx_grade_assignment ON
grade_records(assignment_id);
    CREATE INDEX IF NOT EXISTS idx_grade_student ON
grade_records(student_id);
    \
},
{
    version: 2,
    description: '添加班级和学生信息表',
    sql: `
-- 班级信息缓存表
CREATE TABLE IF NOT EXISTS class_info (
    class_id TEXT PRIMARY KEY,
    class_name TEXT NOT NULL,
    grade TEXT DEFAULT '',
    teacher_id TEXT NOT NULL,
    student_count INTEGER DEFAULT 0,
    last_sync_time INTEGER DEFAULT 0
);

-- 学生信息缓存表
CREATE TABLE IF NOT EXISTS student_info (
    student_id TEXT PRIMARY KEY,
    student_name TEXT NOT NULL,
    class_id TEXT NOT NULL,
    seat_number INTEGER DEFAULT 0,
    pen_device_id TEXT DEFAULT '',
    avatar_path TEXT DEFAULT '',
    FOREIGN KEY (class_id) REFERENCES class_info(class_id)
);
    CREATE INDEX IF NOT EXISTS idx_student_class ON
student_info(class_id);
    CREATE INDEX IF NOT EXISTS idx_student_pen ON
student_info(pen_device_id);
    \
},

```

```

{
  version: 3,
  description: '添加点阵码映射表',
  sql: `
    -- 点阵码映射关系表（课件页面与点阵码ID对应）
    CREATE TABLE IF NOT EXISTS dot_code_mapping (
      dot_code_id TEXT PRIMARY KEY,
      courseware_id TEXT NOT NULL,
      page_index INTEGER NOT NULL,
      region_type TEXT DEFAULT 'answer',
      coordinates TEXT DEFAULT '{}',
      created_at INTEGER DEFAULT (strftime('%s','now'))
    );
    CREATE INDEX IF NOT EXISTS idx_dotcode_courseware ON
dot_code_mapping(courseware_id);
  `,
},
{
  version: 4,
  description: '添加课件元数据表',
  sql: `
    -- 课件元数据索引表
    CREATE TABLE IF NOT EXISTS courseware_meta (
      courseware_id TEXT PRIMARY KEY,
      title TEXT NOT NULL,
      type TEXT DEFAULT 'custom',
      file_path TEXT NOT NULL,
      page_count INTEGER DEFAULT 0,
      file_size INTEGER DEFAULT 0,
      create_time INTEGER NOT NULL,
      last_open_time INTEGER DEFAULT 0,
      cloud_url TEXT DEFAULT '',
      sync_status INTEGER DEFAULT 0
    );
    CREATE INDEX IF NOT EXISTS idx_courseware_type ON
courseware_meta(type);
    CREATE INDEX IF NOT EXISTS idx_courseware_time ON
courseware_meta(last_open_time);
  `,
},
{
  version: 5,
  description: '添加同步日志表用于离线数据追踪',
  sql: `
    -- 数据同步日志表（记录所有待同步操作）
    CREATE TABLE IF NOT EXISTS sync_log (
      id INTEGER PRIMARY KEY AUTOINCREMENT,
      table_name TEXT NOT NULL,
      record_id TEXT NOT NULL,
      operation TEXT NOT NULL,
      payload TEXT DEFAULT '{}',
      sync_status INTEGER DEFAULT 0,
      retry_count INTEGER DEFAULT 0,
      created_at INTEGER DEFAULT (strftime('%s','now')),
      synced_at INTEGER DEFAULT 0
    );
    CREATE INDEX IF NOT EXISTS idx_sync_status ON sync_log(sync_status);
  `
}

```

```

    }
  };
}

/**
 * 执行数据库迁移
 * 检查当前版本号，依次执行未执行的迁移脚本
 * 使用事务确保迁移的原子性
 */
private runMigrations(): void {
  // 创建版本跟踪表
  this.db.exec(`
    CREATE TABLE IF NOT EXISTS schema_version (
      version INTEGER PRIMARY KEY,
      description TEXT,
      applied_at INTEGER DEFAULT (strftime('%s','now'))
    );
  `);

  // 获取当前数据库版本
  const row = this.db.prepare('SELECT MAX(version) as ver FROM
schema_version').get();
  const currentVersion = row?.ver || 0;

  if (currentVersion >= CURRENT_SCHEMA_VERSION) {
    console.log('[DatabaseManager] 数据库已是最新版本:', currentVersion);
    return;
  }

  // 获取待执行的迁移脚本并按版本排序执行
  const migrations = this.getMigrations().filter(m => m.version > currentVersion);
  const runAll = this.db.transaction(() => {
    for (const migration of migrations) {
      console.log(`[DatabaseManager] 执行迁移 v${migration.version}:
${migration.description}`);
      this.db.exec(migration.sql);
      this.db.prepare('INSERT INTO schema_version (version, description)
VALUES (?, ?)')
        .run(migration.version, migration.description);
    }
  });

  runAll();
  console.log(`[DatabaseManager] 迁移完成: v${currentVersion} ->
v${CURRENT_SCHEMA_VERSION}`);
}

/* ===== 笔迹数据操作 ===== */

/** 保存学生笔迹记录（批量插入，提高写入性能） */
saveStrokeRecords(records: StrokeRecord[]): number {
  const insertStmt = this.db.prepare(`
    INSERT OR REPLACE INTO stroke_records
    (id, student_id, student_name, assignment_id, page_index,
    stroke_data, thumbnail_path, collect_time, sync_status, file_size)
    VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
  `);

```

```

`);

// 使用事务批量插入，避免逐条写入导致的性能问题
const insertMany = this.db.transaction((items: StrokeRecord[]) => {
  let count = 0;
  for (const r of items) {
    insertStmt.run(
      r.id, r.studentId, r.studentName, r.assignmentId,
      r.pageIndex, r.strokeData, r.thumbnailPath,
      r.collectTime, r.syncStatus, r.fileSize
    );
    count++;
  }
  // 同时记录同步日志
  const logStmt = this.db.prepare(`
    INSERT INTO sync_log (table_name, record_id, operation, payload)
    VALUES ('stroke_records', ?, 'INSERT', ?)
  `);
  for (const r of items) {
    logStmt.run(r.id, JSON.stringify({ assignmentId: r.assignmentId }));
  }
  return count;
});

return insertMany(records);
}

/** 按作业ID查询笔迹（支持分页） */
getStrokesByAssignment(assignmentId: string, page: number = 0, pageSize: number =
50): StrokeRecord[] {
  const offset = page * pageSize;
  return this.db.prepare(`
    SELECT id, student_id as studentId, student_name as studentName,
      assignment_id as assignmentId, page_index as pageIndex,
      stroke_data as strokeData, thumbnail_path as thumbnailPath,
      collect_time as collectTime, sync_status as syncStatus,
      file_size as fileSize
    FROM stroke_records
    WHERE assignment_id = ?
    ORDER BY collect_time DESC
    LIMIT ? OFFSET ?
  `).all(assignmentId, pageSize, offset);
}

/** 查询某学生的所有笔迹（用于学情分析） */
getStrokesByStudent(studentId: string, startTime?: number, endTime?: number):
StrokeRecord[] {
  let sql = `SELECT * FROM stroke_records WHERE student_id = ?`;
  const params: any[] = [studentId];
  if (startTime) {
    sql += ' AND collect_time >= ?';
    params.push(startTime);
  }
  if (endTime) {
    sql += ' AND collect_time <= ?';
    params.push(endTime);
  }
}

```

```

        sql += ' ORDER BY collect_time DESC';
        return this.db.prepare(sql).all(...params);
    }

    /** 获取未同步的笔迹记录（用于断网重连后批量上传） */
    getUnsyncedStrokes(limit: number = 100): StrokeRecord[] {
        return this.db.prepare(`
            SELECT * FROM stroke_records
            WHERE sync_status = 0
            ORDER BY collect_time ASC
            LIMIT ?
        `).all(limit);
    }

    /** 批量更新笔迹同步状态 */
    updateStrokeSyncStatus(ids: string[], status: number): void {
        const placeholders = ids.map(() => '?').join(',');
        this.db.prepare(`
            UPDATE stroke_records SET sync_status = ?
            WHERE id IN (${placeholders})
        `).run(status, ...ids);
    }

    /** ===== 批改记录操作 ===== */

    /** 保存或更新批改记录 */
    saveGradeRecord(record: GradeRecord): void {
        this.db.prepare(`
            INSERT OR REPLACE INTO grade_records
            (id, assignment_id, student_id, ai_score, teacher_score,
             ai_annotation, teacher_annotation, grade_time, status)
            VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)
        `).run(
            record.id, record.assignmentId, record.studentId,
            record.aiScore, record.teacherScore,
            record.aiAnnotation, record.teacherAnnotation,
            record.gradeTime, record.status
        );
    }

    /** 查询作业的批改结果列表 */
    getGradesByAssignment(assignmentId: string): GradeRecord[] {
        return this.db.prepare(`
            SELECT g.*, s.student_name as studentName
            FROM grade_records g
            LEFT JOIN student_info s ON g.student_id = s.student_id
            WHERE g.assignment_id = ?
            ORDER BY g.grade_time DESC
        `).all(assignmentId);
    }

    /** 获取待教师批改的记录数 */
    getPendingGradeCount(): number {
        const row = this.db.prepare(`
            SELECT COUNT(*) as cnt FROM grade_records WHERE status < 2
        `).get();
        return row?.cnt || 0;
    }

```

```

}

/* ===== 班级/学生信息操作 ===== */

/** 批量同步班级信息（从云端拉取后缓存到本地） */
syncClassInfo(classes: ClassInfo[]): void {
  const upsert = this.db.prepare(`
    INSERT OR REPLACE INTO class_info
    (class_id, class_name, grade, teacher_id, student_count, last_sync_time)
    VALUES (?, ?, ?, ?, ?, ?)
  `);
  const syncAll = this.db.transaction((items: ClassInfo[]) => {
    for (const c of items) {
      upsert.run(c.classId, c.className, c.grade, c.teacherId, c.studentCount,
Date.now());
    }
  });
  syncAll(classes);
}

/** 批量同步学生信息 */
syncStudentInfo(students: StudentInfo[]): void {
  const upsert = this.db.prepare(`
    INSERT OR REPLACE INTO student_info
    (student_id, student_name, class_id, seat_number, pen_device_id,
avatar_path)
    VALUES (?, ?, ?, ?, ?, ?)
  `);
  const syncAll = this.db.transaction((items: StudentInfo[]) => {
    for (const s of items) {
      upsert.run(s.studentId, s.studentName, s.classId, s.seatNumber,
s.penDeviceId, s.avatarPath);
    }
  });
  syncAll(students);
}

/** 按班级查询学生列表 */
getStudentsByClass(classId: string): StudentInfo[] {
  return this.db.prepare(`
    SELECT * FROM student_info WHERE class_id = ? ORDER BY seat_number
  `).all(classId);
}

/** 通过点阵笔设备ID查找学生（用于实时笔迹识别） */
findStudentByPenDevice(penDeviceId: string): StudentInfo | undefined {
  return this.db.prepare(`
    SELECT * FROM student_info WHERE pen_device_id = ?
  `).get(penDeviceId);
}

/* ===== 点阵码映射操作 ===== */

/** 保存点阵码映射关系 */
saveDotCodeMappings(mappings: DotCodeMapping[]): void {
  const upsert = this.db.prepare(`
    INSERT OR REPLACE INTO dot_code_mapping

```

```

        (dot_code_id, courseware_id, page_index, region_type, coordinates)
        VALUES (?, ?, ?, ?, ?)
    `);
    const saveAll = this.db.transaction((items: DotCodeMapping[]) => {
        for (const m of items) {
            upsert.run(m.dotCodeId, m.coursewareId, m.pageIndex, m.regionType,
m.coordinates);
        }
    });
    saveAll(mappings);
}

/** 根据点阵码ID查找对应的课件页面（笔迹数据落点定位） */
findPageByDotCode(dotCodeId: string): DotCodeMapping | undefined {
    return this.db.prepare(`
        SELECT * FROM dot_code_mapping WHERE dot_code_id = ?
    `).get(dotCodeId);
}

/* ===== 课件元数据操作 ===== */

/** 保存课件元数据 */
saveCoursewareMeta(meta: CoursewareMeta): void {
    this.db.prepare(`
        INSERT OR REPLACE INTO courseware_meta
        (courseware_id, title, type, file_path, page_count, file_size,
        create_time, last_open_time, cloud_url, sync_status)
        VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
    `).run(
        meta.coursewareId, meta.title, meta.type, meta.filePath,
        meta.pageCount, meta.fileSize, meta.createTime,
        meta.lastOpenTime, meta.cloudUrl, meta.syncStatus
    );
}

/** 获取最近打开的课件列表 */
getRecentCoursewares(limit: number = 20): CoursewareMeta[] {
    return this.db.prepare(`
        SELECT * FROM courseware_meta ORDER BY last_open_time DESC LIMIT ?
    `).all(limit);
}

/* ===== 数据库维护操作 ===== */

/** 启动自动备份定时器（每6小时备份一次） */
private startAutoBackup(): void {
    const BACKUP_INTERVAL = 6 * 60 * 60 * 1000; // 6小时
    this.backupTimer = setInterval(() => {
        this.createBackup();
    }, BACKUP_INTERVAL);
}

/** 创建数据库备份文件 */
createBackup(): string {
    const backupDir = path.join(path.dirname(this.config.dbPath), 'backups');
    if (!fs.existsSync(backupDir)) {
        fs.mkdirSync(backupDir, { recursive: true });
    }
}

```

```

    }

    // 生成备份文件名 (包含时间戳)
    const timestamp = new Date().toISOString().replace(/[:.]/g, '-');
    const backupPath = path.join(backupDir, `writech_backup_${timestamp}.db`);

    // 使用SQLite的backup API执行在线备份 (不阻塞读写)
    this.db.backup(backupPath);
    console.log('[DatabaseManager] 数据库备份完成:', backupPath);

    // 清理过期备份 (保留最近N个)
    this.cleanOldBackups(backupDir);
    return backupPath;
}

/** 清理过期的备份文件 */
private cleanOldBackups(backupDir: string): void {
    const files = fs.readdirSync(backupDir)
        .filter(f => f.startsWith('writech_backup_'))
        .sort()
        .reverse();

    // 删除超出最大数量的旧备份
    for (let i = this.config.maxBackups; i < files.length; i++) {
        const filePath = path.join(backupDir, files[i]);
        fs.unlinkSync(filePath);
        console.log('[DatabaseManager] 已清理过期备份:', files[i]);
    }
}

/** 启动自动数据库整理 (VACUUM) */
private startAutoVacuum(): void {
    this.vacuumTimer = setInterval(() => {
        try {
            // 清理30天前已同步的笔迹原始数据 (缩略图保留)
            const threshold = Date.now() - 30 * 24 * 60 * 60 * 1000;
            const result = this.db.prepare(`
                DELETE FROM stroke_records
                WHERE sync_status = 1 AND collect_time < ?
            `).run(threshold);
            if (result.changes > 0) {
                console.log(`[DatabaseManager] 清理过期笔迹记录: ${result.changes}条`);
            }

            // 清理已同步的同步日志
            this.db.prepare(`
                DELETE FROM sync_log WHERE sync_status = 1 AND synced_at < ?
            `).run(threshold);

            // 执行VACUUM整理磁盘空间
            this.db.exec('VACUUM');
            console.log('[DatabaseManager] 数据库整理完成');
        } catch (error) {
            console.error('[DatabaseManager] 数据库整理失败:', error);
        }
    }, this.config.autoVacuumInterval);
}

```



```

    /** 获取数据库统计信息（用于状态显示） */
    getStatistics(): Record<string, number> {
        const stats: Record<string, number> = {};
        stats.strokeCount = this.db.prepare('SELECT COUNT(*) as c FROM
stroke_records').get().c;
        stats.gradeCount = this.db.prepare('SELECT COUNT(*) as c FROM
grade_records').get().c;
        stats.studentCount = this.db.prepare('SELECT COUNT(*) as c FROM
student_info').get().c;
        stats.coursewareCount = this.db.prepare('SELECT COUNT(*) as c FROM
courseware_meta').get().c;
        stats.unsyncedCount = this.db.prepare('SELECT COUNT(*) as c FROM sync_log WHERE
sync_status=0').get().c;

        // 计算数据库文件大小
        try {
            const stat = fs.statSync(this.config.dbPath);
            stats.dbSizeBytes = stat.size;
        } catch {
            stats.dbSizeBytes = 0;
        }
        return stats;
    }

    /** 关闭数据库连接并清理资源 */
    close(): void {
        if (this.backupTimer) {
            clearInterval(this.backupTimer);
            this.backupTimer = null;
        }
        if (this.vacuumTimer) {
            clearInterval(this.vacuumTimer);
            this.vacuumTimer = null;
        }
        if (this.db) {
            // 关闭前执行一次checkpoint确保WAL数据写入
            try { this.db.pragma('wal_checkpoint(TRUNCATE)'); } catch {}
            this.db.close();
            this.db = null;
        }
        this.initialized = false;
        console.log('[DatabaseManager] 数据库连接已关闭');
    }
}

/* ===== 单例导出 ===== */

/** 全局数据库管理器实例 */
const dbManager = new DatabaseManager();
export default dbManager;

```

main/

main/device_manager.ts

```
/**
 * 自然写互动课堂PC端应用软件 V1.0
 *
 * device_manager.ts - USB/BLE设备管理
 *
 * 功能说明:
 * - USB HID点阵笔连接管理
 * - BLE蓝牙点阵笔扫描与连接
 * - 设备数据解析 (7字节紧凑坐标解码)
 * - 设备热插拔监听
 * - 多设备并行管理
 */

/* ===== 类型定义 ===== */

/** 设备连接方式 */
enum DeviceInterface {
    USB_HID = 'usb',
    BLE = 'ble'
}

/** 设备状态 */
enum DeviceStatus {
    DISCONNECTED = 'disconnected',
    CONNECTING = 'connecting',
    CONNECTED = 'connected',
    ERROR = 'error'
}

/** 点阵笔设备信息 */
interface PenDevice {
    id: string; /* 设备唯一ID */
    name: string; /* 设备名称 */
    macAddress: string; /* MAC地址 */
    interface: DeviceInterface; /* 连接方式 */
    status: DeviceStatus; /* 连接状态 */
    battery: number; /* 电量百分比 */
    firmwareVersion: string; /* 固件版本 */
    lastConnected: number; /* 最后连接时间戳 */
}

/** 笔迹坐标点 */
interface StrokePoint {
    x: number; /* X坐标(毫米) */
    y: number; /* Y坐标(毫米) */
    pressure: number; /* 压力值(0-1) */
    timestamp: number; /* 时间戳(毫秒) */
    penDown: boolean; /* 落笔标志 */
}

/** 设备事件回调 */
interface DeviceEventCallbacks {
    onDeviceDiscovered: (device: PenDevice) => void;
```

```

    onDeviceConnected: (device: PenDevice) => void;
    onDeviceDisconnected: (deviceId: string) => void;
    onStrokeData: (deviceId: string, points: StrokePoint[]) => void;
    onBatteryUpdate: (deviceId: string, level: number) => void;
    onError: (deviceId: string, error: string) => void;
}

/* ===== USB HID常量 ===== */

/** 自然写点阵笔USB VendorID */
const WRITECH_USB_VID = 0x1234;
/** 自然写点阵笔USB ProductID */
const WRITECH_USB_PID = 0x5678;
/** USB HID报文最大长度 */
const USB_REPORT_SIZE = 64;
/** USB轮询间隔（毫秒） */
const USB_POLL_INTERVAL = 5;

/* ===== BLE常量 ===== */

/** 自然写笔迹服务UUID */
const BLE_SERVICE_UUID = '0000ffe0-0000-1000-8000-00805f9b34fb';
/** 笔迹数据特征UUID (Notify) */
const BLE_STROKE_CHAR_UUID = '0000ffe1-0000-1000-8000-00805f9b34fb';
/** 电量特征UUID */
const BLE_BATTERY_CHAR_UUID = '0000ffe2-0000-1000-8000-00805f9b34fb';
/** 控制特征UUID (Write) */
const BLE_CONTROL_CHAR_UUID = '0000ffe3-0000-1000-8000-00805f9b34fb';

/* ===== 坐标解码 ===== */

/**
 * 解码7字节紧凑坐标编码
 * 编码格式：20位X + 20位Y + 12位压力 + 4位标志
 */
function decodeCompactPoint(data: Buffer, offset: number): StrokePoint {
    /* 提取20位X坐标 */
    const rawX = (data[offset] << 12) |
        (data[offset + 1] << 4) |
        ((data[offset + 2] >> 4) & 0x0F);

    /* 提取20位Y坐标 */
    const rawY = ((data[offset + 2] & 0x0F) << 16) |
        (data[offset + 3] << 8) |
        data[offset + 4];

    /* 提取12位压力值 */
    const rawPressure = (data[offset + 5] << 4) |
        ((data[offset + 6] >> 4) & 0x0F);

    /* 提取4位标志 */
    const flags = data[offset + 6] & 0x0F;

    return {
        x: rawX * 0.3,          /* 点阵码单位转毫米 */
        y: rawY * 0.3,
        pressure: rawPressure / 4095, /* 归一化到0-1 */
        flags
    };
}

```

```

        timestamp: Date.now(),
        penDown: (flags & 0x01) !== 0
    };
}

/**
 * 计算CRC-16 CCITT校验
 */
function crc16CCITT(data: Buffer, length: number): number {
    let crc = 0xFFFF;
    for (let i = 0; i < length; i++) {
        crc ^= data[i] << 8;
        for (let j = 0; j < 8; j++) {
            if (crc & 0x8000) {
                crc = ((crc << 1) ^ 0x1021) & 0xFFFF;
            } else {
                crc = (crc << 1) & 0xFFFF;
            }
        }
    }
    return crc;
}

/* ===== 设备管理器 ===== */

/**
 * 点阵笔设备管理器
 * 统一管理USB和BLE连接的点阵笔设备
 */
class DeviceManager {
    /** 已连接设备列表 */
    private devices: Map<string, PenDevice> = new Map();
    /** 事件回调 */
    private callbacks: DeviceEventCallbacks;
    /** USB轮询定时器 */
    private usbPollTimer: ReturnType<typeof setInterval> | null = null;
    /** BLE扫描状态 */
    private bleScanning: boolean = false;
    /** 是否运行中 */
    private running: boolean = false;

    constructor(callbacks: DeviceEventCallbacks) {
        this.callbacks = callbacks;
        console.log('[设备管理] 初始化');
    }

    /* ===== USB HID管理 ===== */

    /**
     * 启动USB设备监听
     * 使用node-usb库检测设备热插拔
     */
    startUSBMonitor(): void {
        console.log('[设备管理] 启动USB监听');
        this.running = true;

        /* 枚举已连接的USB设备 */

```

```

    this.scanUSBDevices();

    /* 监听USB热插拔事件
       usb.on('attach', (device) => this.onUSBAttach(device));
       usb.on('detach', (device) => this.onUSBDetach(device)); */

    /* 启动USB数据轮询 */
    this.usbPollTimer = setInterval(() => {
        this.pollUSBData();
    }, USB_POLL_INTERVAL);
}

/**
 * 扫描已连接的USB HID设备
 */
private scanUSBDevices(): void {
    /* const devices = HID.devices()
       .filter(d => d.vendorId === WRITECH_USB_VID &&
          d.productId === WRITECH_USB_PID); */

    console.log('[设备管理] USB扫描完成');
}

/**
 * USB设备接入处理
 */
private onUSBAttach(usbDevice: any): void {
    const deviceId = `usb_${usbDevice.serialNumber || Date.now()}`;

    const pen: PenDevice = {
        id: deviceId,
        name: `WritechPen-USB-${deviceId.slice(-4)}`,
        macAddress: '',
        interface: DeviceInterface.USB_HID,
        status: DeviceStatus.CONNECTED,
        battery: 100,
        firmwareVersion: '1.0.0',
        lastConnected: Date.now()
    };

    this.devices.set(deviceId, pen);
    this.callbacks.onDeviceConnected(pen);
    console.log(`[设备管理] USB设备接入: ${pen.name}`);
}

/**
 * USB设备拔出处理
 */
private onUSBDetach(usbDevice: any): void {
    const deviceId = `usb_${usbDevice.serialNumber || ''}`;
    if (this.devices.has(deviceId)) {
        this.devices.delete(deviceId);
        this.callbacks.onDeviceDisconnected(deviceId);
        console.log(`[设备管理] USB设备断开: ${deviceId}`);
    }
}
}

```

```

/**
 * 轮询USB设备数据
 * 读取HID报文并解析坐标
 */
private pollUSBData(): void {
  this.devices.forEach((device, deviceId) => {
    if (device.interface !== DeviceInterface.USB_HID) return;
    if (device.status !== DeviceStatus.CONNECTED) return;

    /* const report = hidDevice.readSync();
    if (report && report.length > 0) {
      this.parseUSBReport(deviceId, Buffer.from(report));
    } */
  });
}

/**
 * 解析USB HID报文
 * 报文格式: [报文类型][数据长度][坐标数据...]
 */
private parseUSBReport(deviceId: string, report: Buffer): void {
  const reportType = report[0];
  const dataLen = report[1];

  if (reportType === 0x01) {
    /* 笔迹数据报文: 每11字节一个坐标点(7字节坐标+4字节时间戳) */
    const points: StrokePoint[] = [];
    const pointSize = 11;

    for (let offset = 2; offset + pointSize <= 2 + dataLen; offset += pointSize)
    {
      const point = decodeCompactPoint(report, offset);
      /* 时间戳从报文中提取 */
      point.timestamp = report.readUInt32LE(offset + 7);
      points.push(point);
    }

    if (points.length > 0) {
      this.callbacks.onStrokeData(deviceId, points);
    }
  } else if (reportType === 0x04) {
    /* 电量报文 */
    const battery = report[2];
    this.callbacks.onBatteryUpdate(deviceId, battery);
  }
}

/* ===== BLE管理 ===== */

/**
 * 启动BLE蓝牙扫描
 */
startBLEScan(): void {
  if (this.bleScanning) return;

  console.log('[设备管理] 启动BLE扫描');
  this.bleScanning = true;
}

```

```

        /* noble.on('discover', (peripheral) => {
            if (peripheral.advertisement.localName?.startsWith('WritechPen')) {
                this.onBLEDiscover(peripheral);
            }
        });
        noble.startScanning([BLE_SERVICE_UUID], true); */
    }

    /**
     * 停止BLE扫描
     */
    stopBLEScan(): void {
        this.bleScanning = false;
        /* noble.stopScanning(); */
        console.log('[设备管理] BLE扫描已停止');
    }

    /**
     * BLE设备发现回调
     */
    private onBLEDiscover(peripheral: any): void {
        const deviceId = `ble_${peripheral.address.replace(/:/g, '')}`;

        if (this.devices.has(deviceId)) return;

        const pen: PenDevice = {
            id: deviceId,
            name: peripheral.advertisement.localName || 'WritechPen',
            macAddress: peripheral.address,
            interface: DeviceInterface.BLE,
            status: DeviceStatus.DISCONNECTED,
            battery: 0,
            firmwareVersion: '',
            lastConnected: 0
        };

        this.callbacks.onDeviceDiscovered(pen);
        console.log(`[设备管理] 发现BLE设备: ${pen.name} [${pen.macAddress}]`);
    }

    /**
     * 连接BLE设备
     */
    async connectBLE(deviceId: string): Promise<boolean> {
        const device = this.devices.get(deviceId);
        if (!device || device.interface !== DeviceInterface.BLE) {
            return false;
        }

        device.status = DeviceStatus.CONNECTING;
        console.log(`[设备管理] 连接BLE设备: ${device.name}`);

        try {
            /* peripheral.connect((err) => { ... });
            peripheral.discoverServices([BLE_SERVICE_UUID], (err, services) => {
                services[0].discoverCharacteristics(..., (err, chars) => {

```

```

        // 订阅笔迹数据Notify
        strokeChar.subscribe();
        strokeChar.on('data', (data) => this.onBLEData(deviceId, data));
    });
    }); */

    device.status = DeviceStatus.CONNECTED;
    device.lastConnected = Date.now();
    this.devices.set(deviceId, device);
    this.callbacks.onDeviceConnected(device);
    return true;
} catch (err: any) {
    device.status = DeviceStatus.ERROR;
    this.callbacks.onError(deviceId, err.message);
    return false;
}
}

/**
 * BLE数据接收回调
 */
private onBLEData(deviceId: string, data: Buffer): void {
    /* BLE数据帧格式与USB类似: [帧头0xAA][类型][长度][数据...][CRC16] */
    if (data[0] !== 0xAA) return;

    const frameType = data[1];
    const payloadLen = data[2];

    /* CRC校验 */
    const expectedCrc = data.readUInt16LE(3 + payloadLen);
    const calcCrc = crc16CCITT(data.slice(0, 3 + payloadLen), 3 + payloadLen);
    if (expectedCrc !== calcCrc) {
        console.warn(`[设备管理] BLE数据CRC校验失败: ${deviceId}`);
        return;
    }

    if (frameType === 0x01) {
        /* 笔迹坐标数据 */
        const points: StrokePoint[] = [];
        const pointSize = 11;
        for (let i = 3; i + pointSize <= 3 + payloadLen; i += pointSize) {
            points.push(decodeCompactPoint(data, i));
        }
        if (points.length > 0) {
            this.callbacks.onStrokeData(deviceId, points);
        }
    } else if (frameType === 0x04) {
        /* 电量数据 */
        this.callbacks.onBatteryUpdate(deviceId, data[3]);
    }
}

/* ===== 公共接口 ===== */

/** 获取所有已连接设备 */
getConnectedDevices(): PenDevice[] {
    return Array.from(this.devices.values())

```



```

        .filter(d => d.status === DeviceStatus.CONNECTED);
    }

    /** 获取设备数量 */
    getDeviceCount(): number {
        return this.devices.size;
    }

    /** 断开指定设备 */
    disconnect(deviceId: string): void {
        const device = this.devices.get(deviceId);
        if (device) {
            device.status = DeviceStatus.DISCONNECTED;
            this.callbacks.onDeviceDisconnected(deviceId);
            console.log(`[设备管理] 断开设备: ${device.name}`);
        }
    }

    /** 停止所有设备管理 */
    shutdown(): void {
        this.running = false;
        if (this.usbPollTimer) {
            clearInterval(this.usbPollTimer);
        }
        this.stopBLEScan();
        this.devices.clear();
        console.log('[设备管理] 已关闭');
    }
}

export { DeviceManager, PenDevice, StrokePoint, DeviceStatus, DeviceInterface };

```

main/main.ts

```

/**
 * 自然写互动课堂PC端应用软件 V1.0
 *
 * main.ts - Electron主进程入口
 *
 * 功能说明:
 * - Electron应用生命周期管理
 * - 主窗口创建与配置
 * - 系统托盘与菜单
 * - IPC通信注册
 * - 自动更新检测
 * - 单实例锁定
 * - 全局异常处理
 */

import { app, BrowserWindow, Menu, Tray, ipcMain, dialog, shell } from 'electron';
import * as path from 'path';
import * as fs from 'fs';

/* ===== 应用配置 ===== */

```

```

/** 应用版本号 */
const APP_VERSION = '1.0.0';
/** 应用名称 */
const APP_NAME = '自然写互动课堂';
/** 窗口默认尺寸 */
const DEFAULT_WIDTH = 1440;
const DEFAULT_HEIGHT = 900;
/** 最小窗口尺寸 */
const MIN_WIDTH = 1024;
const MIN_HEIGHT = 680;
/** 开发模式标志 */
const IS_DEV = process.env.NODE_ENV === 'development';

/* ===== 全局变量 ===== */

/** 主窗口实例 */
let mainWindow: BrowserWindow | null = null;
/** 系统托盘实例 */
let tray: Tray | null = null;
/** 窗口状态保存路径 */
const windowStatePath = path.join(app.getPath('userData'), 'window-state.json');

/* ===== 窗口状态管理 ===== */

/** 保存的窗口状态 */
interface WindowState {
  x?: number;
  y?: number;
  width: number;
  height: number;
  isMaximized: boolean;
}

/**
 * 加载上次保存的窗口状态
 */
function loadWindowState(): WindowState {
  try {
    if (fs.existsSync(windowStatePath)) {
      const data = fs.readFileSync(windowStatePath, 'utf-8');
      return JSON.parse(data);
    }
  } catch (err) {
    console.error('[主进程] 加载窗口状态失败:', err);
  }
  return { width: DEFAULT_WIDTH, height: DEFAULT_HEIGHT, isMaximized: false };
}

/**
 * 保存当前窗口状态
 */
function saveWindowState(win: BrowserWindow): void {
  const bounds = win.getBounds();
  const state: WindowState = {
    x: bounds.x,
    y: bounds.y,

```

```

        width: bounds.width,
        height: bounds.height,
        isMaximized: win.isMaximized()
    });
    try {
        fs.writeFileSync(windowStatePath, JSON.stringify(state, null, 2));
    } catch (err) {
        console.error('[主进程] 保存窗口状态失败:', err);
    }
}

/* ===== 窗口创建 ===== */

/**
 * 创建主窗口
 * 配置安全选项、预加载脚本和窗口参数
 */
function createMainWindow(): void {
    const savedState = loadWindowState();

    mainWindow = new BrowserWindow({
        title: APP_NAME,
        width: savedState.width,
        height: savedState.height,
        x: savedState.x,
        y: savedState.y,
        minWidth: MIN_WIDTH,
        minHeight: MIN_HEIGHT,
        show: false,
        frame: true,
        backgroundColor: '#ffffff',
        webPreferences: {
            /* 安全选项: 渲染进程沙箱化 */
            nodeIntegration: false,
            contextIsolation: true,
            sandbox: true,
            /* 预加载脚本路径 */
            preload: path.join(__dirname, 'preload.js'),
            /* 禁用远程模块 */
            webSecurity: true,
            /* 禁止打开新窗口 */
            allowRunningInsecureContent: false
        }
    });

    /* 加载渲染进程页面 */
    if (IS_DEV) {
        mainWindow.loadURL('http://localhost:5173');
        mainWindow.webContents.openDevTools();
    } else {
        mainWindow.loadFile(path.join(__dirname, '../rendererer/index.html'));
    }

    /* 窗口就绪后显示 (避免白屏闪烁) */
    mainWindow.once('ready-to-show', () => {
        if (savedState.isMaximized) {
            mainWindow?.maximize();
        }
    });
}

```

```

    }
    mainWindow?.show();
    console.log('[主进程] 主窗口已显示');
  });

  /* 窗口关闭前保存状态 */
  mainWindow.on('close', (event) => {
    if (mainWindow) {
      saveWindowState(mainWindow);
    }
  });

  mainWindow.on('closed', () => {
    mainWindow = null;
  });

  /* 拦截外部链接在系统浏览器打开 */
  mainWindow.webContents.setWindowOpenHandler(({ url }) => {
    shell.openExternal(url);
    return { action: 'deny' };
  });

  console.log(`[主进程] 窗口创建完成: ${savedState.width}x${savedState.height}`);
}

/* ===== 系统托盘 ===== */

/**
 * 创建系统托盘图标和菜单
 */
function createTray(): void {
  const iconPath = path.join(__dirname, '../assets/tray-icon.png');
  tray = new Tray(iconPath);
  tray.setToolTip(APP_NAME);

  const contextMenu = Menu.buildFromTemplate([
    { label: '显示主窗口', click: () => mainWindow?.show() },
    { type: 'separator' },
    { label: '设备管理', click: () => sendToRenderer('navigate', '/devices') },
    { label: '设置', click: () => sendToRenderer('navigate', '/settings') },
    { type: 'separator' },
    { label: `版本 ${APP_VERSION}`, enabled: false },
    { label: '退出', click: () => app.quit() }
  ]);

  tray.setContextMenu(contextMenu);
  tray.on('click', () => mainWindow?.show());
}

/* ===== IPC通信处理 ===== */

/**
 * 向渲染进程发送消息
 */
function sendToRenderer(channel: string, data: any): void {
  mainWindow?.webContents.send(channel, data);
}

```

```

/**
 * 注册IPC通信处理器
 * 渲染进程通过IPC调用主进程的系统API
 */
function setupIpcHandlers(): void {
  /* 获取应用信息 */
  ipcMain.handle('app:getInfo', () => ({
    version: APP_VERSION,
    name: APP_NAME,
    platform: process.platform,
    arch: process.arch,
    userDataPath: app.getPath('userData')
  }));

  /* 文件选择对话框 */
  ipcMain.handle('dialog:openFile', async (_, options) => {
    const result = await dialog.showOpenDialog(mainWindow!, {
      title: options.title || '选择文件',
      filters: options.filters || [{ name: '所有文件', extensions: ['*'] }],
      properties: options.properties || ['openFile']
    });
    return result.filePaths;
  });

  /* 保存文件对话框 */
  ipcMain.handle('dialog:saveFile', async (_, options) => {
    const result = await dialog.showSaveDialog(mainWindow!, {
      title: options.title || '保存文件',
      defaultPath: options.defaultPath,
      filters: options.filters || [{ name: '所有文件', extensions: ['*'] }]
    });
    return result.filePath;
  });

  /* 文件读取 */
  ipcMain.handle('fs:readFile', async (_, filePath: string) => {
    return fs.readFileSync(filePath, 'utf-8');
  });

  /* 文件写入 */
  ipcMain.handle('fs:writeFile', async (_, filePath: string, content: string) => {
    fs.writeFileSync(filePath, content, 'utf-8');
    return true;
  });

  /* 打印功能 */
  ipcMain.handle('print:start', async (_, options) => {
    mainWindow?.webContents.print({
      silent: options.silent || false,
      printBackground: true,
      copies: options.copies || 1,
      pageSize: options.pageSize || 'A4'
    });
  });

  /* 窗口控制 */

```

```

ipcMain.on('window:minimize', () => mainWindow?.minimize());
ipcMain.on('window:maximize', () => {
  if (mainWindow?.isMaximized()) {
    mainWindow.unmaximize();
  } else {
    mainWindow?.maximize();
  }
});
ipcMain.on('window:close', () => mainWindow?.close());

console.log('[主进程] IPC处理器注册完成');
}

/* ===== 自动更新 ===== */

/**
 * 检查应用更新
 * 使用electron-updater检查并安装更新
 */
function checkForUpdates(): void {
  if (IS_DEV) return;

  console.log('[主进程] 检查应用更新...');
  /* autoUpdater.checkForUpdatesAndNotify()
   .then(result => { ... })
   .catch(err => { ... }); */
  /* autoUpdater.on('update-available', (info) => {
    sendToRenderer('update:available', info);
  });
  autoUpdater.on('download-progress', (progress) => {
    sendToRenderer('update:progress', progress);
  });
  autoUpdater.on('update-downloaded', (info) => {
    sendToRenderer('update:downloaded', info);
  }); */
}

/* ===== 应用生命周期 ===== */

/** 确保单实例运行 */
const gotLock = app.requestSingleInstanceLock();
if (!gotLock) {
  console.log('[主进程] 已有实例运行, 退出');
  app.quit();
}

app.on('second-instance', () => {
  /* 用户尝试打开第二个实例时, 聚焦已有窗口 */
  if (mainWindow) {
    if (mainWindow.isMinimized()) mainWindow.restore();
    mainWindow.focus();
  }
});

/* 应用就绪 */
app.whenReady().then(() => {
  console.log(`[主进程] ${APP_NAME} v${APP_VERSION} 启动`);

```

```

    createMainWindow();
    createTray();
    setupIpcHandlers();

    /* 延迟检查更新 */
    setTimeout(checkForUpdates, 5000);
  });

  /* macOS特殊处理：所有窗口关闭后重新创建 */
  app.on('activate', () => {
    if (BrowserWindow.getAllWindows().length === 0) {
      createMainWindow();
    }
  });

  /* 所有窗口关闭时退出（macOS除外） */
  app.on('window-all-closed', () => {
    if (process.platform !== 'darwin') {
      app.quit();
    }
  });

  /* 全局异常处理 */
  process.on('uncaughtException', (error) => {
    console.error('[主进程] 未捕获异常:', error);
    dialog.showErrorBox('应用错误', `发生未预期的错误:\n${error.message}`);
  });

```

renderer/api/

renderer/api/cloud_api.ts

```

/**
 * 自然写互动课堂PC端应用软件 V1.0
 *
 * cloud_api.ts - 云平台API通信层
 *
 * 功能说明：
 * - HTTP REST API封装 (Axios)
 * - JWT Token管理与自动刷新
 * - 请求拦截器 (签名/认证/日志)
 * - 响应拦截器 (错误处理/重试)
 * - API类型定义
 * - 离线请求队列
 */

/* ===== 类型定义 ===== */

/** 统一响应格式 */
interface ApiResponse<T = any> {
  code: number;
  msg: string;

```

```
    data: T;
}

/** 分页参数 */
interface PageParams {
    page: number;
    size: number;
    sort?: string;
}

/** 分页响应 */
interface PageResult<T> {
    total: number;
    pages: number;
    current: number;
    records: T[];
}

/** 用户信息 */
interface UserInfo {
    userId: string;
    name: string;
    role: 'admin' | 'teacher' | 'student' | 'parent';
    phone: string;
    schoolId: string;
    schoolName: string;
    avatar: string;
}

/** 课堂信息 */
interface ClassroomInfo {
    classroomId: string;
    className: string;
    grade: string;
    teacherId: string;
    teacherName: string;
    studentCount: number;
    gatewayId: string;
}

/** 作业信息 */
interface AssignmentInfo {
    assignmentId: string;
    title: string;
    type: 'homework' | 'exam' | 'practice';
    classId: string;
    deadline: string;
    status: 'draft' | 'published' | 'closed';
    totalStudents: number;
    submittedCount: number;
}

/** 学情报告 */
interface LearningReport {
    studentId: string;
    studentName: string;
    subject: string;
}
```



```

    overallScore: number;
    writingScore: number;
    strokeOrderAccuracy: number;
    knowledgePoints: { name: string; mastery: number }[];
    trend: { date: string; score: number }[];
  }

  /** 认证令牌 */
  interface AuthTokens {
    accessToken: string;
    refreshToken: string;
    expiresIn: number;    /* 有效期 (秒) */
    tokenType: string;
  }

  /* ===== 配置 ===== */

  /** API基础URL */
  const API_BASE_URL = 'https://api.writech.cn';
  /** 请求超时 */
  const REQUEST_TIMEOUT = 30000;
  /** Token刷新提前量 (毫秒) */
  const TOKEN_REFRESH_AHEAD = 5 * 60 * 1000;
  /** 最大重试次数 */
  const MAX_RETRIES = 3;

  /* ===== Token管理 ===== */

  /** 存储的Token信息 */
  let currentTokens: AuthTokens | null = null;
  /** Token过期时间戳 */
  let tokenExpiresAt: number = 0;
  /** 是否正在刷新Token */
  let isRefreshing: boolean = false;
  /** 等待Token刷新的请求队列 */
  let refreshQueue: Array<(token: string) => void> = [];

  /**
   * 保存认证令牌
   */
  function saveTokens(tokens: AuthTokens): void {
    currentTokens = tokens;
    tokenExpiresAt = Date.now() + tokens.expiresIn * 1000;
    /* 持久化到electron-store */
    console.log(`[API] Token已保存, 有效期至 ${new
Date(tokenExpiresAt).toLocaleString()}`);
  }

  /**
   * 获取当前Access Token
   * 如果即将过期则自动刷新
   */
  async function getValidToken(): Promise<string> {
    if (!currentTokens) {
      throw new Error('未登录');
    }
  }

```

```

/* 检查是否需要刷新 */
if (Date.now() + TOKEN_REFRESH_AHEAD > tokenExpiresAt) {
  if (!isRefreshing) {
    isRefreshing = true;
    try {
      const newTokens = await refreshToken(currentTokens.refreshToken);
      saveTokens(newTokens);
      /* 通知所有等待中的请求 */
      refreshQueue.forEach(resolve => resolve(newTokens.accessToken));
      refreshQueue = [];
    } finally {
      isRefreshing = false;
    }
  } else {
    /* 等待正在进行的刷新完成 */
    return new Promise<string>(resolve => {
      refreshQueue.push(resolve);
    });
  }
}

return currentTokens.accessToken;
}

/* ===== HTTP请求封装 ===== */

/**
 * 通用HTTP请求方法
 */
async function request<T>({
  method: 'GET' | 'POST' | 'PUT' | 'DELETE',
  path: string,
  data?: any,
  retryCount: number = 0
}): Promise<ApiResponse<T>> {
  const url = `${API_BASE_URL}${path}`;
  const headers: Record<string, string> = {
    'Content-Type': 'application/json',
    'Accept': 'application/json'
  };

  /* 添加认证头 */
  try {
    const token = await getValidToken();
    headers['Authorization'] = `Bearer ${token}`;
  } catch {
    /* 登录接口不需要Token */
  }

  /* 添加请求签名 */
  const timestamp = Date.now().toString();
  headers['X-Timestamp'] = timestamp;
  headers['X-Device-Id'] = getDeviceId();

  try {
    const response = await fetch(url, {
      method,

```

```

        headers,
        body: data ? JSON.stringify(data) : undefined,
        signal: AbortSignal.timeout(REQUEST_TIMEOUT)
    });

    const json: ApiResponse<T> = await response.json();

    /* 处理业务错误 */
    if (json.code === 401 && retryCount < 1) {
        /* Token过期, 尝试刷新后重试 */
        console.log('[API] Token过期, 刷新后重试');
        if (currentTokens) {
            const newTokens = await refreshToken(currentTokens.refreshToken);
            saveTokens(newTokens);
            return request<T>(method, path, data, retryCount + 1);
        }
    }

    if (json.code !== 200 && json.code !== 0) {
        console.warn(`[API] 业务错误: ${method} ${path} code=${json.code}
msg=${json.msg}`);
    }

    return json;
} catch (error: any) {
    console.error(`[API] 请求失败: ${method} ${path}`, error.message);

    /* 网络错误重试 */
    if (retryCount < MAX_RETRIES && isNetworkError(error)) {
        const delay = Math.pow(2, retryCount) * 1000;
        console.log(`[API] ${delay}ms后重试 (${retryCount + 1}/${MAX_RETRIES})`);
        await sleep(delay);
        return request<T>(method, path, data, retryCount + 1);
    }

    return { code: -1, msg: error.message || '网络错误', data: null as any };
}

function isNetworkError(error: any): boolean {
    return error.name === 'TypeError' || error.name === 'AbortError';
}

function sleep(ms: number): Promise<void> {
    return new Promise(resolve => setTimeout(resolve, ms));
}

function getDeviceId(): string {
    return 'PC-' + (typeof window !== 'undefined' ?
        navigator.userAgent.slice(-8) : 'unknown');
}

/* ===== API方法 ===== */

/** 用户登录 */
async function login(username: string, password: string):
Promise<ApiResponse<AuthTokens>> {

```

```

    const result = await request<AuthTokens>('POST', '/api/v1/auth/login', {
      username, password, device_type: 'pc'
    });
    if (result.code === 200 && result.data) {
      saveTokens(result.data);
    }
    return result;
  }

  /** 刷新Token */
  async function refreshToken(token: string): Promise<AuthTokens> {
    const resp = await fetch(`${API_BASE_URL}/api/v1/auth/refresh`, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ refresh_token: token })
    });
    const json: ApiResponse<AuthTokens> = await resp.json();
    if (json.code !== 200 || !json.data) {
      throw new Error('Token刷新失败');
    }
    return json.data;
  }

  /** 获取当前用户信息 */
  async function getUserInfo(): Promise<ApiResponse<UserInfo>> {
    return request<UserInfo>('GET', '/api/v1/user/me');
  }

  /** 获取班级列表 */
  async function getClassrooms(): Promise<ApiResponse<ClassroomInfo[]>> {
    return request<ClassroomInfo[]>('GET', '/api/v1/classroom/list');
  }

  /** 获取作业列表 */
  async function getAssignments(classId: string, params: PageParams):
    Promise<ApiResponse<PageResult<AssignmentInfo>>> {
    return request<PageResult<AssignmentInfo>>('GET',
      `/api/v1/assignment/list?
class_id=${classId}&page=${params.page}&size=${params.size}`);
  }

  /** 发布作业 */
  async function publishAssignment(assignment: Partial<AssignmentInfo>):
    Promise<ApiResponse<{ assignmentId: string }>> {
    return request<{ assignmentId: string }>('POST', '/api/v1/assignment/publish',
      assignment);
  }

  /** 上传笔迹数据 */
  async function uploadStrokeData(assignmentId: string, studentId: string,
    strokeData: any[]): Promise<ApiResponse<void>> {
    return request<void>('POST', '/api/v1/stroke/upload', {
      assignment_id: assignmentId,
      student_id: studentId,
      strokes: strokeData
    });
  }
}

```

```

/** 获取AI批改结果 */
async function getGradingResult(assignmentId: string): Promise<ApiResponse<any>> {
    return request<any>('GET', `/api/v1/result/${assignmentId}`);
}

/** 获取学情报告 */
async function getLearningReport(studentId: string):
Promise<ApiResponse<LearningReport>> {
    return request<LearningReport>('GET', `/api/v1/report/student/${studentId}`);
}

/** 下载课件资源 */
async function getResourceDownloadUrl(resourceId: string): Promise<ApiResponse<{ url:
string }>> {
    return request<{ url: string }>('GET', `/api/v1/resource/download/${resourceId}`);
}

/** 退出登录 */
async function logout(): Promise<void> {
    await request<void>('POST', '/api/v1/auth/logout');
    currentTokens = null;
    tokenExpiresAt = 0;
    console.log('[API] 已退出登录');
}

/* ===== 导出 ===== */

export {
    login, logout, getUserInfo, getClassrooms, getAssignments,
    publishAssignment, uploadStrokeData, getGradingResult,
    getLearningReport, getResourceDownloadUrl, saveTokens
};
export type {
    ApiResponse, UserInfo, ClassroomInfo, AssignmentInfo,
    LearningReport, AuthTokens, PageParams, PageResult
};

```

renderer/components/

renderer/components/StrokeCanvas.vue

```

/**
 * 自然写互动课堂PC端应用软件 V1.0
 *
 * StrokeCanvas.vue - 笔迹画布组件
 *
 * 功能说明:
 * - Canvas 2D高性能笔迹渲染
 * - 压力感应笔锋效果
 * - 贝塞尔曲线平滑
 * - 多图层渲染 (背景+已完成笔画+当前笔画)
 * - 笔迹回放动画

```

```

* - 缩放与平移手势
*/

<template>
  <div class="stroke-canvas-container" ref="containerRef">
    <!-- 背景层: 课件/试卷图片 -->
    <canvas ref="bgCanvas" class="canvas-layer canvas-bg"></canvas>
    <!-- 笔迹层: 已完成的笔画 -->
    <canvas ref="strokeCanvas" class="canvas-layer canvas-stroke"></canvas>
    <!-- 活动层: 当前正在绘制的笔画 -->
    <canvas ref="activeCanvas" class="canvas-layer canvas-active"></canvas>

    <!-- 工具栏 -->
    <div class="canvas-toolbar" v-if="showToolbar">
      <button @click="setPenColor('#000000')" :class="{ active: penColor === '#000000' }">黑</button>
      <button @click="setPenColor('#FF0000')" :class="{ active: penColor === '#FF0000' }">红</button>
      <button @click="setPenColor('#0000FF')" :class="{ active: penColor === '#0000FF' }">蓝</button>
      <button @click="toggleEraser" :class="{ active: eraserMode }">橡皮</button>
      <button @click="undo">撤销</button>
      <button @click="redo">重做</button>
      <button @click="clearAll">清空</button>
    </div>

    <!-- 缩放控件 -->
    <div class="zoom-controls">
      <span class="zoom-label">{{ Math.round(scale * 100) }}%</span>
      <button @click="zoomIn">+</button>
      <button @click="zoomOut">-</button>
      <button @click="resetZoom">适应</button>
    </div>
  </div>
</template>

<script setup lang="ts">
import { ref, onMounted, onUnmounted, watch, nextTick } from 'vue';

/* ===== Props与Emits ===== */

interface Props {
  /** 画布宽度 */
  width?: number;
  /** 画布高度 */
  height?: number;
  /** 背景图片URL */
  backgroundImage?: string;
  /** 是否显示工具栏 */
  showToolbar?: boolean;
  /** 是否只读模式 (仅展示笔迹) */
  readonly?: boolean;
}

const props = withDefaults(defineProps<Props>(), {
  width: 1920,
  height: 1080,

```

```

        showToolbar: true,
        readonly: false
    });

    const emit = defineEmits<{
        (e: 'stroke-complete', stroke: StrokeData): void;
        (e: 'stroke-point', point: PointData): void;
    }>();

    /* ===== 类型定义 ===== */

    interface PointData {
        x: number;
        y: number;
        pressure: number;
        timestamp: number;
    }

    interface StrokeData {
        strokeId: string;
        color: string;
        width: number;
        points: PointData[];
    }

    /* ===== 响应式数据 ===== */

    /** DOM引用 */
    const containerRef = ref<HTMLDivElement>();
    const bgCanvas = ref<HTMLCanvasElement>();
    const strokeCanvas = ref<HTMLCanvasElement>();
    const activeCanvas = ref<HTMLCanvasElement>();

    /** 画布上下文 */
    let bgCtx: CanvasRenderingContext2D | null = null;
    let strokeCtx: CanvasRenderingContext2D | null = null;
    let activeCtx: CanvasRenderingContext2D | null = null;

    /** 画笔状态 */
    const penColor = ref('#000000');
    const penWidth = ref(3);
    const eraserMode = ref(false);
    const scale = ref(1.0);

    /** 当前笔画 */
    let currentStroke: StrokeData | null = null;
    /** 已完成笔画列表 */
    const completedStrokes: StrokeData[] = [];
    /** 撤销栈 */
    const undoStack: StrokeData[] = [];
    /** 重做栈 */
    const redoStack: StrokeData[] = [];
    /** 是否正在绘制 */
    let isDrawing = false;

    /* ===== 平滑算法常量 ===== */

```

```

/** 贝塞尔曲线平滑最小距离 */
const SMOOTH_MIN_DIST = 2;
/** 笔锋最小宽度比 */
const PEN_MIN_WIDTH_RATIO = 0.25;
/** 笔锋最大宽度比 */
const PEN_MAX_WIDTH_RATIO = 1.6;

/* ===== 生命周期 ===== */

onMounted(() => {
  initCanvases();
  if (props.backgroundUrl) {
    loadBackground(props.backgroundUrl);
  }
  if (!props.readonly) {
    setupInputHandlers();
  }
});

onUnmounted(() => {
  removeInputHandlers();
});

/* ===== 画布初始化 ===== */

/**
 * 初始化三层画布
 */
function initCanvases(): void {
  const canvases = [bgCanvas.value, strokeCanvas.value, activeCanvas.value];
  canvases.forEach(canvas => {
    if (canvas) {
      canvas.width = props.width;
      canvas.height = props.height;
    }
  });

  bgCtx = bgCanvas.value?.getContext('2d') ?? null;
  strokeCtx = strokeCanvas.value?.getContext('2d') ?? null;
  activeCtx = activeCanvas.value?.getContext('2d') ?? null;

  /** 笔迹层抗锯齿 */
  if (strokeCtx) {
    strokeCtx.lineCap = 'round';
    strokeCtx.lineJoin = 'round';
  }
  if (activeCtx) {
    activeCtx.lineCap = 'round';
    activeCtx.lineJoin = 'round';
  }

  console.log(`[画布] 初始化: ${props.width}x${props.height}`);
}

/**
 * 加载背景图片
 */

```



```

function loadBackground(url: string): void {
  const img = new Image();
  img.onload = () => {
    bgCtx?.drawImage(img, 0, 0, props.width, props.height);
    console.log(`[画布] 背景加载完成: ${url}`);
  };
  img.onerror = () => {
    console.error(`[画布] 背景加载失败: ${url}`);
  };
  img.src = url;
}

/* ===== 输入事件处理 ===== */

function setupInputHandlers(): void {
  const canvas = activeCanvas.value;
  if (!canvas) return;

  canvas.addEventListener('pointerdown', onPointerDown);
  canvas.addEventListener('pointermove', onPointerMove);
  canvas.addEventListener('pointerup', onPointerUp);
  canvas.addEventListener('pointercancel', onPointerUp);

  /* 禁止默认触摸行为（防止页面滚动） */
  canvas.style.touchAction = 'none';
}

function removeInputHandlers(): void {
  const canvas = activeCanvas.value;
  if (!canvas) return;

  canvas.removeEventListener('pointerdown', onPointerDown);
  canvas.removeEventListener('pointermove', onPointerMove);
  canvas.removeEventListener('pointerup', onPointerUp);
  canvas.removeEventListener('pointercancel', onPointerUp);
}

/**
 * 指针按下 - 开始新笔画
 */
function onPointerDown(e: PointerEvent): void {
  if (props.readonly) return;

  isDrawing = true;
  const { canvasX, canvasY } = screenToCanvas(e.offsetX, e.offsetY);
  const pressure = e.pressure || 0.5;

  currentStroke = {
    strokeId: `stroke_${Date.now()}`,
    color: eraserMode.value ? '#FFFFFF' : penColor.value,
    width: penWidth.value,
    points: [{ x: canvasX, y: canvasY, pressure, timestamp: Date.now() }]
  };
}

/**
 * 指针移动 - 添加采样点并实时绘制

```

```

*/
function onPointerMove(e: PointerEvent): void {
  if (!isDrawing || !currentStroke) return;

  const { canvasX, canvasY } = screenToCanvas(e.offsetX, e.offsetY);
  const pressure = e.pressure || 0.5;

  const lastPt = currentStroke.points[currentStroke.points.length - 1];
  const dx = canvasX - lastPt.x;
  const dy = canvasY - lastPt.y;
  const dist = Math.sqrt(dx * dx + dy * dy);

  /* 距离过近跳过 */
  if (dist < SMOOTH_MIN_DIST) return;

  const point: PointData = { x: canvasX, y: canvasY, pressure, timestamp: Date.now() };
};
currentStroke.points.push(point);
emit('stroke-point', point);

/* 增量渲染最新线段 */
drawSegment(activeCtx!, lastPt, point, currentStroke.color, currentStroke.width);
}

/**
 * 指针抬起 - 完成笔画
 */
function onPointerUp(e: PointerEvent): void {
  if (!isDrawing || !currentStroke) return;

  isDrawing = false;

  if (currentStroke.points.length >= 2) {
    completedStrokes.push(currentStroke);
    undoStack.push(currentStroke);
    redoStack.length = 0;

    /* 将笔画绘制到笔迹层 */
    drawFullStroke(strokeCtx!, currentStroke);
    emit('stroke-complete', currentStroke);
  }

  /* 清空活动层 */
  activeCtx?.clearRect(0, 0, props.width, props.height);
  currentStroke = null;
}

/* ===== 绘制函数 ===== */

/**
 * 绘制单个线段（带压力笔锋）
 */
function drawSegment(ctx: CanvasRenderingContext2D, from: PointData,
  to: PointData, color: string, baseWidth: number): void {
  /* 压力感应笔锋：宽度随压力变化 */
  const widthRatio = PEN_MIN_WIDTH_RATIO +
    (PEN_MAX_WIDTH_RATIO - PEN_MIN_WIDTH_RATIO) * to.pressure;

```

```

    const lineWidth = baseWidth * widthRatio;

    ctx.strokeStyle = color;
    ctx.lineWidth = lineWidth;
    ctx.beginPath();
    ctx.moveTo(from.x, from.y);
    ctx.lineTo(to.x, to.y);
    ctx.stroke();
}

/**
 * 绘制完整笔画（贝塞尔曲线平滑）
 */
function drawFullStroke(ctx: CanvasRenderingContext2D, stroke: StrokeData): void {
    const points = stroke.points;
    if (points.length < 2) return;

    ctx.strokeStyle = stroke.color;

    for (let i = 1; i < points.length; i++) {
        const prev = points[i - 1];
        const curr = points[i];

        const widthRatio = PEN_MIN_WIDTH_RATIO +
            (PEN_MAX_WIDTH_RATIO - PEN_MIN_WIDTH_RATIO) * curr.pressure;
        ctx.lineWidth = stroke.width * widthRatio;

        if (i >= 2) {
            /* 二次贝塞尔曲线平滑 */
            const prevPrev = points[i - 2];
            const midX1 = (prevPrev.x + prev.x) / 2;
            const midY1 = (prevPrev.y + prev.y) / 2;
            const midX2 = (prev.x + curr.x) / 2;
            const midY2 = (prev.y + curr.y) / 2;

            ctx.beginPath();
            ctx.moveTo(midX1, midY1);
            ctx.quadraticCurveTo(prev.x, prev.y, midX2, midY2);
            ctx.stroke();
        } else {
            ctx.beginPath();
            ctx.moveTo(prev.x, prev.y);
            ctx.lineTo(curr.x, curr.y);
            ctx.stroke();
        }
    }
}

/* ===== 坐标转换 ===== */

function screenToCanvas(sx: number, sy: number): { canvasX: number; canvasY: number } {
    return {
        canvasX: sx / scale.value,
        canvasY: sy / scale.value
    };
}

```

```

/* ===== 工具栏操作 ===== */

function setPenColor(color: string): void {
    penColor.value = color;
    eraserMode.value = false;
}

function toggleEraser(): void {
    eraserMode.value = !eraserMode.value;
}

function undo(): void {
    const stroke = undoStack.pop();
    if (!stroke) return;

    redoStack.push(stroke);
    completedStrokes.splice(completedStrokes.indexOf(stroke), 1);
    redrawAllStrokes();
}

function redo(): void {
    const stroke = redoStack.pop();
    if (!stroke) return;

    undoStack.push(stroke);
    completedStrokes.push(stroke);
    redrawAllStrokes();
}

function clearAll(): void {
    completedStrokes.length = 0;
    undoStack.length = 0;
    redoStack.length = 0;
    strokeCtx?.clearRect(0, 0, props.width, props.height);
    activeCtx?.clearRect(0, 0, props.width, props.height);
}

function redrawAllStrokes(): void {
    strokeCtx?.clearRect(0, 0, props.width, props.height);
    completedStrokes.forEach(stroke => {
        drawFullStroke(strokeCtx!, stroke);
    });
}

/* ===== 缩放控制 ===== */

function zoomIn(): void {
    scale.value = Math.min(scale.value * 1.25, 3.0);
}

function zoomOut(): void {
    scale.value = Math.max(scale.value / 1.25, 0.25);
}

function resetZoom(): void {
    scale.value = 1.0;
}

```

```

/* ===== 外部笔迹接收 ===== */

/**
 * 接收外部笔迹数据（学生端通过WebSocket推送）
 */
function addExternalStroke(stroke: StrokeData): void {
    completedStrokes.push(stroke);
    drawFullStroke(strokeCtx!, stroke);
}

/**
 * 笔迹回放动画
 */
async function replayStrokes(strokes: StrokeData[], speedMultiplier: number = 1):
Promise<void> {
    for (const stroke of strokes) {
        for (let i = 1; i < stroke.points.length; i++) {
            const prev = stroke.points[i - 1];
            const curr = stroke.points[i];

            drawSegment(strokeCtx!, prev, curr, stroke.color, stroke.width);

            const delay = (curr.timestamp - prev.timestamp) / speedMultiplier;
            await new Promise(resolve => setTimeout(resolve, Math.max(delay, 5)));
        }
    }
}

/* 导出方法供父组件调用 */
defineExpose({ addExternalStroke, replayStrokes, clearAll, loadBackground });
</script>

<style scoped>
.stroke-canvas-container {
    position: relative;
    overflow: hidden;
    background: #f5f5f5;
}
.canvas-layer {
    position: absolute;
    top: 0;
    left: 0;
}
.canvas-bg { z-index: 1; }
.canvas-stroke { z-index: 2; }
.canvas-active { z-index: 3; cursor: crosshair; }
.canvas-toolbar {
    position: absolute;
    bottom: 16px;
    left: 50%;
    transform: translateX(-50%);
    z-index: 10;
    display: flex;
    gap: 8px;
    padding: 8px 16px;
    background: rgba(255,255,255,0.95);
}

```

```

    border-radius: 8px;
    box-shadow: 0 2px 8px rgba(0,0,0,0.15);
  }
  .canvas-toolbar button {
    padding: 6px 14px;
    border: 1px solid #ddd;
    border-radius: 4px;
    background: #fff;
    cursor: pointer;
    font-size: 13px;
  }
  .canvas-toolbar button.active {
    background: #1976d2;
    color: #fff;
    border-color: #1976d2;
  }
  .zoom-controls {
    position: absolute;
    top: 16px;
    right: 16px;
    z-index: 10;
    display: flex;
    align-items: center;
    gap: 6px;
    padding: 4px 10px;
    background: rgba(255,255,255,0.9);
    border-radius: 6px;
    box-shadow: 0 1px 4px rgba(0,0,0,0.1);
  }
  .zoom-label { font-size: 12px; color: #666; min-width: 36px; text-align: center; }
  .zoom-controls button {
    width: 28px;
    height: 28px;
    border: 1px solid #ddd;
    border-radius: 4px;
    background: #fff;
    cursor: pointer;
    font-size: 14px;
  }
</style>

```

renderer/store/

renderer/store/index.ts

```

/**
 * 自然写互动课堂PC端应用软件 V1.0
 *
 * index.ts - Pinia状态管理 (全局Store)
 *
 * 功能说明:
 * - 用户认证状态管理
 * - 课堂状态管理 (当前课堂/学生列表/笔迹数据)

```

```

* - 设备连接状态管理
* - 作业批改状态管理
* - WebSocket实时数据同步
* - 持久化存储 (electron-store)
*/

import { defineStore } from 'pinia';
import { ref, computed, reactive } from 'vue';

/* ===== 类型定义 ===== */

/** 应用视图模式 */
type ViewMode = 'prepare' | 'lesson' | 'grade' | 'report';

/** 设备信息 */
interface DeviceState {
  id: string;
  name: string;
  type: 'usb' | 'ble';
  status: 'connected' | 'disconnected' | 'error';
  battery: number;
}

/** 学生在线状态 */
interface StudentOnlineState {
  studentId: string;
  name: string;
  penId: string;
  online: boolean;
  lastActive: number;
  strokeCount: number;
}

/** 课堂互动数据 */
interface ClassroomLiveData {
  classroomId: string;
  className: string;
  startTime: number;
  onlineStudents: StudentOnlineState[];
  totalStrokes: number;
  isRecording: boolean;
}

/** 批改任务 */
interface GradeTask {
  assignmentId: string;
  studentId: string;
  studentName: string;
  status: 'pending' | 'ai_graded' | 'reviewed' | 'completed';
  aiScore: number;
  teacherScore: number;
  feedback: string;
}

/* ===== 用户Store ===== */

/**

```

```

* 用户认证与信息状态管理
*/
export const useUserStore = defineStore('user', () => {
  /** 是否已登录 */
  const isLoggedIn = ref(false);
  /** 当前用户信息 */
  const userInfo = ref<{
    userId: string;
    name: string;
    role: string;
    phone: string;
    schoolId: string;
    schoolName: string;
    avatar: string;
  } | null>(null);
  /** 登录时间 */
  const loginTime = ref(0);
  /** Token过期时间 */
  const tokenExpiresAt = ref(0);

  /** 用户角色显示名 */
  const roleLabel = computed(() => {
    const roleMap: Record<string, string> = {
      admin: '管理员',
      teacher: '教师',
      student: '学生',
      parent: '家长'
    };
    return roleMap[userInfo.value?.role || ''] || '未知';
  });

  /**
   * 登录成功后设置用户状态
   */
  function setLoggedIn(user: typeof userInfo.value, expiresAt: number): void {
    isLoggedIn.value = true;
    userInfo.value = user;
    loginTime.value = Date.now();
    tokenExpiresAt.value = expiresAt;
    console.log(`[Store] 用户登录: ${user?.name} (${user?.role})`);
  }

  /**
   * 退出登录
   */
  function logout(): void {
    isLoggedIn.value = false;
    userInfo.value = null;
    loginTime.value = 0;
    tokenExpiresAt.value = 0;
    console.log('[Store] 用户已退出');
  }

  return { isLoggedIn, userInfo, loginTime, tokenExpiresAt, roleLabel, setLoggedIn,
    logout };
});

```



```

/* ===== 课堂Store ===== */

/**
 * 课堂状态管理
 * 管理当前课堂的实时数据
 */
export const useClassroomStore = defineStore('classroom', () => {
  /** 当前视图模式 */
  const viewMode = ref<ViewMode>('prepare');
  /** 当前课堂数据 */
  const liveData = ref<ClassroomLiveData | null>(null);
  /** 是否在课堂中 */
  const isInClass = ref(false);
  /** WebSocket连接状态 */
  const wsConnected = ref(false);

  /** 在线学生数 */
  const onlineCount = computed(() =>
    liveData.value?.onlineStudents.filter(s => s.online).length || 0
  );
  /** 总学生数 */
  const totalStudents = computed(() =>
    liveData.value?.onlineStudents.length || 0
  );
  /** 在线率 */
  const onlineRate = computed(() => {
    const total = totalStudents.value;
    return total > 0 ? Math.round((onlineCount.value / total) * 100) : 0;
  });

  /**
   * 开始课堂
   */
  function startClass(classroomId: string, className: string, students:
StudentOnlineState[]): void {
    liveData.value = {
      classroomId,
      className,
      startTime: Date.now(),
      onlineStudents: students,
      totalStrokes: 0,
      isRecording: false
    };
    isInClass.value = true;
    viewMode.value = 'lesson';
    console.log(`[Store] 课堂开始: ${className}, 学生${students.length}人`);
  }

  /**
   * 结束课堂
   */
  function endClass(): void {
    const duration = liveData.value ? Date.now() - liveData.value.startTime : 0;
    console.log(`[Store] 课堂结束, 时长=${Math.round(duration / 60000)}分钟, ` +
      `笔迹=${liveData.value?.totalStrokes}`);
    isInClass.value = false;
    liveData.value = null;
  }
});

```

```

    }

    /**
     * 更新学生在线状态
     */
    function updateStudentStatus(studentId: string, online: boolean): void {
        const student = liveData.value?.onlineStudents.find(s => s.studentId ===
studentId);
        if (student) {
            student.online = online;
            student.lastActive = Date.now();
        }
    }

    /**
     * 累加笔迹数据计数
     */
    function addStrokeCount(count: number): void {
        if (liveData.value) {
            liveData.value.totalStrokes += count;
        }
    }

    /**
     * 切换视图模式
     */
    function setViewMode(mode: ViewMode): void {
        viewMode.value = mode;
        console.log(`[Store] 视图切换: ${mode}`);
    }

    return {
        viewMode, liveData, isInClass, wsConnected,
        onlineCount, totalStudents, onlineRate,
        startClass, endClass, updateStudentStatus, addStrokeCount, setViewMode
    };
});

/* ===== 设备Store ===== */

/**
 * 设备连接状态管理
 */
export const useDeviceStore = defineStore('device', () => {
    /** 已连接设备列表 */
    const devices = ref<DeviceState[]>([]);
    /** 正在扫描BLE */
    const isScanning = ref(false);

    /** 已连接设备数 */
    const connectedCount = computed(() =>
        devices.value.filter(d => d.status === 'connected').length
    );

    /**
     * 添加或更新设备
     */

```

```

function upsertDevice(device: DeviceState): void {
  const idx = devices.value.findIndex(d => d.id === device.id);
  if (idx >= 0) {
    devices.value[idx] = device;
  } else {
    devices.value.push(device);
  }
}

/**
 * 移除设备
 */
function removeDevice(deviceId: string): void {
  devices.value = devices.value.filter(d => d.id !== deviceId);
}

/**
 * 更新设备电量
 */
function updateBattery(deviceId: string, battery: number): void {
  const device = devices.value.find(d => d.id === deviceId);
  if (device) {
    device.battery = battery;
  }
}

return { devices, isScanning, connectedCount, upsertDevice, removeDevice,
updateBattery };
});

/* ===== 批改Store ===== */

/**
 * 作业批改状态管理
 */
export const useGradeStore = defineStore('grade', () => {
  /** 当前批改的作业ID */
  const currentAssignmentId = ref('');
  /** 批改任务列表 */
  const gradeTasks = ref<GradeTask[]>([]);
  /** 当前批改的学生索引 */
  const currentTaskIndex = ref(0);

  /** 待批改数 */
  const pendingCount = computed(() =>
    gradeTasks.value.filter(t => t.status === 'ai_graded' || t.status ===
'pending').length
  );
  /** 已完成数 */
  const completedCount = computed(() =>
    gradeTasks.value.filter(t => t.status === 'completed' || t.status ===
'reviewed').length
  );
  /** 总体进度百分比 */
  const progressPercent = computed(() => {
    const total = gradeTasks.value.length;
    return total > 0 ? Math.round((completedCount.value / total) * 100) : 0;
  });
});

```

```

});
/** 当前批改任务 */
const currentTask = computed(() => gradeTasks.value[currentTaskIndex.value] ||
null);

/**
 * 加载批改任务列表
 */
function loadTasks(assignmentId: string, tasks: GradeTask[]): void {
    currentAssignmentId.value = assignmentId;
    gradeTasks.value = tasks;
    currentTaskIndex.value = 0;
    console.log(`[Store] 加载批改任务: ${tasks.length}份作业`);
}

/**
 * 提交教师批改结果
 */
function submitGrade(studentId: string, score: number, feedback: string): void {
    const task = gradeTasks.value.find(t => t.studentId === studentId);
    if (task) {
        task.teacherScore = score;
        task.feedback = feedback;
        task.status = 'reviewed';
        console.log(`[Store] 批改完成: ${task.studentName}, 分数=${score}`);
    }
}

/**
 * 切换到下一个待批改任务
 */
function nextTask(): boolean {
    for (let i = currentTaskIndex.value + 1; i < gradeTasks.value.length; i++) {
        if (gradeTasks.value[i].status !== 'completed' && gradeTasks.value[i].status
!== 'reviewed') {
            currentTaskIndex.value = i;
            return true;
        }
    }
    return false;
}

/**
 * 切换到上一个任务
 */
function prevTask(): boolean {
    if (currentTaskIndex.value > 0) {
        currentTaskIndex.value--;
        return true;
    }
    return false;
}

return {
    currentAssignmentId, gradeTasks, currentTaskIndex,
    pendingCount, completedCount, progressPercent, currentTask,
    loadTasks, submitGrade, nextTask, prevTask

```

```
};  
});
```