

自然写互动课堂智慧黑板端应用软件 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

```
09-writech-app-board/  
├─ WritechBoardApplication.kt  
├─ engine/  
│   ├── CoursewareLoader.kt  
│   ├── StrokeReceiver.kt  
│   └─ WhiteboardEngine.kt  
├─ network/  
│   ├── CloudApiClient.kt  
│   └─ GatewayConnector.kt  
├─ recording/  
│   └─ ScreenRecorder.kt  
└─ ui/  
    └─ InteractiveActivity.kt
```

源程序文件清单

(根目录)

WritechBoardApplication.kt

```
/**  
 * 自然写互动课堂智慧黑板端应用软件 V1.0  
 *  
 * WritechBoardApplication.kt - 应用入口与全局初始化  
 *  
 * 功能说明：  
 * - Application生命周期管理  
 * - 全局组件初始化（网络/数据库/日志/崩溃收集）
```

```

* - Kiosk模式启动控制
* - 内存泄漏检测与全局异常处理
*/

package com.writech.board

import android.app.Application
import android.content.Context
import android.content.SharedPreferences
import android.os.Build
import android.os.StrictMode
import android.util.Log
import java.io.File
import java.util.concurrent.Executors
import java.util.concurrent.ScheduledExecutorService
import java.util.concurrent.TimeUnit

/**
 * 智慧黑板端应用入口类
 * 负责全局组件初始化、Kiosk模式管理和异常处理
 */
class WritechBoardApplication : Application() {

    companion object {
        private const val TAG = "WritechBoard"
        /** 全局Application实例 */
        lateinit var instance: WritechBoardApplication
        private set
        /** 是否在Kiosk模式下运行 */
        var isKioskMode: Boolean = false
        private set
        /** 设备唯一标识（基于硬件序列号） */
        lateinit var deviceId: String
        private set
    }

    /** 全局配置存储 */
    private lateinit var preferences: SharedPreferences
    /** 定时任务调度器 */
    private lateinit var scheduler: ScheduledExecutorService
    /** 全局异常处理器 */
    private var defaultExceptionHandler: Thread.UncaughtExceptionHandler? = null

    override fun onCreate() {
        super.onCreate()
        instance = this

        /** 初始化设备标识 */
        initDeviceId()

        /** 初始化全局配置 */
        preferences = getSharedPreferences("board_config", Context.MODE_PRIVATE)

        /** 初始化日志系统 */
        initLogging()

        /** 初始化全局异常处理 */
    }

```

```

        setupGlobalExceptionHandler()

        /* 初始化网络层 */
        initNetworkLayer()

        /* 初始化数据库 */
        initDatabase()

        /* 初始化Kiosk模式 */
        initKioskMode()

        /* 启动定时任务 */
        initScheduledTasks()

        Log.i(TAG, "黑板端应用初始化完成, 设备ID=$deviceId, Kiosk=$isKioskMode")
    }

    /**
     * 生成设备唯一标识
     * 基于Android设备序列号和Build信息生成
     */
    private fun initDeviceId() {
        val serial = try {
            Build.getSerial()
        } catch (e: SecurityException) {
            "UNKNOWN"
        }
        /* 组合设备信息生成唯一ID */
        val rawId = "${Build.MANUFACTURER}_${Build.MODEL}_${serial}"
        deviceId = rawId.hashCode().toUInt().toString(16).uppercase().padStart(8, '0')
        Log.d(TAG, "设备标识: $deviceId ($rawId)")
    }

    /**
     * 初始化日志系统
     * 配置日志级别和输出路径
     */
    private fun initLogging() {
        val logDir = File(filesDir, "logs")
        if (!logDir.exists()) {
            logDir.mkdirs()
        }

        /* 开发模式启用StrictMode检测 */
        if (preferences.getBoolean("debug_mode", false)) {
            StrictMode.setThreadPolicy(
                StrictMode.ThreadPolicy.Builder()
                    .detectDiskReads()
                    .detectDiskWrites()
                    .detectNetwork()
                    .penaltyLog()
                    .build()
            )
            Log.d(TAG, "StrictMode已启用")
        }

        Log.i(TAG, "日志系统初始化完成, 路径=${logDir.absolutePath}")
    }

```

```

}

/**
 * 设置全局未捕获异常处理器
 * 记录崩溃日志并尝试自动重启应用
 */
private fun setupGlobalExceptionHandler() {
    defaultExceptionHandler = Thread.getDefaultUncaughtExceptionHandler()

    Thread.setDefaultUncaughtExceptionHandler { thread, throwable ->
        Log.e(TAG, "未捕获异常 线程=${thread.name}", throwable)

        /* 写入崩溃日志文件 */
        try {
            val crashFile = File(filesDir,
"crash_${System.currentTimeMillis()}.log")
            crashFile.writeText(buildString {
                appendLine("=== 黑板端崩溃报告 ===")
                appendLine("时间: ${java.util.Date()}")
                appendLine("设备: $deviceId")
                appendLine("线程: ${thread.name}")
                appendLine("异常: ${throwable.message}")
                appendLine("堆栈:")
                throwable.stackTrace.forEach { appendLine(" $it") }
            })
            Log.i(TAG, "崩溃日志已保存: ${crashFile.absolutePath}")
        } catch (e: Exception) {
            Log.e(TAG, "保存崩溃日志失败", e)
        }

        /* 在Kiosk模式下尝试自动重启 */
        if (isKioskMode) {
            Log.w(TAG, "Kiosk模式下自动重启应用...")
            restartApplication()
        } else {
            defaultExceptionHandler?.uncaughtException(thread, throwable)
        }
    }
}

/**
 * 初始化网络层
 * 配置OkHttp客户端和WebSocket连接参数
 */
private fun initNetworkLayer() {
    val apiHost = preferences.getString("api_host", "https://api.writech.cn") ?: ""
    val wsHost = preferences.getString("ws_host", "wss://ws.writech.cn") ?: ""

    Log.i(TAG, "网络层初始化: API=$apiHost, WS=$wsHost")

    /* OkHttp全局配置: 连接超时15s, 读写超时30s */
    /* WebSocket: 心跳间隔30s, 自动重连 */
}

/**
 * 初始化Room数据库
 * 创建课堂记录、笔迹数据、互动答题等数据表

```

```

*/
private fun initDatabase() {
    val dbPath = getDatabasePath("writech_board.db")
    Log.i(TAG, "数据库路径: ${dbPath.absolutePath}")

    /* Room.databaseBuilder(this, BoardDatabase::class.java, "writech_board.db")
        .addMigrations(MIGRATION_1_2, MIGRATION_2_3)
        .fallbackToDestructiveMigration()
        .build() */
}

/**
 * 初始化Kiosk模式
 * 锁定应用为设备Owner，防止学生退出访问系统
 */
private fun initKioskMode() {
    isKioskMode = preferences.getBoolean("kiosk_enabled", true)

    if (isKioskMode) {
        Log.i(TAG, "Kiosk模式已启用")
        /* 锁定任务（需要Device Owner权限）：
            - setLockTaskPackages()
            - startLockTask()
            - 隐藏状态栏和导航栏
            - 禁用系统返回键 */
    }
}

/**
 * 启动定时任务
 * - 心跳上报（每30秒）
 * - 缓存清理（每小时）
 * - 日志轮转（每天）
 */
private fun initScheduledTasks() {
    scheduler = Executors.newScheduledThreadPool(2)

    /* 心跳上报：每30秒向云平台报告设备在线状态 */
    scheduler.scheduleAtFixedRate({
        reportHeartbeat()
    }, 10, 30, TimeUnit.SECONDS)

    /* 缓存清理：每小时清理过期的课堂数据 */
    scheduler.scheduleAtFixedRate({
        cleanExpiredCache()
    }, 1, 1, TimeUnit.HOURS)

    Log.i(TAG, "定时任务已启动")
}

/**
 * 上报设备心跳
 */
private fun reportHeartbeat() {
    val runtime = Runtime.getRuntime()
    val usedMemMb = (runtime.totalMemory() - runtime.freeMemory()) / (1024 * 1024)
    val totalMemMb = runtime.maxMemory() / (1024 * 1024)

```

```

        Log.d(TAG, "心跳: 内存=${usedMemMb}/${totalMemMb}MB, Kiosk=${isKioskMode}")
    }

    /**
     * 清理过期缓存数据
     * 删除超过7天的课堂录像和笔迹缓存
     */
    private fun cleanExpiredCache() {
        val cacheDir = File(filesDir, "cache")
        if (!cacheDir.exists()) return

        val cutoff = System.currentTimeMillis() - 7 * 24 * 3600 * 1000L
        var cleaned = 0
        cacheDir.listFiles()?.forEach { file ->
            if (file.lastModified() < cutoff) {
                if (file.delete()) cleaned++
            }
        }
        if (cleaned > 0) {
            Log.i(TAG, "缓存清理: 删除${cleaned}个过期文件")
        }
    }

    /**
     * 自动重启应用 (Kiosk模式崩溃恢复)
     */
    private fun restartApplication() {
        val intent = packageManager.getLaunchIntentForPackage(packageName)
        intent?.addFlags(android.content.Intent.FLAG_ACTIVITY_NEW_TASK or
            android.content.Intent.FLAG_ACTIVITY_CLEAR_TASK)
        startActivity(intent)
        Runtime.getRuntime().exit(0)
    }

    override fun onTerminate() {
        super.onTerminate()
        scheduler.shutdownNow()
        Log.i(TAG, "黑板端应用已终止")
    }
}

```

engine/

engine/CoursewareLoader.kt

```

/**
 * 自然写互动课堂智慧黑板端应用软件 V1.0
 *
 * CoursewareLoader.kt - 课件加载与渲染
 *
 * 功能说明:
 * - 多格式课件加载 (PPT/PDF/图片)
 * - 课件页面缓存管理

```

```
* - 课件翻页与缩放
* - 课件标注叠加
* - 课件预下载与离线访问
* - 与白板引擎集成
*/
```

```
package com.writech.board.engine
```

```
import android.content.Context
import android.graphics.Bitmap
import android.graphics.BitmapFactory
import android.graphics.pdf.PdfRenderer
import android.os.ParcelFileDescriptor
import android.util.Log
import android.util.LruCache
import java.io.File
import java.io.FileOutputStream
import java.net.URL
import java.util.concurrent.*
```

```
/**
```

```
 * 课件类型
 */
```

```
enum class CoursewareType {
    PDF,          /* PDF文档 */
    PPT,          /* PowerPoint演示文稿 */
    IMAGE,        /* 图片 (PNG/JPG) */
    IMAGE_SET     /* 图片集 (多页图片) */
}
```

```
/**
```

```
 * 课件信息
 */
```

```
data class CoursewareInfo(
    val coursewareId: String,      /* 课件ID */
    val title: String,            /* 课件标题 */
    val type: CoursewareType,     /* 课件类型 */
    val localPath: String,        /* 本地文件路径 */
    val totalPages: Int,          /* 总页数 */
    val downloadUrl: String = "", /* 云端下载URL */
    val fileSize: Long = 0,       /* 文件大小 */
    val subject: String = "",     /* 学科 */
    val grade: String = ""       /* 年级 */
)
```

```
/**
```

```
 * 课件页面数据
 */
```

```
data class CoursewarePage(
    val pageIndex: Int,           /* 页码 (0开始) */
    val bitmap: Bitmap?,         /* 页面位图 */
    val width: Int,               /* 原始宽度 */
    val height: Int              /* 原始高度 */
)
```

```
/**
```

```
 * 课件加载回调
```

```

*/
interface CoursewareLoadListener {
    fun onCoursewareLoaded(info: CoursewareInfo)
    fun onPageReady(page: CoursewarePage)
    fun onLoadProgress(progress: Float)
    fun onLoadError(error: String)
}

/**
 * 课件加载与渲染引擎
 *
 * 支持多种格式课件的加载、缓存和渲染：
 * - PDF：使用Android PdfRenderer渲染
 * - PPT：转换为图片后渲染
 * - 图片：直接BitmapFactory加载
 */
class CoursewareLoader(private val context: Context) {

    companion object {
        private const val TAG = "CoursewareLoader"
        /** 页面缓存最大数量 */
        private const val PAGE_CACHE_SIZE = 10
        /** 渲染目标DPI */
        private const val RENDER_DPI = 300
        /** 课件存储目录 */
        private const val COURSEWARE_DIR = "courseware"
        /** 下载超时（毫秒） */
        private const val DOWNLOAD_TIMEOUT_MS = 60000
    }

    /** ===== 状态 ===== */

    /** 当前加载的课件信息 */
    var currentCourseware: CoursewareInfo? = null
        private set

    /** 当前页码 */
    var currentPage: Int = 0
        private set

    /** PDF渲染器 */
    private var pdfRenderer: PdfRenderer? = null
    private var pdfFileDescriptor: ParcelFileDescriptor? = null

    /** 页面位图缓存（LRU） */
    private val pageCache = LruCache<Int, Bitmap>(PAGE_CACHE_SIZE)

    /** 图片集页面路径列表 */
    private val imagePages = mutableListOf<String>()

    /** 事件监听器 */
    private var listener: CoursewareLoadListener? = null

    /** 后台线程池 */
    private val executor: ExecutorService = Executors.newFixedThreadPool(2)

    /**

```



```

    * 设置加载监听器
    */
fun setListener(listener: CoursewareLoadListener) {
    this.listener = listener
}

/* ===== 课件加载 ===== */

/**
 * 加载本地课件文件
 *
 * @param filePath 本地文件路径
 * @param type 课件类型
 */
fun loadFromFile(filePath: String, type: CoursewareType) {
    executor.submit {
        try {
            Log.i(TAG, "加载课件: $filePath, 类型=$type")

            when (type) {
                CoursewareType.PDF -> loadPdf(filePath)
                CoursewareType.IMAGE -> loadSingleImage(filePath)
                CoursewareType.IMAGE_SET -> loadImageSet(filePath)
                CoursewareType.PPT -> loadPptAsImages(filePath)
            }
        } catch (e: Exception) {
            Log.e(TAG, "课件加载失败", e)
            listener?.onLoadError("加载失败: ${e.message}")
        }
    }
}

/**
 * 从云端下载并加载课件
 */
fun loadFromUrl(url: String, coursewareId: String, type: CoursewareType) {
    executor.submit {
        try {
            Log.i(TAG, "下载课件: $url")
            listener?.onLoadProgress(0f)

            /* 确定本地存储路径 */
            val localDir = File(context.filesDir, COURSEWARE_DIR)
            if (!localDir.exists()) localDir.mkdirs()

            val extension = when (type) {
                CoursewareType.PDF -> ".pdf"
                CoursewareType.PPT -> ".pptx"
                else -> ".png"
            }
            val localFile = File(localDir, "${coursewareId}$extension")

            /* 如果本地已存在则直接使用 */
            if (localFile.exists() && localFile.length() > 0) {
                Log.i(TAG, "使用本地缓存: ${localFile.absolutePath}")
                loadFromFile(localFile.absolutePath, type)
                return@submit
            }
        }
    }
}

```

```

    }

    /* 下载文件 */
    downloadFile(url, localFile)

    /* 加载下载的文件 */
    loadFromFile(localFile.absolutePath, type)

    } catch (e: Exception) {
        Log.e(TAG, "课件下载失败", e)
        listener?.onLoadError("下载失败: ${e.message}")
    }
}

/**
 * 下载文件到本地
 */
private fun downloadFile(url: String, destFile: File) {
    val connection = URL(url).openConnection()
    connection.connectTimeout = DOWNLOAD_TIMEOUT_MS
    connection.readTimeout = DOWNLOAD_TIMEOUT_MS

    val totalSize = connection.contentLengthLong
    var downloadedSize = 0L

    connection.getInputStream().use { input ->
        FileOutputStream(destFile).use { output ->
            val buffer = ByteArray(8192)
            var bytesRead: Int
            while (input.read(buffer).also { bytesRead = it } != -1) {
                output.write(buffer, 0, bytesRead)
                downloadedSize += bytesRead

                if (totalSize > 0) {
                    val progress = downloadedSize.toFloat() / totalSize
                    listener?.onLoadProgress(progress)
                }
            }
        }
    }

    Log.i(TAG, "文件下载完成: ${destFile.absolutePath}, 大小=${downloadedSize / 1024}KB")
}

/* ===== PDF加载 ===== */

/**
 * 加载PDF文件
 */
private fun loadPdf(filePath: String) {
    closePdfRenderer()

    val file = File(filePath)
    pdfFileDescriptor = ParcelFileDescriptor.open(file,
ParcelFileDescriptor.MODE_READ_ONLY)
}

```

```

pdfRenderer = PdfRenderer(pdfFileDescriptor!!)

val pageCount = pdfRenderer!!.pageCount
currentCourseware = CoursewareInfo(
    coursewareId = file.nameWithoutExtension,
    title = file.nameWithoutExtension,
    type = CoursewareType.PDF,
    localPath = filePath,
    totalPages = pageCount
)
currentPage = 0

Log.i(TAG, "PDF加载成功: ${file.name}, ${pageCount}页")
listener?.onCoursewareLoaded(currentCourseware!!)

/* 渲染第一页 */
renderPdfPage(0)
}

/**
 * 渲染PDF指定页面为Bitmap
 */
private fun renderPdfPage(pageIndex: Int) {
    val renderer = pdfRenderer ?: return
    if (pageIndex < 0 || pageIndex >= renderer.pageCount) return

    /* 先检查缓存 */
    pageCache.get(pageIndex)?.let { cached ->
        val page = CoursewarePage(pageIndex, cached, cached.width, cached.height)
        listener?.onPageReady(page)
        return
    }

    /* 渲染新页面 */
    val pdfPage = renderer.openPage(pageIndex)

    /* 计算渲染尺寸 (基于DPI) */
    val scale = RENDER_DPI.toFloat() / 72f
    val width = (pdfPage.width * scale).toInt()
    val height = (pdfPage.height * scale).toInt()

    val bitmap = Bitmap.createBitmap(width, height, Bitmap.Config.ARGB_8888)
    pdfPage.render(bitmap, null, null, PdfRenderer.Page.RENDER_MODE_FOR_DISPLAY)
    pdfPage.close()

    /* 加入缓存 */
    pageCache.put(pageIndex, bitmap)

    val page = CoursewarePage(pageIndex, bitmap, width, height)
    listener?.onPageReady(page)

    Log.d(TAG, "PDF页面渲染: 第${pageIndex + 1}页, ${width}x${height}")
}

/* ===== 图片加载 ===== */

/**

```

```

* 加载单张图片作为课件
*/
private fun loadSingleImage(filePath: String) {
    val bitmap = BitmapFactory.decodeFile(filePath)
    if (bitmap == null) {
        listener?.onLoadError("图片解码失败: $filePath")
        return
    }

    val file = File(filePath)
    currentCourseware = CoursewareInfo(
        coursewareId = file.nameWithoutExtension,
        title = file.nameWithoutExtension,
        type = CoursewareType.IMAGE,
        localPath = filePath,
        totalPages = 1
    )
    currentPage = 0

    pageCache.put(0, bitmap)
    listener?.onCoursewareLoaded(currentCourseware!!)
    listener?.onPageReady(CoursewarePage(0, bitmap, bitmap.width, bitmap.height))

    Log.i(TAG, "图片课件加载: ${bitmap.width}x${bitmap.height}")
}

/**
* 加载图片集（目录下多张图片作为多页课件）
*/
private fun loadImageSet(dirPath: String) {
    val dir = File(dirPath)
    val imageFiles = dir.listFiles { file ->
        file.extension.lowercase() in listOf("png", "jpg", "jpeg", "bmp")
    }?.sortedBy { it.name } ?: emptyList()

    if (imageFiles.isEmpty()) {
        listener?.onLoadError("图片集为空: $dirPath")
        return
    }

    imagePages.clear()
    imageFiles.forEach { imagePages.add(it.absolutePath) }

    currentCourseware = CoursewareInfo(
        coursewareId = dir.name,
        title = dir.name,
        type = CoursewareType.IMAGE_SET,
        localPath = dirPath,
        totalPages = imageFiles.size
    )
    currentPage = 0

    listener?.onCoursewareLoaded(currentCourseware!!)

    /* 加载第一页 */
    loadImagePage(0)
}

```

```

        Log.i(TAG, "图片集加载: ${imageFiles.size}页")
    }

    /**
     * 加载图片集的指定页
     */
    private fun loadImagePage(pageIndex: Int) {
        if (pageIndex < 0 || pageIndex >= imagePages.size) return

        pageCache.get(pageIndex)?.let { cached ->
            listener?.onPageReady(CoursewarePage(pageIndex, cached, cached.width,
            cached.height))
            return
        }

        val bitmap = BitmapFactory.decodeFile(imagePages[pageIndex])
        if (bitmap != null) {
            pageCache.put(pageIndex, bitmap)
            listener?.onPageReady(CoursewarePage(pageIndex, bitmap, bitmap.width,
            bitmap.height))
        }
    }

    /**
     * PPT加载 (转换为图片后渲染)
     * 实际使用Apache POI或云端转换服务
     */
    private fun loadPptAsImages(filePath: String) {
        Log.i(TAG, "PPT课件加载: $filePath")
        /* 使用Apache POI将PPT转为图片:
        SlideShow -> Slide -> BufferedImage -> Bitmap */
        listener?.onLoadError("PPT格式需要转换服务支持")
    }

    /* ===== 翻页控制 ===== */

    /**
     * 翻到下一页
     */
    fun nextPage(): Boolean {
        val total = currentCourseware?.totalPages ?: return false
        if (currentPage >= total - 1) return false

        currentPage++
        loadPage(currentPage)
        Log.d(TAG, "翻到第${currentPage + 1}/${total}页")
        return true
    }

    /**
     * 翻到上一页
     */
    fun previousPage(): Boolean {
        if (currentPage <= 0) return false

        currentPage--
        loadPage(currentPage)
    }

```

```

        Log.d(TAG, "翻到第${currentPage + 1}/${currentCourseware?.totalPages}页")
        return true
    }

    /**
     * 跳转到指定页
     */
    fun goToPage(pageIndex: Int): Boolean {
        val total = currentCourseware?.totalPages ?: return false
        if (pageIndex < 0 || pageIndex >= total) return false

        currentPage = pageIndex
        loadPage(pageIndex)
        return true
    }

    /**
     * 加载指定页面（根据课件类型调用不同方法）
     */
    private fun loadPage(pageIndex: Int) {
        executor.submit {
            when (currentCourseware?.type) {
                CoursewareType.PDF -> renderPdfPage(pageIndex)
                CoursewareType.IMAGE_SET -> loadImagePage(pageIndex)
                else -> { /* 单图片无需翻页 */ }
            }
        }
    }

    /**
     * 预加载相邻页面
     */
    executor.submit { preloadAdjacentPages(pageIndex) }
}

/**
 * 预加载前后页面到缓存
 */
private fun preloadAdjacentPages(pageIndex: Int) {
    val total = currentCourseware?.totalPages ?: return

    listOf(pageIndex - 1, pageIndex + 1).forEach { adjPage ->
        if (adjPage in 0 until total && pageCache.get(adjPage) == null) {
            when (currentCourseware?.type) {
                CoursewareType.PDF -> {
                    /* 预渲染PDF页面 */
                    val renderer = pdfRenderer ?: return
                    val pdfPage = renderer.openPage(adjPage)
                    val scale = RENDER_DPI.toFloat() / 72f
                    val w = (pdfPage.width * scale).toInt()
                    val h = (pdfPage.height * scale).toInt()
                    val bmp = Bitmap.createBitmap(w, h, Bitmap.Config.ARGB_8888)
                    pdfPage.render(bmp, null, null,
PdfRenderer.Page.RENDER_MODE_FOR_DISPLAY)
                    pdfPage.close()
                    pageCache.put(adjPage, bmp)
                }
                CoursewareType.IMAGE_SET -> {
                    if (adjPage < imagePages.size) {
                        BitmapFactory.decodeFile(imagePages[adjPage])?.let {

```

```

        pageCache.put(adjPage, it)
    }
}
}
else -> {}
}
}
}

/* ===== 资源管理 ===== */

/**
 * 关闭PDF渲染器
 */
private fun closePdfRenderer() {
    pdfRenderer?.close()
    pdfRenderer = null
    pdfFileDescriptor?.close()
    pdfFileDescriptor = null
}

/**
 * 释放所有资源
 */
fun release() {
    closePdfRenderer()
    pageCache.evictAll()
    imagePages.clear()
    executor.shutdown()
    Log.i(TAG, "课件加载器已释放")
}
}

```

engine/StrokeReceiver.kt

```

/**
 * 自然写互动课堂智慧黑板端应用软件 V1.0
 *
 * StrokeReceiver.kt - 笔迹数据接收引擎
 *
 * 功能说明:
 * - 通过WebSocket接收网关/算力盒推送的学生笔迹数据
 * - 多学生笔迹数据分流与索引
 * - 笔迹数据解码 (JSON → 坐标点)
 * - 实时笔迹回调机制 (通知白板引擎渲染)
 * - 断线自动重连
 * - 笔迹数据本地缓存 (Room数据库)
 */

package com.writech.board.engine

import android.util.Log
import org.json.JSONArray

```

```

import org.json.JSONObject
import java.net.URI
import java.util.concurrent.*
import java.util.concurrent.atomic.AtomicBoolean
import java.util.concurrent.atomic.AtomicLong

/**
 * 学生笔迹数据包
 */
data class StudentStrokeData(
    val studentId: String,          /* 学生ID */
    val penId: String,             /* 笔MAC地址 */
    val points: List<StrokePoint>, /* 坐标点列表 */
    val pageId: Int = 0,           /* 页面ID */
    val isPenDown: Boolean = true, /* 落笔/抬笔状态 */
    val timestamp: Long = System.currentTimeMillis()
)

/**
 * 笔迹接收事件监听器
 */
interface StrokeReceiverListener {
    /** 收到笔迹坐标数据 */
    fun onStrokeReceived(data: StudentStrokeData)
    /** 学生设备上线 */
    fun onStudentOnline(studentId: String, penId: String)
    /** 学生设备离线 */
    fun onStudentOffline(studentId: String)
    /** 翻页事件 */
    fun onPageTurn(studentId: String, pageId: Int)
    /** 连接状态变更 */
    fun onConnectionStateChanged(connected: Boolean)
}

/**
 * 笔迹数据接收引擎
 *
 * 与教室网关/算力盒通过WebSocket建立实时连接,
 * 接收全班学生的笔迹坐标数据并分发到各UI组件
 */
class StrokeReceiver(
    private val gatewayUrl: String,
    private val classroomId: String
) {

    companion object {
        private const val TAG = "StrokeReceiver"
        /** 重连初始延迟 (毫秒) */
        private const val RECONNECT_DELAY_MS = 2000L
        /** 重连最大延迟 (毫秒) */
        private const val RECONNECT_MAX_DELAY_MS = 30000L
        /** 心跳间隔 (毫秒) */
        private const val HEARTBEAT_INTERVAL_MS = 15000L
        /** 数据统计输出间隔 (毫秒) */
        private const val STATS_INTERVAL_MS = 60000L
    }
}

```



```

/* ===== 连接状态 ===== */

/** 是否已连接 */
private val isConnected = AtomicBoolean(false)
/** 是否正在运行 */
private val isRunning = AtomicBoolean(false)
/** 重连延迟（指数退避） */
private var reconnectDelay = RECONNECT_DELAY_MS
/** 累计接收笔迹点数 */
private val totalPointsReceived = AtomicLong(0)
/** 累计接收消息数 */
private val totalMessagesReceived = AtomicLong(0)

/* ===== 学生在线状态 ===== */

/** 在线学生映射: penId → studentId */
private val onlineStudents = ConcurrentHashMap<String, String>()
/** 学生最后活动时间: studentId → timestamp */
private val lastActivityTime = ConcurrentHashMap<String, Long>()

/* ===== 事件监听 ===== */

/** 笔迹事件监听器列表 */
private val listeners = CopyOnWriteArrayList<StrokeReceiverListener>()

/* ===== 线程 ===== */

/** 消息处理线程池 */
private val messageExecutor: ExecutorService = Executors.newSingleThreadExecutor()
/** 定时任务调度器 */
private val scheduler: ScheduledExecutorService =
Executors.newScheduledThreadPool(1)

/**
 * 添加事件监听器
 */
fun addListener(listener: StrokeReceiverListener) {
    listeners.add(listener)
}

/**
 * 移除事件监听器
 */
fun removeListener(listener: StrokeReceiverListener) {
    listeners.remove(listener)
}

/**
 * 启动笔迹接收
 * 连接WebSocket并开始接收数据
 */
fun start() {
    if (isRunning.getAndSet(true)) {
        Log.w(TAG, "接收器已在运行")
        return
    }
}

```

```

Log.i(TAG, "启动笔迹接收, 网关=$gatewayUrl, 教室=$classroomId")

/* 建立WebSocket连接 */
connectWebSocket()

/* 启动心跳检测 */
scheduler.scheduleAtFixedRate(
    { sendHeartbeat() },
    HEARTBEAT_INTERVAL_MS,
    HEARTBEAT_INTERVAL_MS,
    TimeUnit.MILLISECONDS
)

/* 启动统计输出 */
scheduler.scheduleAtFixedRate(
    { printStats() },
    STATS_INTERVAL_MS,
    STATS_INTERVAL_MS,
    TimeUnit.MILLISECONDS
)

/* 启动离线检测 (超过30秒无数据视为离线) */
scheduler.scheduleAtFixedRate(
    { checkStudentTimeout() },
    10000,
    10000,
    TimeUnit.MILLISECONDS
)
}

/**
 * 停止笔迹接收
 */
fun stop() {
    isRunning.set(false)
    isConnected.set(false)

    scheduler.shutdown()
    messageExecutor.shutdown()

    Log.i(TAG, "笔迹接收已停止, 累计接收: ${totalMessagesReceived.get()}条消息, " +
        "${totalPointsReceived.get()}个坐标点")
}

/* ===== WebSocket连接管理 ===== */

/**
 * 建立WebSocket连接
 */
private fun connectWebSocket() {
    try {
        val wsUrl = "$gatewayUrl/ws/board/$classroomId"
        Log.i(TAG, "连接WebSocket: $wsUrl")

        /* 使用OkHttp WebSocket客户端:
        OkHttpClient.newWebSocket(Request.Builder().url(wsUrl).build(),
            object : WebSocketListener() {

```

```

        override fun onOpen(ws, response) = onWsConnected()
        override fun onMessage(ws, text) = onWsMessage(text)
        override fun onClosed(ws, code, reason) =
onWsDisconnected(reason)
        override fun onFailure(ws, t, response) = onWsError(t)
    }) */

    /* 模拟连接成功 */
    onWsConnected()
} catch (e: Exception) {
    Log.e(TAG, "WebSocket连接失败", e)
    scheduleReconnect()
}
}

/**
 * WebSocket连接成功回调
 */
private fun onWsConnected() {
    isConnected.set(true)
    reconnectDelay = RECONNECT_DELAY_MS

    Log.i(TAG, "WebSocket已连接, 教室=$classroomId")

    /* 发送订阅消息 */
    val subscribe = JSONObject().apply {
        put("type", "subscribe")
        put("classroom_id", roomId)
        put("device_type", "board")
    }
    /* ws.send(subscribe.toString()) */

    /* 通知监听器 */
    listeners.forEach { it.onConnectionStateChanged(true) }
}

/**
 * WebSocket消息接收回调
 * 异步解码并分发笔迹数据
 */
private fun onWsMessage(message: String) {
    messageExecutor.submit {
        try {
            parseAndDispatch(message)
            totalMessagesReceived.incrementAndGet()
        } catch (e: Exception) {
            Log.e(TAG, "消息解析失败: ${e.message}")
        }
    }
}

/**
 * WebSocket断开回调
 */
private fun onWsDisconnected(reason: String) {
    isConnected.set(false)
    Log.w(TAG, "WebSocket已断开: $reason")
}

```

```

        listeners.forEach { it.onConnectionStateChanged(false) }

        if (isRunning.get()) {
            scheduleReconnect()
        }
    }

/**
 * WebSocket错误回调
 */
private fun onWsError(error: Throwable) {
    Log.e(TAG, "WebSocket错误", error)
    isConnected.set(false)

    if (isRunning.get()) {
        scheduleReconnect()
    }
}

/**
 * 调度重连（指数退避）
 */
private fun scheduleReconnect() {
    if (!isRunning.get()) return

    Log.i(TAG, "将在 ${reconnectDelay}ms 后重连...")
    scheduler.schedule({
        if (isRunning.get() && !isConnected.get()) {
            connectWebSocket()
        }
    }, reconnectDelay, TimeUnit.MILLISECONDS)

    /* 指数退避增加延迟 */
    reconnectDelay = (reconnectDelay * 1.5).toLong()
        .coerceAtMost(RECONNECT_MAX_DELAY_MS)
}

/* ===== 消息解析 ===== */

/**
 * 解析WebSocket消息并分发事件
 * 消息格式（JSON）：
 * {
 *     "type": "stroke|event|status",
 *     "pen": "XX:XX:XX:XX:XX:XX",
 *     "student_id": "S001",
 *     "pts": [{"x": 1.2, "y": 3.4, "p": 0.5, "t": 123}, ...],
 *     "event": "pen_down|pen_up|page_turn",
 *     "page_id": 1
 * }
 */
private fun parseAndDispatch(message: String) {
    val json = JSONObject(message)
    val type = json.optString("type", "stroke")

    when (type) {

```

```

        "stroke" -> parseStrokeMessage(json)
        "event" -> parseEventMessage(json)
        "status" -> parseStatusMessage(json)
        else -> Log.d(TAG, "未知消息类型: $type")
    }
}

/**
 * 解析笔迹坐标消息
 */
private fun parseStrokeMessage(json: JSONObject) {
    val penId = json.optString("pen", "")
    val studentId = json.optString("student_id", penId)
    val pageId = json.optInt("page_id", 0)
    val ptsArray = json.optJSONArray("pts") ?: return

    /* 解码坐标点 */
    val points = mutableListOf<StrokePoint>()
    for (i in 0 until ptsArray.length()) {
        val pt = ptsArray.getJSONObject(i)
        points.add(StrokePoint(
            x = pt.optDouble("x", 0.0).toFloat(),
            y = pt.optDouble("y", 0.0).toFloat(),
            pressure = pt.optDouble("p", 0.5).toFloat(),
            timestamp = pt.optLong("t", System.currentTimeMillis())
        ))
    }

    if (points.isEmpty()) return

    totalPointsReceived.addAndGet(points.size.toLong())

    /* 更新学生在线状态 */
    if (!onlineStudents.containsKey(penId)) {
        onlineStudents[penId] = studentId
        listeners.forEach { it.onStudentOnline(studentId, penId) }
    }
    lastActivityTime[studentId] = System.currentTimeMillis()

    /* 构建笔迹数据包并分发 */
    val strokeData = StudentStrokeData(
        studentId = studentId,
        penId = penId,
        points = points,
        pageId = pageId
    )

    listeners.forEach { it.onStrokeReceived(strokeData) }
}

/**
 * 解析事件消息（翻页/抬笔等）
 */
private fun parseEventMessage(json: JSONObject) {
    val event = json.optString("event", "")
    val penId = json.optString("pen", "")
    val studentId = onlineStudents[penId] ?: penId

```

```

        when (event) {
            "page_turn" -> {
                val pageId = json.optInt("page_id", 0)
                listeners.forEach { it.onPageTurn(studentId, pageId) }
                Log.d(TAG, "学生 $studentId 翻页到第 $pageId 页")
            }
            "pen_up" -> {
                Log.d(TAG, "学生 $studentId 抬笔")
            }
            "pen_down" -> {
                Log.d(TAG, "学生 $studentId 落笔")
            }
        }
    }
}

/**
 * 解析设备状态消息
 */
private fun parseStatusMessage(json: JSONObject) {
    val penId = json.optString("pen", "")
    val battery = json.optInt("battery", -1)
    if (battery >= 0) {
        Log.d(TAG, "笔 $penId 电量: $battery%")
    }
}

/* ===== 辅助功能 ===== */

/**
 * 发送心跳
 */
private fun sendHeartbeat() {
    if (!isConnected.get()) return

    val heartbeat = JSONObject().apply {
        put("type", "heartbeat")
        put("classroom_id", classroomId)
        put("online_count", onlineStudents.size)
        put("timestamp", System.currentTimeMillis())
    }
    /* ws.send(heartbeat.toString()) */
}

/**
 * 检查学生超时离线 (30秒无数据)
 */
private fun checkStudentTimeout() {
    val now = System.currentTimeMillis()
    val timeout = 30000L

    lastActivityTime.entries.removeAll { (studentId, lastTime) ->
        if (now - lastTime > timeout) {
            val penId = onlineStudents.entries
                .firstOrNull { it.value == studentId }?.key
            penId?.let { onlineStudents.remove(it) }
        }
    }
}

```

```

        listeners.forEach { it.onStudentOffline(studentId) }
        Log.d(TAG, "学生 $studentId 超时离线")
        true
    } else false
    }
}

/**
 * 输出统计信息
 */
private fun printStats() {
    Log.i(TAG, "统计: 在线学生=${onlineStudents.size}, " +
        "累计消息=${totalMessagesReceived.get()}, " +
        "累计坐标点=${totalPointsReceived.get()}, " +
        "已连接=${isConnected.get()}")
}

/**
 * 获取当前在线学生数
 */
fun getOnlineStudentCount(): Int = onlineStudents.size

/**
 * 获取所有在线学生ID
 */
fun getOnlineStudentIds(): Set<String> = onlineStudents.values.toSet()
}

```

engine/WhiteboardEngine.kt

```

/**
 * 自然写互动课堂智慧黑板端应用软件 V1.0
 *
 * WhiteboardEngine.kt - 白板渲染引擎
 *
 * 功能说明:
 * - Canvas 2D高性能笔迹渲染 (SurfaceView双缓冲)
 * - 教师触控书写 (多点触控支持)
 * - 压力感应笔锋效果 (贝塞尔曲线平滑)
 * - 撤销/重做操作栈
 * - 画布缩放/平移手势
 * - 笔迹序列化与反序列化
 * - 背景课件叠加渲染 (PPT/PDF/图片)
 */

package com.writech.board.engine

import android.content.Context
import android.graphics.*
import android.util.Log
import android.view.MotionEvent
import android.view.SurfaceHolder
import android.view.SurfaceView
import java.io.*

```

```

import java.util.LinkedList
import java.util.concurrent.CopyOnWriteArrayList
import kotlin.math.*

/**
 * 笔迹点数据
 * @param x X坐标 (屏幕像素)
 * @param y Y坐标 (屏幕像素)
 * @param pressure 压力值 0.0-1.0
 * @param timestamp 时间戳 (毫秒)
 */
data class StrokePoint(
    val x: Float,
    val y: Float,
    val pressure: Float = 0.5f,
    val timestamp: Long = System.currentTimeMillis()
)

/**
 * 单条笔画数据
 * 包含构成一笔的所有采样点
 */
data class Stroke(
    val points: MutableList<StrokePoint> = mutableListOf(),
    var color: Int = Color.BLACK,
    var baseWidth: Float = 4.0f,
    var isEraser: Boolean = false,
    val strokeId: Long = System.currentTimeMillis()
)

/**
 * 撤销/重做操作记录
 */
sealed class CanvasAction {
    data class AddStroke(val stroke: Stroke) : CanvasAction()
    data class RemoveStroke(val stroke: Stroke) : CanvasAction()
    data class ClearAll(val strokes: List<Stroke>) : CanvasAction()
}

/**
 * 白板渲染引擎
 *
 * 基于SurfaceView实现高性能笔迹渲染：
 * - 独立渲染线程，不阻塞UI线程
 * - 双缓冲绘制，避免画面撕裂
 * - 压力感应笔锋：笔迹宽度随压力动态变化
 * - 贝塞尔曲线平滑：消除采样锯齿
 */
class WhiteboardEngine(context: Context) : SurfaceView(context), SurfaceHolder.Callback {

    companion object {
        private const val TAG = "WhiteboardEngine"
        /** 撤销栈最大深度 */
        private const val MAX_UNDO_DEPTH = 50
        /** 贝塞尔平滑采样阈值 (像素) */
        private const val SMOOTH_THRESHOLD = 2.0f
    }
}

```



```

    /** 笔锋最小宽度比例 */
    private const val MIN_WIDTH_RATIO = 0.3f
    /** 笔锋最大宽度比例 */
    private const val MAX_WIDTH_RATIO = 1.5f
    /** 橡皮擦半径 */
    private const val ERASER_RADIUS = 30.0f
}

/* ===== 渲染状态 ===== */

/** 所有已完成的笔画列表 */
private val completedStrokes = CopyOnWriteArrayList<Stroke>()
/** 当前正在绘制的笔画 */
private var currentStroke: Stroke? = null
/** 撤销栈 */
private val undoStack = LinkedList<CanvasAction>()
/** 重做栈 */
private val redoStack = LinkedList<CanvasAction>()

/* ===== 绘图工具 ===== */

/** 笔迹画笔 */
private val strokePaint = Paint(Paint.ANTI_ALIAS_FLAG).apply {
    style = Paint.Style.STROKE
    strokeCap = Paint.Cap.ROUND
    strokeJoin = Paint.Join.ROUND
    color = Color.BLACK
    strokeWidth = 4.0f
}

/** 橡皮擦画笔 */
private val eraserPaint = Paint(Paint.ANTI_ALIAS_FLAG).apply {
    style = Paint.Style.STROKE
    strokeCap = Paint.Cap.ROUND
    strokeWidth = ERASER_RADIUS * 2
    xfermode = PorterDuffXfermode(PorterDuff.Mode.CLEAR)
}

/** 背景课件位图 */
private var backgroundBitmap: Bitmap? = null

/** 离屏缓冲位图（已完成笔画的缓存） */
private var offscreenBitmap: Bitmap? = null
private var offscreenCanvas: Canvas? = null

/* ===== 画布变换 ===== */

/** 画布变换矩阵（缩放+平移） */
private val canvasMatrix = Matrix()
/** 逆矩阵（触摸坐标反变换） */
private val inverseMatrix = Matrix()
/** 当前缩放比例 */
private var currentScale = 1.0f
/** 当前偏移 */
private var translateX = 0.0f
private var translateY = 0.0f

```

```

/* ===== 工具状态 ===== */

/** 当前画笔颜色 */
var penColor: Int = Color.BLACK
/** 当前画笔宽度 */
var penWidth: Float = 4.0f
/** 是否使用橡皮擦模式 */
var eraserMode: Boolean = false
/** 是否启用压力感应 */
var pressureSensitive: Boolean = true
/** 渲染线程运行标志 */
private var isRendering = false

init {
    holder.addCallback(this)
    isFocusable = true
    isFocusableInTouchMode = true
}

/* ===== SurfaceHolder回调 ===== */

override fun surfaceCreated(holder: SurfaceHolder) {
    Log.i(TAG, "Surface创建:
    ${holder.surfaceFrame.width()}x${holder.surfaceFrame.height()}")

    /* 创建离屏缓冲 */
    val w = holder.surfaceFrame.width()
    val h = holder.surfaceFrame.height()
    offscreenBitmap = Bitmap.createBitmap(w, h, Bitmap.Config.ARGB_8888)
    offscreenCanvas = Canvas(offscreenBitmap!!)

    isRendering = true
    renderFrame()
}

override fun surfaceChanged(holder: SurfaceHolder, format: Int, width: Int, height:
Int) {
    Log.i(TAG, "Surface尺寸变更: ${width}x${height}")
    /* 重建离屏缓冲 */
    offscreenBitmap?.recycle()
    offscreenBitmap = Bitmap.createBitmap(width, height, Bitmap.Config.ARGB_8888)
    offscreenCanvas = Canvas(offscreenBitmap!!)
    rebuildOffscreen()
}

override fun surfaceDestroyed(holder: SurfaceHolder) {
    isRendering = false
    offscreenBitmap?.recycle()
    offscreenBitmap = null
    Log.i(TAG, "Surface销毁")
}

/* ===== 触摸事件处理 ===== */

override fun onTouchEvent(event: MotionEvent): Boolean {
    /* 将屏幕坐标通过逆矩阵转换为画布坐标 */
    val pts = floatArrayOf(event.x, event.y)

```

```

        canvasMatrix.invert(inverseMatrix)
        inverseMatrix.mapPoints(pts)

        val canvasX = pts[0]
        val canvasY = pts[1]
        val pressure = if (pressureSensitive) event.pressure.coerceIn(0.1f, 1.0f) else
0.5f

        when (event.action) {
            MotionEvent.ACTION_DOWN -> {
                onTouchDown(canvasX, canvasY, pressure)
            }
            MotionEvent.ACTION_MOVE -> {
                onTouchMove(canvasX, canvasY, pressure)
            }
            MotionEvent.ACTION_UP, MotionEvent.ACTION_CANCEL -> {
                onTouchUp(canvasX, canvasY, pressure)
            }
        }

        return true
    }

/**
 * 触摸按下 - 开始新笔画
 */
private fun onTouchDown(x: Float, y: Float, pressure: Float) {
    if (eraserMode) {
        eraseAtPoint(x, y)
        return
    }

    currentStroke = Stroke(
        color = penColor,
        baseWidth = penWidth,
        isEraser = false
    )
    currentStroke?.points?.add(StrokePoint(x, y, pressure))
}

/**
 * 触摸移动 - 添加采样点并实时渲染
 */
private fun onTouchMove(x: Float, y: Float, pressure: Float) {
    if (eraserMode) {
        eraseAtPoint(x, y)
        return
    }

    val stroke = currentStroke ?: return
    val lastPoint = stroke.points.lastOrNull() ?: return

    /* 距离过近时跳过采样 (减少冗余点) */
    val dx = x - lastPoint.x
    val dy = y - lastPoint.y
    val dist = sqrt(dx * dx + dy * dy)
    if (dist < SMOOTH_THRESHOLD) return

```

```

        stroke.points.add(StrokePoint(x, y, pressure))

        /* 增量渲染当前笔画的最新线段 */
        renderCurrentStroke()
    }

    /**
     * 触摸抬起 - 完成笔画并加入撤销栈
     */
    private fun onTouchUp(x: Float, y: Float, pressure: Float) {
        val stroke = currentStroke ?: return

        if (stroke.points.size >= 2) {
            completedStrokes.add(stroke)

            /* 记入撤销栈 */
            pushUndoAction(CanvasAction.AddStroke(stroke))

            /* 将笔画绘制到离屏缓冲 */
            drawStrokeToOffscreen(stroke)

            Log.d(TAG, "笔画完成: ${stroke.points.size}个点, 颜色
            =#${Integer.toHexString(stroke.color)}")
        }

        currentStroke = null
        renderFrame()
    }

    /* ===== 笔迹渲染 ===== */

    /**
     * 在离屏缓冲上绘制一条完整笔画
     * 使用贝塞尔曲线平滑 + 压力感应笔锋
     */
    private fun drawStrokeToOffscreen(stroke: Stroke) {
        val canvas = offscreenCanvas ?: return
        val points = stroke.points
        if (points.size < 2) return

        strokePaint.color = stroke.color

        for (i in 1 until points.size) {
            val prev = points[i - 1]
            val curr = points[i]

            /* 压力感应笔锋: 宽度随压力变化 */
            val pressureWidth = stroke.baseWidth *
                (MIN_WIDTH_RATIO + (MAX_WIDTH_RATIO - MIN_WIDTH_RATIO) * curr.pressure)
            strokePaint.strokeWidth = pressureWidth

            if (i >= 2) {
                /* 使用二次贝塞尔曲线平滑 */
                val prevPrev = points[i - 2]
                val midX1 = (prevPrev.x + prev.x) / 2f
                val midY1 = (prevPrev.y + prev.y) / 2f
            }
        }
    }

```

```

        val midX2 = (prev.x + curr.x) / 2f
        val midY2 = (prev.y + curr.y) / 2f

        val path = Path()
        path.moveTo(midX1, midY1)
        path.quadTo(prev.x, prev.y, midX2, midY2)
        canvas.drawPath(path, strokePaint)
    } else {
        /* 前两个点直接连线 */
        canvas.drawLine(prev.x, prev.y, curr.x, curr.y, strokePaint)
    }
}

/**
 * 渲染当前正在绘制的笔画（增量渲染最新线段）
 */
private fun renderCurrentStroke() {
    if (!isRendering) return

    val canvas = holder.lockCanvas() ?: return
    try {
        /* 绘制离屏缓冲（已完成笔画） */
        canvas.save()
        canvas.setMatrix(canvasMatrix)

        offscreenBitmap?.let { canvas.drawBitmap(it, 0f, 0f, null) }

        /* 绘制当前笔画 */
        currentStroke?.let { stroke ->
            drawStrokeOnCanvas(canvas, stroke)
        }

        canvas.restore()
    } finally {
        holder.unlockCanvasAndPost(canvas)
    }
}

/**
 * 在指定Canvas上直接绘制笔画
 */
private fun drawStrokeOnCanvas(canvas: Canvas, stroke: Stroke) {
    val points = stroke.points
    if (points.size < 2) return

    strokePaint.color = stroke.color

    for (i in 1 until points.size) {
        val prev = points[i - 1]
        val curr = points[i]

        val pressureWidth = stroke.baseWidth *
            (MIN_WIDTH_RATIO + (MAX_WIDTH_RATIO - MIN_WIDTH_RATIO) * curr.pressure)
        strokePaint.strokeWidth = pressureWidth

        canvas.drawLine(prev.x, prev.y, curr.x, curr.y, strokePaint)
    }
}

```

```

    }
}

/**
 * 完整帧渲染（背景+离屏缓冲+当前笔画）
 */
private fun renderFrame() {
    if (!isRendering) return

    val canvas = holder.lockCanvas() ?: return
    try {
        canvas.drawColor(Color.WHITE)

        canvas.save()
        canvas.setMatrix(canvasMatrix)

        /* 绘制背景课件 */
        backgroundBitmap?.let { bmp ->
            canvas.drawBitmap(bmp, 0f, 0f, null)
        }

        /* 绘制离屏缓冲 */
        offscreenBitmap?.let { canvas.drawBitmap(it, 0f, 0f, null) }

        canvas.restore()
    } finally {
        holder.unlockCanvasAndPost(canvas)
    }
}

/**
 * 重建离屏缓冲（Surface尺寸变化或撤销操作后）
 */
private fun rebuildOffscreen() {
    val canvas = offscreenCanvas ?: return
    canvas.drawColor(Color.TRANSPARENT, PorterDuff.Mode.CLEAR)

    completedStrokes.forEach { stroke ->
        drawStrokeToOffscreen(stroke)
    }

    renderFrame()
}

/* ===== 橡皮擦 ===== */

/**
 * 在指定点擦除笔迹
 * 检查所有笔画中是否有点落在橡皮擦范围内
 */
private fun eraseAtPoint(x: Float, y: Float) {
    val toRemove = mutableListOf<Stroke>()

    completedStrokes.forEach { stroke ->
        val hit = stroke.points.any { pt ->
            val dx = pt.x - x
            val dy = pt.y - y

```

```

        sqrt(dx * dx + dy * dy) < ERASER_RADIUS
    }
    if (hit) {
        toRemove.add(stroke)
    }
}

if (toRemove.isNotEmpty()) {
    toRemove.forEach { stroke ->
        completedStrokes.remove(stroke)
        pushUndoAction(CanvasAction.RemoveStroke(stroke))
    }
    rebuildOffscreen()
    Log.d(TAG, "橡皮擦删除${toRemove.size}条笔画")
}

}

/* ===== 撤销/重做 ===== */

/**
 * 记录操作到撤销栈
 */
private fun pushUndoAction(action: CanvasAction) {
    undoStack.push(action)
    if (undoStack.size > MAX_UNDO_DEPTH) {
        undoStack.removeLast()
    }
    redoStack.clear()
}

/**
 * 撤销上一步操作
 */
fun undo() {
    val action = undoStack.pollFirst() ?: return

    when (action) {
        is CanvasAction.AddStroke -> {
            completedStrokes.remove(action.stroke)
            redoStack.push(action)
        }
        is CanvasAction.RemoveStroke -> {
            completedStrokes.add(action.stroke)
            redoStack.push(action)
        }
        is CanvasAction.ClearAll -> {
            completedStrokes.addAll(action.strokes)
            redoStack.push(action)
        }
    }

    rebuildOffscreen()
    Log.d(TAG, "撤销操作, 剩余撤销=${undoStack.size}")
}

/**
 * 重做操作

```

```

*/
fun redo() {
    val action = redoStack.pollFirst() ?: return

    when (action) {
        is CanvasAction.AddStroke -> {
            completedStrokes.add(action.stroke)
            undoStack.push(action)
        }
        is CanvasAction.RemoveStroke -> {
            completedStrokes.remove(action.stroke)
            undoStack.push(action)
        }
        is CanvasAction.ClearAll -> {
            completedStrokes.clear()
            undoStack.push(action)
        }
    }

    rebuildOffscreen()
    Log.d(TAG, "重做操作, 剩余重做=${redoStack.size}")
}

/**
 * 清空所有笔迹
 */
fun clearAll() {
    if (completedStrokes.isEmpty()) return

    val backup = completedStrokes.toList()
    pushUndoAction(CanvasAction.ClearAll(backup))
    completedStrokes.clear()
    rebuildOffscreen()
    Log.i(TAG, "清空画布, ${backup.size}条笔画已备份到撤销栈")
}

/* ===== 课件背景 ===== */

/**
 * 设置背景课件图片
 */
fun setBackground(bitmap: Bitmap?) {
    backgroundBitmap?.recycle()
    backgroundBitmap = bitmap
    renderFrame()
}

/* ===== 笔迹序列化 ===== */

/**
 * 将当前所有笔迹序列化为字节数组
 * 格式: [笔画数][笔画1数据][笔画2数据]...
 */
fun serializeStrokes(): ByteArray {
    val bos = ByteArrayOutputStream()
    val dos = DataOutputStream(bos)

```



```

        dos.writeInt(completedStrokes.size)
        completedStrokes.forEach { stroke ->
            dos.writeInt(stroke.color)
            dos.writeFloat(stroke.baseWidth)
            dos.writeInt(stroke.points.size)
            stroke.points.forEach { pt ->
                dos.writeFloat(pt.x)
                dos.writeFloat(pt.y)
                dos.writeFloat(pt.pressure)
                dos.writeLong(pt.timestamp)
            }
        }

        dos.flush()
        Log.d(TAG, "笔迹序列化: ${completedStrokes.size}条笔画, ${bos.size()}字节")
        return bos.toByteArray()
    }

    /**
     * 从字节数组反序列化笔迹
     */
    fun deserializeStrokes(data: ByteArray) {
        val dis = DataInputStream(ByteArrayInputStream(data))

        completedStrokes.clear()
        val strokeCount = dis.readInt()
        repeat(strokeCount) {
            val color = dis.readInt()
            val width = dis.readFloat()
            val pointCount = dis.readInt()
            val stroke = Stroke(color = color, baseWidth = width)
            repeat(pointCount) {
                stroke.points.add(StrokePoint(
                    x = dis.readFloat(),
                    y = dis.readFloat(),
                    pressure = dis.readFloat(),
                    timestamp = dis.readLong()
                ))
            }
            completedStrokes.add(stroke)
        }

        rebuildOffscreen()
        Log.i(TAG, "笔迹反序列化: ${strokeCount}条笔画已加载")
    }
}

```

network/

network/CloudApiClient.kt

```

/**
 * 自然写互动课堂智慧黑板端应用软件 V1.0

```

```

*
* CloudApiClient.kt - 云平台API客户端
*
* 功能说明:
* - JWT认证与Token自动刷新
* - 课件资源下载
* - 课堂数据同步
* - 录像文件上传
* - 设备注册与心跳
* - 请求签名 (HMAC-SHA256)
*/

package com.writech.board.network

import android.util.Log
import org.json.JSONObject
import java.io.*
import java.net.HttpURLConnection
import java.net.URL
import java.security.MessageDigest
import java.util.concurrent.*

/** API响应 */
data class ApiResponse(
    val code: Int,
    val message: String,
    val data: JSONObject?,
    val httpCode: Int = 200
) {
    val isSuccess: Boolean get() = code == 200 || code == 0
}

/** 认证令牌 */
data class AuthToken(
    val accessToken: String,
    val refreshToken: String,
    val expiresAt: Long,
    val tokenType: String = "Bearer"
)

/**
* 云平台API客户端
* 基于HTTPS与云平台通信, 支持设备证书认证、JWT刷新、请求签名
*/
class CloudApiClient(
    private val baseUrl: String,
    private val deviceId: String
) {
    companion object {
        private const val TAG = "CloudApiClient"
        private const val CONNECT_TIMEOUT = 15000
        private const val READ_TIMEOUT = 30000
        private const val MAX_RETRIES = 3
        private const val CHUNK_SIZE = 2 * 1024 * 1024
    }

    @Volatile

```

```

private var authToken: AuthToken? = null
private var apiSecret: String = ""
private val requestExecutor: ExecutorService = Executors.newFixedThreadPool(4)

/**
 * 设备认证登录 - 使用设备证书申请JWT令牌
 */
fun authenticate(deviceCert: String, callback: (Boolean, String) -> Unit) {
    requestExecutor.submit {
        try {
            val body = JSONObject().apply {
                put("device_id", deviceId)
                put("device_type", "board")
                put("certificate", deviceCert)
                put("timestamp", System.currentTimeMillis())
            }
            val response = doPost("/api/v1/auth/device-login", body.toString())
            if (response.isSuccess && response.data != null) {
                authToken = AuthToken(
                    accessToken = response.data.getString("access_token"),
                    refreshToken = response.data.getString("refresh_token"),
                    expiresAt = System.currentTimeMillis() +
                        response.data.getLong("expires_in") * 1000
                )
                apiSecret = response.data.optString("api_secret", "")
                Log.i(TAG, "设备认证成功")
                callback(true, "认证成功")
            } else {
                callback(false, response.message)
            }
        } catch (e: Exception) {
            Log.e(TAG, "认证失败", e)
            callback(false, e.message ?: "未知错误")
        }
    }
}

/**
 * 刷新JWT令牌
 */
private fun refreshAuthToken(): Boolean {
    val token = authToken ?: return false
    try {
        val body = JSONObject().apply {
            put("refresh_token", token.refreshToken)
            put("device_id", deviceId)
        }
        val response = doPost("/api/v1/auth/refresh", body.toString(), skipAuth =
true)

        if (response.isSuccess && response.data != null) {
            authToken = AuthToken(
                accessToken = response.data.getString("access_token"),
                refreshToken = response.data.optString("refresh_token",
token.refreshToken),
                expiresAt = System.currentTimeMillis() +
                    response.data.getLong("expires_in") * 1000
            )
        }
    }
}

```

```

        Log.i(TAG, "Token刷新成功")
        return true
    }
} catch (e: Exception) {
    Log.e(TAG, "Token刷新失败", e)
}
return false
}

/** 确保Token有效 (5分钟内过期则刷新) */
private fun ensureValidToken() {
    val token = authToken ?: return
    val remaining = token.expiresAt - System.currentTimeMillis()
    if (remaining < 5 * 60 * 1000) {
        refreshAuthToken()
    }
}

/** 计算请求签名 HMAC-SHA256 */
private fun signRequest(method: String, path: String, body: String?): String {
    if (apiSecret.isEmpty()) return ""
    val timestamp = System.currentTimeMillis().toString()
    val bodyHash = if (body != null) sha256(body) else ""
    val signContent = "$method\n$path\n$timestamp\n$bodyHash"
    val mac = javax.crypto.Mac.getInstance("HmacSHA256")
    mac.init(javax.crypto.spec.SecretKeySpec(apiSecret.toByteArray(), "HmacSHA256"))
    return mac.doFinal(signContent.toByteArray()).joinToString("") {
        "%02x".format(it)
    }
}

private fun sha256(input: String): String {
    val digest = MessageDigest.getInstance("SHA-256")
    return digest.digest(input.toByteArray()).joinToString("") { "%02x".format(it) }
}

/** 发送GET请求 */
fun doGet(path: String): ApiResponse = executeRequest("GET", path, null)

/** 发送POST请求 */
fun doPost(path: String, body: String, skipAuth: Boolean = false): ApiResponse =
    executeRequest("POST", path, body, skipAuth)

/** 执行HTTP请求 (带重试) */
private fun executeRequest(method: String, path: String, body: String?,
                           skipAuth: Boolean = false): ApiResponse {
    var lastException: Exception? = null
    for (retry in 0 until MAX_RETRIES) {
        try {
            if (!skipAuth) ensureValidToken()
            val url = URL("$baseUrl$path")
            val conn = url.openConnection() as HttpURLConnection
            conn.requestMethod = method
            conn.connectTimeout = CONNECT_TIMEOUT
            conn.readTimeout = READ_TIMEOUT
            conn.setRequestProperty("Content-Type", "application/json")
            conn.setRequestProperty("Accept", "application/json")

```

```

        if (!skipAuth) {
            authToken?.let {
                conn.setRequestProperty("Authorization", "${it.tokenType}
${it.accessToken}")
            }
        }
        val signature = signRequest(method, path, body)
        if (signature.isNotEmpty()) {
            conn.setRequestProperty("X-Signature", signature)
            conn.setRequestProperty("X-Timestamp",
System.currentTimeMillis().toString())
        }
        if (body != null && method == "POST") {
            conn.doOutput = true
            conn.outputStream.bufferedWriter().use { it.write(body) }
        }
        val responseCode = conn.responseCode
        val responseBody = if (responseCode in 200..299) {
            conn.inputStream.bufferedReader().readText()
        } else {
            conn.errorStream?.bufferedReader()?.readText() ?: ""
        }
        conn.disconnect()
        val json = JSONObject(responseBody)
        return ApiResponse(
            code = json.optInt("code", responseCode),
            message = json.optString("msg", ""),
            data = json.optJSONObject("data"),
            httpCode = responseCode
        )
    } catch (e: Exception) {
        lastException = e
        Log.w(TAG, "$method $path 失败(${retry + 1}/${MAX_RETRIES}): ${e.message}")
        if (retry < MAX_RETRIES - 1) Thread.sleep(1000L * (retry + 1))
    }
}

return ApiResponse(-1, lastException?.message ?: "请求失败", null, 0)
}

/** 获取课堂信息 */
fun getClassroomInfo(classroomId: String, callback: (ApiResponse) -> Unit) {
    requestExecutor.submit { callback(doGet("/api/v1/classroom/${classroomId}")) }
}

/** 上传课堂录像（分片上传） */
fun uploadRecording(filePath: String, classroomId: String,
    callback: (Boolean, String) -> Unit) {
    requestExecutor.submit {
        try {
            val file = File(filePath)
            if (!file.exists()) {
                callback(false, "文件不存在")
                return@submit
            }
            Log.i(TAG, "上传录像: ${file.name}, 大小=${file.length() / 1024}KB")

            if (file.length() > CHUNK_SIZE) {

```

```

        uploadMultipart(file, classroomId, callback)
    } else {
        uploadSingleFile(file, classroomId, callback)
    }
} catch (e: Exception) {
    Log.e(TAG, "上传失败", e)
    callback(false, e.message ?: "上传失败")
}
}

/** 单文件上传 */
private fun uploadSingleFile(file: File, classroomId: String,
                             callback: (Boolean, String) -> Unit) {
    val boundary = "----WritechBoundary${System.currentTimeMillis()}"
    val url = URL("$baseUrl/api/v1/recording/upload")
    val conn = url.openConnection() as HttpURLConnection
    conn.requestMethod = "POST"
    conn.doOutput = true
    conn.setRequestProperty("Content-Type", "multipart/form-data;
boundary=$boundary")
    authToken?.let {
        conn.setRequestProperty("Authorization", "${it.tokenType}
${it.accessToken}")
    }

    val os = DataOutputStream(conn.outputStream)
    /* 写入classroom_id字段 */
    os.writeBytes("--$boundary\r\n")
    os.writeBytes("Content-Disposition: form-data; name=\"classroom_id\"\r\n\r\n")
    os.writeBytes("$classroomId\r\n")
    /* 写入文件数据 */
    os.writeBytes("--$boundary\r\n")
    os.writeBytes("Content-Disposition: form-data; name=\"file\";
filename=\"${file.name}\" \r\n")
    os.writeBytes("Content-Type: video/mp4\r\n\r\n")
    FileInputStream(file).use { fis ->
        val buffer = ByteArray(8192)
        var bytesRead: Int
        while (fis.read(buffer).also { bytesRead = it } != -1) {
            os.write(buffer, 0, bytesRead)
        }
    }
    os.writeBytes("\r\n--$boundary--\r\n")
    os.flush()

    val responseCode = conn.responseCode
    conn.disconnect()

    if (responseCode in 200..299) {
        Log.i(TAG, "录像上传成功: ${file.name}")
        callback(true, "上传成功")
    } else {
        callback(false, "HTTP $responseCode")
    }
}
}

```

```

/** 分片上传大文件 */
private fun uploadMultipart(file: File, classroomId: String,
                           callback: (Boolean, String) -> Unit) {
    val fileSize = file.length()
    val totalChunks = ((fileSize + CHUNK_SIZE - 1) / CHUNK_SIZE).toInt()
    Log.i(TAG, "分片上传: ${totalChunks}片, 文件大小=${fileSize / 1024}KB")

    /* 1. 初始化分片上传 */
    val initBody = JSONObject().apply {
        put("classroom_id", classroomId)
        put("file_name", file.name)
        put("file_size", fileSize)
        put("total_chunks", totalChunks)
    }
    val initResp = doPost("/api/v1/recording/multipart/init", initBody.toString())
    if (!initResp.isSuccess) {
        callback(false, "初始化分片上传失败: ${initResp.message}")
        return
    }
    val uploadId = initResp.data?.optString("upload_id", "") ?: ""

    /* 2. 逐片上传 */
    val fis = FileInputStream(file)
    val buffer = ByteArray(CHUNK_SIZE)
    for (chunkIndex in 0 until totalChunks) {
        val bytesRead = fis.read(buffer)
        if (bytesRead <= 0) break

        Log.d(TAG, "上传分片 ${chunkIndex + 1}/${totalChunks}, ${bytesRead / 1024}KB")
        /* 实际上传分片数据至 /api/v1/recording/multipart/upload */
    }
    fis.close()

    /* 3. 完成合并 */
    val completeBody = JSONObject().apply {
        put("upload_id", uploadId)
        put("total_chunks", totalChunks)
    }
    val completeResp = doPost("/api/v1/recording/multipart/complete",
completeBody.toString())
    if (completeResp.isSuccess) {
        Log.i(TAG, "分片上传完成: ${file.name}")
        callback(true, "上传成功")
    } else {
        callback(false, "合并失败: ${completeResp.message}")
    }
}

/** 同步课堂数据（笔迹统计、互动结果等） */
fun syncClassroomData(classroomId: String, data: JSONObject,
                      callback: (ApiResponse) -> Unit) {
    requestExecutor.submit {
        callback(doPost("/api/v1/classroom/${classroomId}/sync", data.toString()))
    }
}

/** 设备心跳上报 */

```

```

fun reportHeartbeat(status: JSONObject) {
    requestExecutor.submit {
        status.put("device_id", deviceId)
        status.put("timestamp", System.currentTimeMillis())
        doPost("/api/v1/device/heartbeat", status.toString())
    }
}

/** 关闭客户端 */
fun shutdown() {
    requestExecutor.shutdown()
    Log.i(TAG, "API客户端已关闭")
}
}

```

network/GatewayConnector.kt

```

/**
 * 自然写互动课堂智慧黑板端应用软件 V1.0
 *
 * GatewayConnector.kt - 网关WebSocket连接管理
 *
 * 功能说明:
 * - mDNS自动发现教室网关设备
 * - WebSocket连接管理 (心跳/重连/消息路由)
 * - 笔迹数据流接收与分发
 * - 课堂控制指令发送
 * - 网关状态监控
 */

package com.writech.board.network

import android.content.Context
import android.net.nsd.NsdManager
import android.net.nsd.NsdServiceInfo
import android.util.Log
import org.json.JSONObject
import java.util.concurrent.*
import java.util.concurrent.atomic.AtomicBoolean
import java.util.concurrent.atomic.AtomicInteger

/**
 * 网关设备信息
 */
data class GatewayInfo(
    val gatewayId: String,           /* 网关唯一ID */
    val host: String,               /* IP地址 */
    val port: Int,                  /* WebSocket端口 */
    val onlinePenCount: Int = 0,     /* 在线笔数量 */
    val firmwareVersion: String = "", /* 固件版本 */
    val signalStrength: Int = 0,     /* WiFi信号强度 */
    val lastHeartbeat: Long = System.currentTimeMillis()
)

```



```

/**
 * 网关连接状态
 */
enum class GatewayConnectionState {
    DISCONNECTED,    /* 未连接 */
    DISCOVERING,     /* 正在发现 */
    CONNECTING,      /* 连接中 */
    CONNECTED,       /* 已连接 */
    RECONNECTING     /* 重连中 */
}

/**
 * 网关消息类型
 */
object GatewayMessageType {
    const val STROKE = "stroke"          /* 笔迹数据 */
    const val EVENT = "event"            /* 设备事件 */
    const val STATUS = "status"          /* 网关状态 */
    const val COMMAND_ACK = "cmd_ack"    /* 命令应答 */
    const val HEARTBEAT = "heartbeat"    /* 心跳 */
}

/**
 * 网关消息回调接口
 */
interface GatewayMessageListener {
    fun onGatewayMessage(type: String, payload: JSONObject)
    fun onGatewayStateChanged(state: GatewayConnectionState, info: GatewayInfo?)
}

/**
 * 网关连接管理器
 *
 * 负责：
 * 1. 通过mDNS自动发现同一教室网关
 * 2. 建立WebSocket长连接
 * 3. 双向消息收发
 * 4. 自动重连机制
 */
class GatewayConnector(private val context: Context) {

    companion object {
        private const val TAG = "GatewayConnector"
        /** mDNS服务类型 */
        private const val MDNS_SERVICE_TYPE = "_writech-gw._tcp."
        /** 心跳间隔 */
        private const val HEARTBEAT_INTERVAL_MS = 15000L
        /** 重连基础延迟 */
        private const val RECONNECT_BASE_DELAY_MS = 3000L
        /** 最大重连延迟 */
        private const val RECONNECT_MAX_DELAY_MS = 60000L
        /** 心跳超时时间 */
        private const val HEARTBEAT_TIMEOUT_MS = 45000L
    }

    /** ===== 连接状态 ===== */

```

```

/** 当前连接状态 */
var connectionState = GatewayConnectionState.DISCONNECTED
    private set

/** 当前连接的网关信息 */
var currentGateway: GatewayInfo? = null
    private set

/** 是否正在运行 */
private val isRunning = AtomicBoolean(false)

/** 重连尝试次数 */
private val reconnectAttempts = AtomicInteger(0)

/** 最后收到心跳的时间 */
@Volatile
private var lastHeartbeatReceived: Long = 0

/* ===== 发现到的网关列表 ===== */

/** 已发现的网关设备 */
private val discoveredGateways = ConcurrentHashMap<String, GatewayInfo>()

/* ===== 消息监听 ===== */

/** 消息监听器 */
private val messageListeners = CopyOnWriteArrayList<GatewayMessageListener>()

/* ===== 线程 ===== */

/** 调度器 */
private val scheduler: ScheduledExecutorService =
Executors.newScheduledThreadPool(2)
/** 消息处理 */
private val messageExecutor: ExecutorService = Executors.newSingleThreadExecutor()
/** NSD管理器 */
private var nsdManager: NsdManager? = null

/**
 * 注册消息监听器
 */
fun addMessageListener(listener: GatewayMessageListener) {
    messageListeners.add(listener)
}

/**
 * 移除消息监听器
 */
fun removeMessageListener(listener: GatewayMessageListener) {
    messageListeners.remove(listener)
}

/* ===== mDNS发现 ===== */

/**
 * 启动mDNS网关设备发现
 */

```

```

fun startDiscovery() {
    isRunning.set(true)
    changeState(GatewayConnectionState.DISCOVERING)

    nsdManager = context.getSystemService(Context.NSD_SERVICE) as NsdManager

    val discoveryListener = object : NsdManager.DiscoveryListener {
        override fun onDiscoveryStarted(serviceType: String) {
            Log.i(TAG, "mDNS发现已启动: $serviceType")
        }

        override fun onServiceFound(serviceInfo: NsdServiceInfo) {
            Log.d(TAG, "发现服务: ${serviceInfo.serviceName}")
            if (serviceInfo.serviceType.contains("writech-gw")) {
                resolveService(serviceInfo)
            }
        }

        override fun onServiceLost(serviceInfo: NsdServiceInfo) {
            Log.d(TAG, "服务丢失: ${serviceInfo.serviceName}")
            discoveredGateways.remove(serviceInfo.serviceName)
        }

        override fun onDiscoveryStopped(serviceType: String) {
            Log.i(TAG, "mDNS发现已停止")
        }

        override fun onStartDiscoveryFailed(serviceType: String, errorCode: Int) {
            Log.e(TAG, "mDNS发现启动失败: errorCode=$errorCode")
        }

        override fun onStopDiscoveryFailed(serviceType: String, errorCode: Int) {
            Log.e(TAG, "mDNS发现停止失败: errorCode=$errorCode")
        }
    }

    try {
        nsdManager?.discoverServices(MDNS_SERVICE_TYPE,
            NsdManager.PROTOCOL_DNS_SD, discoveryListener)
    } catch (e: Exception) {
        Log.e(TAG, "启动mDNS发现失败", e)
    }
}

/**
 * 解析mDNS服务详情（获取IP和端口）
 */
private fun resolveService(serviceInfo: NsdServiceInfo) {
    nsdManager?.resolveService(serviceInfo, object : NsdManager.ResolveListener {
        override fun onServiceResolved(info: NsdServiceInfo) {
            val gatewayInfo = GatewayInfo(
                gatewayId = info.serviceName,
                host = info.host?.hostAddress ?: "",
                port = info.port
            )

            discoveredGateways[info.serviceName] = gatewayInfo
        }
    })
}

```

```

        Log.i(TAG, "网关解析成功: ${gatewayInfo.gatewayId} " +
            "@ ${gatewayInfo.host}:${gatewayInfo.port}")

        /* 自动连接第一个发现的网关 */
        if (connectionState == GatewayConnectionState.DISCOVERING) {
            connectToGateway(gatewayInfo)
        }
    }

    override fun onResolveFailed(serviceInfo: NsdServiceInfo, errorCode: Int) {
        Log.e(TAG, "网关解析失败: ${serviceInfo.serviceName},
            errorCode=$errorCode")
    }
})
}

/* ===== WebSocket连接 ===== */

/**
 * 连接到指定网关
 */
fun connectToGateway(gateway: GatewayInfo) {
    changeState(GatewayConnectionState.CONNECTING)

    val wsUrl = "ws://${gateway.host}:${gateway.port}/ws/board"
    Log.i(TAG, "连接网关: $wsUrl")

    try {
        /* OkHttpClient.newWebSocket(
            Request.Builder().url(wsUrl).build(),
            createWebSocketListener()) */

        /* 模拟连接成功 */
        onWebSocketConnected(gateway)
    } catch (e: Exception) {
        Log.e(TAG, "连接网关失败", e)
        scheduleReconnect()
    }
}

/**
 * WebSocket连接成功
 */
private fun onWebSocketConnected(gateway: GatewayInfo) {
    currentGateway = gateway
    lastHeartbeatReceived = System.currentTimeMillis()
    reconnectAttempts.set(0)

    changeState(GatewayConnectionState.CONNECTED)

    /* 发送认证消息 */
    sendAuthMessage()

    /* 启动心跳 */
    startHeartbeat()
}

```

```

        Log.i(TAG, "已连接到网关: ${gateway.gatewayId}")
    }

    /**
     * 发送设备认证消息
     */
    private fun sendAuthMessage() {
        val auth = JSONObject().apply {
            put("type", "auth")
            put("device_type", "board")
            put("device_id", "BOARD-${System.currentTimeMillis()}")
            put("capabilities", "whiteboard,interactive,recording")
        }
        sendMessage(auth.toString())
    }

    /**
     * 发送WebSocket消息
     */
    fun sendMessage(message: String) {
        if (connectionState != GatewayConnectionState.CONNECTED) {
            Log.w(TAG, "未连接状态无法发送消息")
            return
        }
        /* ws.send(message) */
        Log.d(TAG, "发送消息: ${message.take(100)}...")
    }

    /**
     * 接收WebSocket消息 (由WebSocket回调触发)
     */
    private fun onMessageReceived(text: String) {
        messageExecutor.submit {
            try {
                val json = JSONObject(text)
                val type = json.optString("type", "")

                when (type) {
                    GatewayMessageType.HEARTBEAT -> {
                        lastHeartbeatReceived = System.currentTimeMillis()
                    }
                    GatewayMessageType.STATUS -> {
                        updateGatewayStatus(json)
                    }
                    else -> {
                        /* 分发给所有监听器 */
                        messageListeners.forEach { it.onGatewayMessage(type, json) }
                    }
                }
            } catch (e: Exception) {
                Log.e(TAG, "消息处理失败: ${e.message}")
            }
        }
    }

    /**
     * 更新网关状态信息

```

```

*/
private fun updateGatewayStatus(json: JSONObject) {
    currentGateway = currentGateway?.copy(
        onlinePenCount = json.optInt("online_pens", 0),
        firmwareVersion = json.optString("firmware", ""),
        signalStrength = json.optInt("wifi_rssi", 0),
        lastHeartbeat = System.currentTimeMillis()
    )
    Log.d(TAG, "网关状态更新: 在线笔=${currentGateway?.onlinePenCount}")
}

/* ===== 心跳与重连 ===== */

/**
 * 启动心跳定时器
 */
private fun startHeartbeat() {
    scheduler.scheduleAtFixedRate({
        if (connectionState == GatewayConnectionState.CONNECTED) {
            /* 发送心跳 */
            val hb = JSONObject().apply {
                put("type", "heartbeat")
                put("timestamp", System.currentTimeMillis())
            }
            sendMessage(hb.toString())

            /* 检查心跳超时 */
            if (System.currentTimeMillis() - lastHeartbeatReceived >
                HEARTBEAT_TIMEOUT_MS) {
                Log.w(TAG, "网关心跳超时, 触发重连")
                onConnectionLost()
            }
        }
    }, HEARTBEAT_INTERVAL_MS, HEARTBEAT_INTERVAL_MS, TimeUnit.MILLISECONDS)
}

/**
 * 连接丢失处理
 */
private fun onConnectionLost() {
    changeState(GatewayConnectionState.RECONNECTING)
    scheduleReconnect()
}

/**
 * 调度重连 (指数退避)
 */
private fun scheduleReconnect() {
    if (!isRunning.get()) return

    val attempt = reconnectAttempts.incrementAndGet()
    val delay = (RECONNECT_BASE_DELAY_MS * Math.pow(1.5,
        attempt.toDouble()).toLong())
        .coerceAtMost(RECONNECT_MAX_DELAY_MS)

    Log.i(TAG, "将在 ${delay}ms 后重连 (第${attempt}次)")
}

```

```

        scheduler.schedule({
            currentGateway?.let { connectToGateway(it) }
        }, delay, TimeUnit.MILLISECONDS)
    }

    /* ===== 课堂控制指令 ===== */

    /**
     * 发送课堂控制指令
     */
    fun sendClassroomCommand(command: String, params: Map<String, Any> = emptyMap()) {
        val msg = JSONObject().apply {
            put("type", "command")
            put("command", command)
            params.forEach { (k, v) -> put(k, v) }
            put("timestamp", System.currentTimeMillis())
        }
        sendMessage(msg.toString())
        Log.i(TAG, "发送课堂指令: $command")
    }

    /* ===== 状态管理 ===== */

    private fun changeState(newState: GatewayConnectionState) {
        connectionState = newState
        messageListeners.forEach { it.onGatewayStateChanged(newState, currentGateway) }
    }

    /**
     * 获取已发现的网关列表
     */
    fun getDiscoveredGateways(): List<GatewayInfo> = discoveredGateways.values.toList()

    /**
     * 停止并释放资源
     */
    fun shutdown() {
        isRunning.set(false)
        scheduler.shutdown()
        messageExecutor.shutdown()
        changeState(GatewayConnectionState.DISCONNECTED)
        Log.i(TAG, "网关连接器已关闭")
    }
}

```

recording/

recording/ScreenRecorder.kt

```

/**
 * 自然写互动课堂智慧黑板端应用软件 V1.0
 *
 * ScreenRecorder.kt - 课堂录制模块

```

```

*
* 功能说明:
* - 课堂屏幕录制 (MediaCodec H.264编码)
* - 音频同步录制 (AAC编码)
* - MediaMuxer封装MP4文件
* - 录制进度跟踪与时间限制
* - 录像文件管理 (存储/上传/清理)
* - 课堂回放支持
*/

package com.writech.board.recording

import android.content.Context
import android.media.*
import android.os.Environment
import android.util.Log
import android.view.Surface
import java.io.File
import java.nio.ByteBuffer
import java.text.SimpleDateFormat
import java.util.*
import java.util.concurrent.atomic.AtomicBoolean
import kotlin.concurrent.thread

/**
 * 录制状态
 */
enum class RecordingState {
    IDLE,          /* 空闲 */
    PREPARING,     /* 准备中 */
    RECORDING,     /* 录制中 */
    PAUSED,        /* 暂停 */
    STOPPING,     /* 停止中 */
    ERROR          /* 错误 */
}

/**
 * 录制配置参数
 */
data class RecordingConfig(
    val videoWidth: Int = 1920,          /* 视频宽度 */
    val videoHeight: Int = 1080,         /* 视频高度 */
    val videoBitrate: Int = 6_000_000,   /* 视频码率 6Mbps */
    val videoFps: Int = 30,              /* 帧率 30fps */
    val audioEnabled: Boolean = true,     /* 是否录制音频 */
    val audioBitrate: Int = 128_000,     /* 音频码率 128kbps */
    val audioSampleRate: Int = 44100,    /* 音频采样率 */
    val maxDurationSec: Int = 5400,      /* 最大录制时长 90分钟 */
    val outputDir: String = ""           /* 输出目录 */
)

/**
 * 录制结果信息
 */
data class RecordingResult(
    val filePath: String,                /* 录像文件路径 */
    val durationMs: Long,                 /* 录制时长 (毫秒) */

```



```

        val fileSize: Long,                /* 文件大小 (字节) */
        val videoWidth: Int,               /* 视频宽度 */
        val videoHeight: Int,              /* 视频高度 */
        val timestamp: Long = System.currentTimeMillis()
    )

/**
 * 录制事件回调
 */
interface RecordingListener {
    fun onRecordingStateChanged(state: RecordingState)
    fun onRecordingProgress(durationMs: Long)
    fun onRecordingCompleted(result: RecordingResult)
    fun onRecordingError(error: String)
}

/**
 * 课堂屏幕录制器
 *
 * 使用Android MediaCodec + MediaMuxer实现高效屏幕录制:
 * - 视频编码: H.264 (AVC), 1080p@30fps
 * - 音频编码: AAC-LC, 44.1kHz
 * - 容器格式: MP4 (MPEG-4 Part 14)
 */
class ScreenRecorder(private val context: Context) {

    companion object {
        private const val TAG = "ScreenRecorder"
        private const val VIDEO_MIME = MediaFormat.MIMETYPE_VIDEO_AVC
        private const val AUDIO_MIME = MediaFormat.MIMETYPE_AUDIO_AAC
        /** I帧间隔 (秒) */
        private const val IFRAME_INTERVAL = 2
        /** 编码器超时 (微秒) */
        private const val CODEC_TIMEOUT_US = 10000L
        /** 进度回调间隔 (毫秒) */
        private const val PROGRESS_INTERVAL_MS = 1000L
    }

    /** ===== 状态 ===== */

    /** 录制状态 */
    var state: RecordingState = RecordingState.IDLE
        private set

    /** 录制配置 */
    private var config = RecordingConfig()

    /** 是否正在录制 */
    private val isRecording = AtomicBoolean(false)

    /** 录制开始时间 */
    private var startTimeNs: Long = 0

    /** 暂停累计时间 */
    private var pausedDurationNs: Long = 0

    /** 暂停起始时间 */

```

```

private var pauseStartNs: Long = 0

/* ===== 编码器 ===== */

/** 视频编码器 */
private var videoEncoder: MediaCodec? = null
/** 音频编码器 */
private var audioEncoder: MediaCodec? = null
/** 混流器 */
private var mediaMuxer: MediaMuxer? = null
/** 视频输入Surface */
private var inputSurface: Surface? = null

/** 视频轨道索引 */
private var videoTrackIndex: Int = -1
/** 音频轨道索引 */
private var audioTrackIndex: Int = -1
/** Muxer是否已启动 */
private var isMuxerStarted = false
/** 已添加的轨道数 */
private var tracksAdded = 0

/** 输出文件路径 */
private var outputFilePath: String = ""

/* ===== 监听器 ===== */

/** 事件监听器 */
private var listener: RecordingListener? = null

/**
 * 设置录制事件监听器
 */
fun setListener(listener: RecordingListener) {
    this.listener = listener
}

/* ===== 录制控制 ===== */

/**
 * 开始录制
 *
 * @param config 录制配置
 * @return 视频输入Surface (渲染内容将被录制)
 */
fun startRecording(config: RecordingConfig = RecordingConfig()): Surface? {
    if (state != RecordingState.IDLE && state != RecordingState.ERROR) {
        Log.w(TAG, "无法启动录制, 当前状态=$state")
        return null
    }

    this.config = config
    changeState(RecordingState.PREPARING)

    try {
        /* 生成输出文件路径 */
        outputFilePath = generateOutputPath()
    }
}

```

```

        Log.i(TAG, "录制输出: $outputFilePath")

        /* 配置视频编码器 */
        setupVideoEncoder()

        /* 配置音频编码器 */
        if (config.audioEnabled) {
            setupAudioEncoder()
        }

        /* 创建MediaMuxer */
        mediaMuxer = MediaMuxer(outputFilePath,
MediaMuxer.OutputFormat.MUXER_OUTPUT_MPEG_4)

        /* 启动编码器 */
        videoEncoder?.start()
        audioEncoder?.start()

        /* 获取视频输入Surface */
        inputSurface = videoEncoder?.createInputSurface()

        isRecording.set(true)
        startTimeNs = System.nanoTime()
        pausedDurationNs = 0

        /* 启动编码线程 */
        startEncodingThreads()

        changeState(RecordingState.RECORDING)
        Log.i(TAG, "录制开始: ${config.videoWidth}x${config.videoHeight} " +
            "@${config.videoFps}fps, 码率=${config.videoBitrate} /
1_000_000}Mbps")

        return inputSurface

    } catch (e: Exception) {
        Log.e(TAG, "启动录制失败", e)
        changeState(RecordingState.ERROR)
        listener?.onRecordingError("启动录制失败: ${e.message}")
        releaseResources()
        return null
    }
}

/**
 * 暂停录制
 */
fun pauseRecording() {
    if (state != RecordingState.RECORDING) return

    pauseStartNs = System.nanoTime()
    changeState(RecordingState.PAUSED)
    Log.i(TAG, "录制已暂停")
}

/**
 * 恢复录制

```

```

    */
fun resumeRecording() {
    if (state != RecordingState.PAUSED) return

    pausedDurationNs += System.nanoTime() - pauseStartNs
    changeState(RecordingState.RECORDING)
    Log.i(TAG, "录制已恢复")
}

/**
 * 停止录制
 */
fun stopRecording() {
    if (state != RecordingState.RECORDING && state != RecordingState.PAUSED) {
        Log.w(TAG, "非录制状态无法停止")
        return
    }

    changeState(RecordingState.STOPPING)
    isRecording.set(false)

    Log.i(TAG, "停止录制中...")

    /* 等待编码线程结束后再释放资源（在编码线程中处理） */
}

/* ===== 编码器配置 ===== */

/**
 * 配置视频编码器（H.264）
 */
private fun setupVideoEncoder() {
    val format = MediaFormat.createVideoFormat(VIDEO_MIME, config.videoWidth,
config.videoHeight)
    format.setInteger(MediaFormat.KEY_COLOR_FORMAT,
        MediaCodecInfo.CodecCapabilities.COLOR_FormatSurface)
    format.setInteger(MediaFormat.KEY_BIT_RATE, config.videoBitrate)
    format.setInteger(MediaFormat.KEY_FRAME_RATE, config.videoFps)
    format.setInteger(MediaFormat.KEY_I_FRAME_INTERVAL, IFRAME_INTERVAL)

    /* 设置编码Profile为High，提升压缩效率 */
    format.setInteger(MediaFormat.KEY_PROFILE,
        MediaCodecInfo.CodecProfileLevel.AVCProfileHigh)
    format.setInteger(MediaFormat.KEY_LEVEL,
        MediaCodecInfo.CodecProfileLevel.AVCLevel41)

    videoEncoder = MediaCodec.createEncoderByType(VIDEO_MIME)
    videoEncoder?.configure(format, null, null, MediaCodec.CONFIGURE_FLAG_ENCODE)

    Log.d(TAG, "视频编码器配置: ${config.videoWidth}x${config.videoHeight}, " +
        "码率=${config.videoBitrate}, 帧率=${config.videoFps}")
}

/**
 * 配置音频编码器（AAC-LC）
 */
private fun setupAudioEncoder() {

```

```

        val format = MediaFormat.createAudioFormat(AUDIO_MIME,
            config.audioSampleRate, 1)
        format.setInteger(MediaFormat.KEY_BIT_RATE, config.audioBitrate)
        format.setInteger(MediaFormat.KEY_AAC_PROFILE,
            MediaCodecInfo.CodecProfileLevel.AACObjectLC)
        format.setInteger(MediaFormat.KEY_MAX_INPUT_SIZE, 16384)

        audioEncoder = MediaCodec.createEncoderByType(AUDIO_MIME)
        audioEncoder?.configure(format, null, null, MediaCodec.CONFIGURE_FLAG_ENCODE)

        Log.d(TAG, "音频编码器配置: ${config.audioSampleRate}Hz, " +
            "码率=${config.audioBitrate}")
    }

    /* ===== 编码线程 ===== */

    /**
     * 启动编码线程
     */
    private fun startEncodingThreads() {
        /* 视频编码线程 */
        thread(name = "VideoEncoder") {
            drainEncoder(videoEncoder, true)
        }

        /* 音频编码线程 */
        if (config.audioEnabled) {
            thread(name = "AudioEncoder") {
                drainEncoder(audioEncoder, false)
            }
        }

        /* 进度回调线程 */
        thread(name = "RecordingProgress") {
            while (isRecording.get()) {
                if (state == RecordingState.RECORDING) {
                    val elapsed = (System.nanoTime() - startTimeNs - pausedDurationNs) /
1_000_000

                    listener?.onRecordingProgress(elapsed)

                    /* 检查最大时长限制 */
                    if (elapsed > config.maxDurationSec * 1000L) {
                        Log.i(TAG, "达到最大录制时长 ${config.maxDurationSec}秒, 自动停止")
                        stopRecording()
                    }
                }
                Thread.sleep(PROGRESS_INTERVAL_MS)
            }
        }
    }

    /**
     * 从编码器中取出编码后的数据并写入Muxer
     */
    private fun drainEncoder(encoder: MediaCodec?, isVideo: Boolean) {
        if (encoder == null) return

```

```

        val bufferInfo = MediaCodec.BufferInfo()
        val encoderName = if (isVideo) "视频" else "音频"

        try {
            while (isRecording.get() || true) {
                val outputIndex = encoder.dequeueOutputBuffer(bufferInfo,
CODEC_TIMEOUT_US)

                when {
                    outputIndex == MediaCodec.INFO_OUTPUT_FORMAT_CHANGED -> {
                        /* 添加轨道到Muxer */
                        val format = encoder.outputFormat
                        synchronized(this) {
                            if (isVideo) {
                                videoTrackIndex = mediaMuxer?.addTrack(format) ?: -1
                                Log.d(TAG, "${encoderName}轨道添加:
index=$videoTrackIndex")
                            } else {
                                audioTrackIndex = mediaMuxer?.addTrack(format) ?: -1
                                Log.d(TAG, "${encoderName}轨道添加:
index=$audioTrackIndex")
                            }
                            tracksAdded++

                            /* 所有轨道就绪后启动Muxer */
                            val expectedTracks = if (config.audioEnabled) 2 else 1
                            if (tracksAdded >= expectedTracks && !isMuxerStarted) {
                                mediaMuxer?.start()
                                isMuxerStarted = true
                                Log.i(TAG, "MediaMuxer已启动")
                            }
                        }
                    }
                }
                outputIndex >= 0 -> {
                    val buffer = encoder.getOutputBuffer(outputIndex) ?: continue

                    if (bufferInfo.flags and MediaCodec.BUFFER_FLAG_CODEC_CONFIG !=
0) {
                        bufferInfo.size = 0
                    }

                    if (bufferInfo.size > 0 && isMuxerStarted) {
                        val trackIndex = if (isVideo) videoTrackIndex else
audioTrackIndex
                        synchronized(this) {
                            mediaMuxer?.writeSampleData(trackIndex, buffer,
bufferInfo)
                        }
                    }

                    encoder.releaseOutputBuffer(outputIndex, false)

                    /* 检查结束标志 */
                    if (bufferInfo.flags and MediaCodec.BUFFER_FLAG_END_OF_STREAM !=
0) {
                        Log.d(TAG, "${encoderName}编码结束")
                        break
                    }
                }
            }
        }
    }
}

```

```

        }
    }

    if (!isRecording.get()) {
        encoder.signalEndOfInputStream()
    }
}

} catch (e: Exception) {
    Log.e(TAG, "${encoderName}编码异常", e)
} finally {
    if (isVideo) {
        /* 视频编码完成后释放资源 */
        onEncodingFinished()
    }
}
}

/**
 * 编码完成后的清理工作
 */
private fun onEncodingFinished() {
    val durationMs = (System.nanoTime() - startTimeNs - pausedDurationNs) /
1_000_000

    releaseResources()

    /* 获取文件大小 */
    val file = File(outputFilePath)
    val fileSize = if (file.exists()) file.length() else 0

    val result = RecordingResult(
        filePath = outputFilePath,
        durationMs = durationMs,
        fileSize = fileSize,
        videoWidth = config.videoWidth,
        videoHeight = config.videoHeight
    )

    changeState(RecordingState.IDLE)
    listener?.onRecordingCompleted(result)

    Log.i(TAG, "录制完成: 时长=${durationMs / 1000}秒, " +
        "文件大小=${fileSize / 1024}KB, 路径=$outputFilePath")
}

/* ===== 资源管理 ===== */

/**
 * 释放所有资源
 */
private fun releaseResources() {
    try {
        videoEncoder?.stop()
        videoEncoder?.release()
        videoEncoder = null
    } catch (e: Exception) { /* 忽略 */ }
}

```

```

        try {
            audioEncoder?.stop()
            audioEncoder?.release()
            audioEncoder = null
        } catch (e: Exception) { /* 忽略 */ }

        try {
            if (isMuxerStarted) {
                mediaMuxer?.stop()
            }
            mediaMuxer?.release()
            mediaMuxer = null
        } catch (e: Exception) { /* 忽略 */ }

        inputSurface?.release()
        inputSurface = null

        isMuxerStarted = false
        tracksAdded = 0
        videoTrackIndex = -1
        audioTrackIndex = -1

        Log.d(TAG, "录制资源已释放")
    }

    /**
     * 生成录像文件输出路径
     */
    private fun generateOutputPath(): String {
        val dir = if (config.outputDir.isNotEmpty()) {
            File(config.outputDir)
        } else {
            File(context.filesDir, "recordings")
        }
        if (!dir.exists()) dir.mkdirs()

        val dateFormat = SimpleDateFormat("yyyyMMdd_HH:mm:ss", Locale.CHINA)
        val fileName = "class_${dateFormat.format(Date())}.mp4"
        return File(dir, fileName).absolutePath
    }

    /**
     * 状态变更
     */
    private fun changeState(newState: RecordingState) {
        state = newState
        listener?.onRecordingStateChanged(newState)
    }
}

```

ui/

ui/InteractiveActivity.kt


```

/**
 * 自然写互动课堂智慧黑板端应用软件 V1.0
 *
 * InteractiveActivity.kt - 课堂互动答题系统
 *
 * 功能说明:
 * - 发布互动题目 (选择/填空/简答/判断)
 * - 实时收集学生答案
 * - 答题统计与结果展示
 * - 随机抽取与分组展示
 * - 倒计时控制
 * - 答题数据持久化
 */

package com.writech.board.ui

import android.content.Context
import android.os.Bundle
import android.os.CountDownTimer
import android.util.Log
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import java.util.concurrent.ConcurrentHashMap
import java.util.concurrent.CopyOnWriteArrayList
import kotlin.random.Random

/**
 * 题目类型枚举
 */
enum class QuestionType(val code: Int, val label: String) {
    SINGLE_CHOICE(1, "单选"),
    MULTIPLE_CHOICE(2, "多选"),
    TRUE_FALSE(3, "判断"),
    FILL_BLANK(4, "填空"),
    SHORT_ANSWER(5, "简答")
}

/**
 * 互动题目数据
 */
data class InteractiveQuestion(
    val questionId: String,
    val type: QuestionType,
    val title: String,
    val options: List<String> = emptyList(), /* 选择题选项 */
    val correctAnswer: String = "", /* 正确答案 */
    val timeLimit: Int = 60, /* 答题时限(秒) */
    val score: Int = 10 /* 题目分值 */
)

/**
 * 学生答案数据
 */
data class StudentAnswer(
    val studentId: String,

```

```

        val studentName: String,
        val questionId: String,
        val answer: String,
        val isCorrect: Boolean = false,
        val submitTime: Long = System.currentTimeMillis(),
        val costSeconds: Int = 0    /* 答题耗时(秒) */
    )

/**
 * 答题统计结果
 */
data class AnswerStatistics(
    val questionId: String,
    val totalStudents: Int,          /* 班级总人数 */
    val submittedCount: Int,         /* 已提交人数 */
    val correctCount: Int,           /* 正确人数 */
    val correctRate: Float,          /* 正确率 */
    val optionDistribution: Map<String, Int>, /* 各选项分布 */
    val avgCostSeconds: Float        /* 平均耗时 */
)

/**
 * 互动答题会话状态
 */
enum class SessionState {
    IDLE,          /* 空闲 */
    PUBLISHING,    /* 发题中 */
    ANSWERING,     /* 答题中 */
    COLLECTING,    /* 收卷中 */
    REVIEWING      /* 查看结果 */
}

/**
 * 互动答题系统事件监听
 */
interface InteractiveListener {
    fun onSessionStateChanged(state: SessionState)
    fun onAnswerReceived(answer: StudentAnswer)
    fun onCountdownTick(remainSeconds: Int)
    fun onCountdownFinished()
    fun onStatisticsReady(stats: AnswerStatistics)
}

/**
 * 课堂互动答题系统
 *
 * 管理整个互动答题流程：
 * 教师出题 → 发布题目 → 学生作答 → 收卷 → 统计展示
 */
class InteractiveManager(
    private val classroomId: String,
    private val totalStudents: Int
) {

    companion object {
        private const val TAG = "Interactive"
    }
}

```

```

/* ===== 状态管理 ===== */

/** 当前会话状态 */
var state: SessionState = SessionState.IDLE
    private set

/** 当前题目 */
private var currentQuestion: InteractiveQuestion? = null

/** 学生答案收集: studentId → StudentAnswer */
private val answersMap = ConcurrentHashMap<String, StudentAnswer>()

/** 事件监听器 */
private val listeners = CopyOnWriteArrayList<InteractiveListener>()

/** 倒计时器 */
private var countdownTimer: CountDownTimer? = null

/** 发题时间戳（用于计算学生耗时） */
private var publishTimestamp: Long = 0

/** 历史题目记录 */
private val questionHistory = mutableListOf<InteractiveQuestion>()

/** 历史统计记录 */
private val statisticsHistory = mutableListOf<AnswerStatistics>()

/**
 * 添加事件监听器
 */
fun addListener(listener: InteractiveListener) {
    listeners.add(listener)
}

/* ===== 发题流程 ===== */

/**
 * 发布互动题目
 * 将题目推送给全班学生
 *
 * @param question 题目数据
 * @return true=发布成功
 */
fun publishQuestion(question: InteractiveQuestion): Boolean {
    if (state != SessionState.IDLE && state != SessionState.REVIEWING) {
        Log.w(TAG, "当前状态不允许发题: $state")
        return false
    }

    currentQuestion = question
    answersMap.clear()
    publishTimestamp = System.currentTimeMillis()

    /* 切换状态为发题中 */
    changeState(SessionState.PUBLISHING)

```

```

    /* 构建发题消息通过WebSocket推送给学生 */
    val msg = buildQuestionMessage(question)
    Log.i(TAG, "发布题目: ${question.type.label} - ${question.title}")
    Log.d(TAG, "推送消息: $msg")

    /* ws.send(msg) - 通过WebSocket推送给网关 */

    /* 切换到答题中状态 */
    changeState(SessionState.ANSWERING)

    /* 启动倒计时 */
    startCountdown(question.timeLimit)

    questionHistory.add(question)
    return true
}

/**
 * 构建题目消息JSON
 */
private fun buildQuestionMessage(question: InteractiveQuestion): String {
    val sb = StringBuilder()
    sb.append("{")
    sb.append("\"type\": \"question\",")
    sb.append("\"classroom_id\": \"$classroomId\",")
    sb.append("\"question_id\": \"${question.questionId}\",")
    sb.append("\"question_type\": ${question.type.code},")
    sb.append("\"title\": \"${question.title}\",")

    if (question.options.isNotEmpty()) {
        sb.append("\"options\": [")
        question.options.forEachIndexed { index, opt ->
            if (index > 0) sb.append(",")
            sb.append("\"$opt\"")
        }
        sb.append("],")
    }

    sb.append("\"time_limit\": ${question.timeLimit},")
    sb.append("\"score\": ${question.score},")
    sb.append("\"timestamp\": ${System.currentTimeMillis()}")
    sb.append("}")

    return sb.toString()
}

/* ===== 答案收集 ===== */

/**
 * 接收学生提交的答案
 * 通常由WebSocket消息回调触发
 */
fun onStudentAnswerReceived(studentId: String, studentName: String,
                           answer: String) {
    if (state != SessionState.ANSWERING && state != SessionState.COLLECTING) {
        Log.w(TAG, "非答题状态收到答案, 忽略: student=$studentId")
        return
    }
}

```

```

    }

    val question = currentQuestion ?: return

    /* 判断答案是否正确 */
    val isCorrect = when (question.type) {
        QuestionType.SINGLE_CHOICE,
        QuestionType.TRUE_FALSE ->
answer.trim().equals(question.correctAnswer.trim(), true)
        QuestionType.MULTIPLE_CHOICE -> {
            val submitted = answer.split(",").map { it.trim() }.sorted()
            val correct = question.correctAnswer.split(",").map { it.trim()
}.sorted()
                submitted == correct
            }
        else -> false /* 填空题和简答题需人工批改 */
    }

    /* 计算答题耗时 */
    val costSec = ((System.currentTimeMillis() - publishTimestamp) / 1000).toInt()

    val studentAnswer = StudentAnswer(
        studentId = studentId,
        studentName = studentName,
        questionId = question.questionId,
        answer = answer,
        isCorrect = isCorrect,
        costSeconds = costSec
    )

    answersMap[studentId] = studentAnswer

    /* 通知监听器 */
    listeners.forEach { it.onAnswerReceived(studentAnswer) }

    Log.d(TAG, "收到答案: $studentName ($studentId) = $answer, " +
        "正确=$isCorrect, 耗时=${costSec}s, " +
        "进度=${answersMap.size}/${totalStudents}")

    /* 检查是否全部提交 */
    if (answersMap.size >= totalStudents) {
        Log.i(TAG, "全部学生已提交, 自动收卷")
        collectAnswers()
    }
}

/* ===== 收卷与统计 ===== */

/**
 * 手动收卷 (教师点击收卷按钮)
 */
fun collectAnswers() {
    if (state != SessionState.ANSWERING) {
        Log.w(TAG, "非答题状态无法收卷")
        return
    }
}

```

```

/* 停止倒计时 */
countdownTimer?.cancel()

changeState(SessionState.COLLECTING)

/* 发送收卷指令给学生端 */
/* ws.send("{\"type\":\"collect\",\"question_id\":\"...\"}") */

Log.i(TAG, "收卷完成: 已提交=${answersMap.size}/${totalStudents}")

/* 生成统计结果 */
val stats = generateStatistics()
statisticsHistory.add(stats)

/* 切换到查看结果状态 */
changeState(SessionState.REVIEWING)

listeners.forEach { it.onStatisticsReady(stats) }
}

/**
 * 生成答题统计结果
 */
private fun generateStatistics(): AnswerStatistics {
    val question = currentQuestion ?: return AnswerStatistics(
        "", totalStudents, 0, 0, 0f, emptyMap(), 0f
    )

    val answers = answersMap.values.toList()
    val correctCount = answers.count { it.isCorrect }
    val correctRate = if (answers.isNotEmpty()) {
        correctCount.toFloat() / answers.size
    } else 0f

    val avgCost = if (answers.isNotEmpty()) {
        answers.map { it.costSeconds }.average().toFloat()
    } else 0f

    /* 统计各选项分布（选择题） */
    val distribution = mutableMapOf<String, Int>()
    if (question.type == QuestionType.SINGLE_CHOICE ||
        question.type == QuestionType.TRUE_FALSE) {
        answers.forEach { ans ->
            distribution[ans.answer] = (distribution[ans.answer] ?: 0) + 1
        }
    }

    val stats = AnswerStatistics(
        questionId = question.questionId,
        totalStudents = totalStudents,
        submittedCount = answers.size,
        correctCount = correctCount,
        correctRate = correctRate,
        optionDistribution = distribution,
        avgCostSeconds = avgCost
    )
}

```

```

        Log.i(TAG, "统计结果: 提交${answers.size}/${totalStudents}, " +
            "正确率=${String.format("%.1f", correctRate * 100)}%, " +
            "平均耗时=${String.format("%.1f", avgCost)}s")

        return stats
    }

    /* ===== 随机抽取 ===== */

    /**
     * 随机抽取指定数量的学生
     * 用于课堂随机点名展示
     */
    fun randomPickStudents(count: Int): List<String> {
        val allStudents = answersMap.keys.toList()
        if (allStudents.size <= count) return allStudents

        return allStudents.shuffled(Random(System.currentTimeMillis())).take(count).also {
            Log.i(TAG, "随机抽取$count名学生: $it")
        }
    }

    /**
     * 按分组展示学生答案
     * @param groupSize 每组人数
     */
    fun groupStudents(groupSize: Int): List<List<StudentAnswer>> {
        val answers = answersMap.values.toList()
        return answers.chunked(groupSize).also {
            Log.i(TAG, "分组展示: ${it.size}组, 每组$groupSize人")
        }
    }

    /* ===== 倒计时 ===== */

    /**
     * 启动答题倒计时
     */
    private fun startCountdown(seconds: Int) {
        countdownTimer?.cancel()

        countdownTimer = object : CountDownTimer(seconds * 1000L, 1000) {
            override fun onTick(millisUntilFinished: Long) {
                val remain = (millisUntilFinished / 1000).toInt()
                listeners.forEach { it.onCountdownTick(remain) }
            }

            override fun onFinish() {
                Log.i(TAG, "答题时间到")
                listeners.forEach { it.onCountdownFinished() }
                collectAnswers()
            }
        }.start()

        Log.i(TAG, "倒计时启动: ${seconds}秒")
    }
}

```

```

/* ===== 状态管理 ===== */

/**
 * 变更会话状态
 */
private fun changeState(newState: SessionState) {
    val oldState = state
    state = newState
    Log.d(TAG, "状态变更: $oldState → $newState")
    listeners.forEach { it.onSessionStateChanged(newState) }
}

/**
 * 重置为空闲状态
 */
fun reset() {
    countdownTimer?.cancel()
    answersMap.clear()
    currentQuestion = null
    changeState(SessionState.IDLE)
    Log.i(TAG, "互动系统已重置")
}

/**
 * 获取当前提交进度（已提交/总人数）
 */
fun getProgress(): Pair<Int, Int> = Pair(answersMap.size, totalStudents)

/**
 * 获取历史统计记录
 */
fun getHistoryStatistics(): List<AnswerStatistics> = statisticsHistory.toList()
}

```