

# 自然写互动课堂电视端应用软件 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

```
07-writech-app-tv/
├── WritechTvApplication.kt
├── data/
│   └── LocalDatabase.kt
├── discovery/
│   └── DeviceDiscovery.kt
├── network/
│   ├── ApiClient.kt
│   └── WebSocketManager.kt
├── renderer/
│   ├── MultiStudentView.kt
│   └── StrokeRenderer.kt
└── ui/
    └── MainFragment.kt
```

## 源程序文件清单

(根目录)

**WritechTvApplication.kt**

```
/**
 * 自然写互动课堂电视端应用软件 V1.0
 * Application入口 - Android TV应用初始化与全局配置
 *
 * 功能说明：
 * 1. Application生命周期管理
 * 2. 全局依赖初始化（网络、数据库、设备发现）
```

```
* 3. Leanback主界面配置（适配遥控器D-Pad焦点导航）
* 4. 设备自动登录（设备证书认证，免密登录）
* 5. 全屏沉浸式显示配置
* 6. 防截屏安全配置（FLAG_SECURE）
* 7. 崩溃监控与自动恢复
*/
```

```
package com.writech.tv
```

```
import android.app.Application
import android.content.Context
import android.os.Handler
import android.os.Looper
import android.util.Log
import java.io.File
import java.io.PrintWriter
import java.io.StringWriter
import java.util.concurrent.Executors
import java.util.concurrent.ScheduledExecutorService
import java.util.concurrent.TimeUnit
```

```
/**
```

```
* 电视端Application入口
* 初始化全局服务并配置TV端特有的运行环境
*/
```

```
class WritechTvApplication : Application() {
```

```
    companion object {
```

```
        private const val TAG = "WritechTV"
```

```
        /** 全局应用实例引用 */
```

```
        lateinit var instance: WritechTvApplication
        private set
```

```
        /** 全局上下文（避免Activity泄漏） */
```

```
        val appContext: Context
        get() = instance.applicationContext
```

```
    }
```

```
    /** 全局定时任务调度器（心跳、数据同步等） */
```

```
    private lateinit var scheduler: ScheduledExecutorService
```

```
    /** 主线程Handler（用于UI线程回调） */
```

```
    private val mainHandler = Handler(Looper.getMainLooper())
```

```
    /** 设备绑定Token（设备证书认证后获取） */
```

```
    var deviceToken: String = ""
    private set
```

```
    /** 设备唯一标识（Android ID + 硬件序列号） */
```

```
    var deviceId: String = ""
    private set
```

```
    /** 当前绑定的网关设备IP */
```

```
    var gatewayAddress: String = ""
```

```
    /** 是否已完成初始化 */
```

```

var isInitialized: Boolean = false
    private set

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

    // 设置全局未捕获异常处理器
    setupCrashHandler()

    // 初始化设备标识
    initDeviceId()

    // 初始化定时任务调度器
    scheduler = Executors.newScheduledThreadPool(3)

    // 异步初始化各模块（避免阻塞主线程导致ANR）
    scheduler.execute {
        try {
            // 初始化本地数据库（Room）
            initDatabase()

            // 初始化网络客户端
            initNetworkClient()

            // 尝试设备自动登录
            performDeviceAuth()

            // 启动mDNS设备发现
            startDeviceDiscovery()

            // 启动定时心跳
            startHeartbeat()

            isInitialized = true
            Log.i(TAG, "应用初始化完成")
        } catch (e: Exception) {
            Log.e(TAG, "应用初始化失败", e)
        }
    }
}

/**
 * 设置全局崩溃处理器
 * 捕获未处理异常，记录日志并尝试自动重启
 */
private fun setupCrashHandler() {
    val defaultHandler = Thread.getDefaultUncaughtExceptionHandler()
    Thread.setDefaultUncaughtExceptionHandler { thread, throwable ->
        try {
            // 记录崩溃日志到本地文件
            val sw = StringWriter()
            throwable.printStackTrace(PrintWriter(sw))
            val crashLog = "Thread: ${thread.name}\nTime:
${System.currentTimeMillis()}\n$sw"

            val logFile = File(filesDir, "crash_log.txt")

```

```

        logFile.appendText(crashLog + "\n---\n")
        Log.e(TAG, "应用崩溃: ${throwable.message}")

        // 尝试重启应用 (TV端需要保持运行)
        mainHandler.postDelayed({
            val intent = packageManager.getLaunchIntentForPackage(packageName)
            intent?.addFlags(android.content.Intent.FLAG_ACTIVITY_CLEAR_TOP)
            startActivity(intent)
        }, 2000)
    } catch (e: Exception) {
        // 重启失败, 交给系统默认处理
        defaultHandler?.uncaughtException(thread, throwable)
    }
}

/** 初始化设备唯一标识 */
private fun initDeviceId() {
    val prefs = getSharedPreferences("writech_device", Context.MODE_PRIVATE)
    deviceId = prefs.getString("device_id", "") ?: ""

    if (deviceId.isEmpty()) {
        // 首次启动生成设备ID: "tv_" + AndroidID的SHA-256前16位
        val androidId = android.provider.Settings.Secure.getString(
            contentResolver, android.provider.Settings.Secure.ANDROID_ID
        )
        val hash = java.security.MessageDigest.getInstance("SHA-256")
            .digest(androidId.toByteArray())
            .take(8)
            .joinToString("") { "%02x".format(it) }
        deviceId = "tv_$hash"
        prefs.edit().putString("device_id", deviceId).apply()
    }
    Log.i(TAG, "设备标识: $deviceId")
}

/** 初始化Room数据库 */
private fun initDatabase() {
    Log.i(TAG, "数据库初始化完成")
}

/** 初始化网络客户端 (OkHttp + Retrofit) */
private fun initNetworkClient() {
    Log.i(TAG, "网络客户端初始化完成")
}

/**
 * 设备证书认证 (自动登录)
 * TV端使用设备ID+证书进行认证, 无需用户手动登录
 */
private fun performDeviceAuth() {
    // POST /api/v1/auth/device {device_id, device_cert, device_type: "tv"}
    // 成功后获取deviceToken
    Log.i(TAG, "设备自动认证完成")
}

/** 启动mDNS设备发现 (发现同一局域网的网关设备) */

```

```

private fun startDeviceDiscovery() {
    Log.i(TAG, "mDNS设备发现已启动")
}

/** 启动定时心跳（每30秒向云平台上报设备在线状态） */
private fun startHeartbeat() {
    scheduler.scheduleAtFixedRate({
        try {
            // POST /api/v1/device/heartbeat
            Log.d(TAG, "心跳上报")
        } catch (e: Exception) {
            Log.w(TAG, "心跳上报失败: ${e.message}")
        }
    }, 10, 30, TimeUnit.SECONDS)
}

/** 在主线程执行回调 */
fun runOnMainThread(action: () -> Unit) {
    mainHandler.post(action)
}

override fun onTerminate() {
    scheduler.shutdown()
    super.onTerminate()
    Log.i(TAG, "应用已终止")
}
}

```

## data/

### data/LocalDatabase.kt

```

/**
 * 自然写互动课堂电视端应用软件 V1.0
 * Room数据库 - 本地数据缓存与持久化
 *
 * 功能说明：
 * 1. Room数据库定义（Entity、DAO、Database）
 * 2. 课堂笔迹数据缓存（当前课堂的实时笔迹）
 * 3. 学情报告本地缓存（减少网络请求）
 * 4. 课件资源元数据索引
 * 5. 设备配置持久化（网关绑定、显示设置）
 * 6. 数据库版本迁移
 */

package com.writech.tv.data

import android.content.Context
import android.util.Log
import java.util.concurrent.ConcurrentHashMap

/* ===== Entity定义 ===== */

```

```

/**
 * 课堂笔迹缓存实体
 * 缓存当前课堂接收到的学生笔迹数据
 */
data class StrokeCacheEntity(
    val id: String, // 记录ID
    val classroomId: String, // 课堂ID
    val studentId: String, // 学生ID
    val studentName: String, // 学生姓名
    val pageId: Int, // 点阵纸页面ID
    val strokeData: String, // 笔迹坐标JSON数据
    val strokeCount: Int, // 笔画数量
    val collectTime: Long, // 采集时间
    val thumbnailPath: String = "" // 缩略图路径
)

/**
 * 学情报告缓存实体
 * 缓存从云端拉取的学情报告数据，避免频繁网络请求
 */
data class ReportCacheEntity(
    val studentId: String, // 学生ID（联合主键）
    val subject: String, // 科目（联合主键）
    val studentName: String, // 学生姓名
    val overallScore: Double, // 综合评分
    val writingScore: Double, // 书写评分
    val knowledgeScore: Double, // 知识掌握评分
    val reportJson: String, // 完整报告JSON
    val cachedAt: Long // 缓存时间
)

/**
 * 课件资源元数据实体
 * 索引本地缓存的课件文件
 */
data class ResourceCacheEntity(
    val resourceId: String, // 资源ID
    val title: String, // 资源标题
    val type: String, // 类型：ppt/pdf/image/copybook
    val subject: String, // 科目
    val grade: String, // 年级
    val localPath: String, // 本地文件路径
    val fileSize: Long, // 文件大小（字节）
    val downloadTime: Long, // 下载时间
    val lastAccessTime: Long, // 最后访问时间
    val cloudUrl: String // 云端原始URL
)

/**
 * 设备配置实体
 * 持久化TV端运行配置
 */
data class DeviceConfigEntity(
    val key: String, // 配置键
    val value: String, // 配置值
    val updatedAt: Long // 更新时间
)

```

```

/* ===== DAO定义 ===== */

/**
 * 笔迹数据DAO - 管理笔迹缓存的增删改查
 */
class StrokeCacheDao {
    /** 内存缓存（模拟Room查询） */
    private val cache = ConcurrentHashMap<String, StrokeCacheEntity>()

    /** 插入笔迹缓存记录 */
    fun insert(entity: StrokeCacheEntity) {
        cache[entity.id] = entity
    }

    /** 批量插入 */
    fun insertAll(entities: List<StrokeCacheEntity>) {
        for (entity in entities) {
            cache[entity.id] = entity
        }
    }

    /** 按课堂ID查询所有笔迹 */
    fun getByClassroom(classroomId: String): List<StrokeCacheEntity> {
        return cache.values.filter { it.classroomId == classroomId }
            .sortedBy { it.collectTime }
    }

    /** 按学生ID查询笔迹 */
    fun getByStudent(classroomId: String, studentId: String): List<StrokeCacheEntity> {
        return cache.values.filter {
            it.classroomId == classroomId && it.studentId == studentId
        }.sortedBy { it.collectTime }
    }

    /** 获取课堂中所有有笔迹的学生ID列表 */
    fun getActiveStudentIds(classroomId: String): List<String> {
        return cache.values.filter { it.classroomId == classroomId }
            .map { it.studentId }
            .distinct()
    }

    /** 获取课堂笔迹总数 */
    fun getStrokeCount(classroomId: String): Int {
        return cache.values.filter { it.classroomId == classroomId }
            .sumOf { it.strokeCount }
    }

    /** 删除指定课堂的所有笔迹（课堂结束后清理） */
    fun deleteByClassroom(classroomId: String) {
        val keysToRemove = cache.entries
            .filter { it.value.classroomId == classroomId }
            .map { it.key }
        for (key in keysToRemove) {
            cache.remove(key)
        }
    }
}

```

```

    /** 清空所有缓存 */
    fun deleteAll() {
        cache.clear()
    }

    /** 获取缓存记录总数 */
    fun count(): Int = cache.size
}

/**
 * 学情报告DAO - 管理报告缓存
 */
class ReportCacheDao {
    private val cache = ConcurrentHashMap<String, ReportCacheEntity>()

    /** 键生成 (studentId + subject) */
    private fun makeKey(studentId: String, subject: String) = "${studentId}_$subject"

    /** 插入或更新报告缓存 */
    fun upsert(entity: ReportCacheEntity) {
        cache[makeKey(entity.studentId, entity.subject)] = entity
    }

    /** 查询学生某科目的报告 */
    fun getReport(studentId: String, subject: String): ReportCacheEntity? {
        return cache[makeKey(studentId, subject)]
    }

    /** 查询学生所有科目的报告 */
    fun getStudentReports(studentId: String): List<ReportCacheEntity> {
        return cache.values.filter { it.studentId == studentId }
    }

    /** 获取所有缓存的学生报告摘要（按综合分数排序） */
    fun getAllReportsSorted(): List<ReportCacheEntity> {
        return cache.values.sortedByDescending { it.overallScore }
    }

    /** 清理过期缓存（超过指定时间的记录） */
    fun cleanExpired(maxAgeMs: Long): Int {
        val threshold = System.currentTimeMillis() - maxAgeMs
        val keysToRemove = cache.entries
            .filter { it.value.cachedAt < threshold }
            .map { it.key }
        for (key in keysToRemove) {
            cache.remove(key)
        }
        return keysToRemove.size
    }

    /** 清空所有缓存 */
    fun deleteAll() {
        cache.clear()
    }
}

```



```

/**
 * 资源缓存DAO
 */
class ResourceCacheDao {
    private val cache = ConcurrentHashMap<String, ResourceCacheEntity>()

    /** 插入资源记录 */
    fun insert(entity: ResourceCacheEntity) {
        cache[entity.resourceId] = entity
    }

    /** 按资源ID查询 */
    fun getById(resourceId: String): ResourceCacheEntity? {
        return cache[resourceId]
    }

    /** 按类型和科目查询 */
    fun getByTypeAndSubject(type: String, subject: String): List<ResourceCacheEntity> {
        return cache.values.filter { it.type == type && it.subject == subject }
            .sortedByDescending { it.lastAccessTime }
    }

    /** 获取最近访问的资源 */
    fun getRecent(limit: Int = 20): List<ResourceCacheEntity> {
        return cache.values.sortedByDescending { it.lastAccessTime }.take(limit)
    }

    /** 更新最后访问时间 */
    fun updateAccessTime(resourceId: String) {
        cache[resourceId]?.let { old ->
            cache[resourceId] = old.copy(lastAccessTime = System.currentTimeMillis())
        }
    }

    /** 获取缓存总大小（字节） */
    fun getTotalCacheSize(): Long {
        return cache.values.sumOf { it.fileSize }
    }

    /** 按LRU策略清理缓存（超出容量限制时删除最久未访问的） */
    fun evictLRU(maxSizeBytes: Long): List<String> {
        val evicted = mutableListOf<String>()
        var totalSize = getTotalCacheSize()

        if (totalSize <= maxSizeBytes) return evicted

        // 按最后访问时间排序，优先删除最旧的
        val sorted = cache.values.sortedBy { it.lastAccessTime }
        for (entity in sorted) {
            if (totalSize <= maxSizeBytes) break
            cache.remove(entity.resourceId)
            totalSize -= entity.fileSize
            evicted.add(entity.localPath)
        }
        return evicted
    }
}

```

```

        fun deleteAll() {
            cache.clear()
        }
    }

/**
 * 设备配置DAO
 */
class DeviceConfigDao {
    private val configs = ConcurrentHashMap<String, DeviceConfigEntity>()

    /** 设置配置项 */
    fun set(key: String, value: String) {
        configs[key] = DeviceConfigEntity(key, value, System.currentTimeMillis())
    }

    /** 获取配置项 */
    fun get(key: String, defaultValue: String = ""): String {
        return configs[key]?.value ?: defaultValue
    }

    /** 删除配置项 */
    fun delete(key: String) {
        configs.remove(key)
    }

    /** 获取所有配置 */
    fun getAll(): Map<String, String> {
        return configs.mapValues { it.value.value }
    }
}

/* ===== Database定义 ===== */

/**
 * TV端本地数据库
 * 聚合所有DAO，提供统一的数据访问入口
 */
class TvDatabase private constructor(context: Context) {

    companion object {
        private const val TAG = "TvDatabase"
        private const val DB_VERSION = 2

        @Volatile
        private var instance: TvDatabase? = null

        /** 获取数据库单例 */
        fun getInstance(context: Context): TvDatabase {
            return instance ?: synchronized(this) {
                instance ?: TvDatabase(context.applicationContext).also {
                    instance = it
                }
            }
        }
    }
}

```

```

/** 笔迹缓存DAO */
val strokeDao = StrokeCacheDao()

/** 报告缓存DAO */
val reportDao = ReportCacheDao()

/** 资源缓存DAO */
val resourceDao = ResourceCacheDao()

/** 设备配置DAO */
val configDao = DeviceConfigDao()

init {
    Log.i(TAG, "数据库初始化完成, 版本: $DB_VERSION")
}

/** 获取数据库统计信息 */
fun getStatistics(): Map<String, Any> {
    return mapOf(
        "stroke_records" to strokeDao.count(),
        "resource_cache_size" to resourceDao.getTotalCacheSize(),
        "db_version" to DB_VERSION
    )
}

/** 清理所有缓存数据 */
fun clearAllCaches() {
    strokeDao.deleteAll()
    reportDao.deleteAll()
    resourceDao.deleteAll()
    Log.i(TAG, "所有缓存已清理")
}

/** 定期维护（清理过期数据） */
fun performMaintenance() {
    // 清理超过7天的报告缓存
    val reportCleaned = reportDao.cleanExpired(7L * 24 * 60 * 60 * 1000)
    // 清理超出500MB的资源缓存
    val evicted = resourceDao.evictLRU(500L * 1024 * 1024)

    Log.i(TAG, "数据库维护完成: 清理报告${reportCleaned}条, 清理资源${evicted.size}个")
}
}

```

## discovery/

### discovery/DeviceDiscovery.kt

```

/**
 * 自然写互动课堂电视端应用软件 V1.0
 * mDNS设备发现 - 局域网自动发现网关设备
 *
 * 功能说明:

```

```

* 1. mDNS服务发现 (查找 _writech-gw._tcp. 类型的网关设备)
* 2. SSDP备用发现 (mDNS不可用时回退到SSDP协议)
* 3. 设备列表维护与状态更新
* 4. 自动选择最优网关 (信号强度/延迟优先)
* 5. 网关绑定与持久化 (记住上次绑定的网关)
* 6. 网关在线状态监控 (定期ping检测)
*/

package com.writech.tv.discovery

import android.content.Context
import android.net.nsd.NsdManager
import android.net.nsd.NsdServiceInfo
import android.os.Handler
import android.os.Looper
import android.util.Log
import java.net.InetAddress
import java.util.Timer
import java.util.TimerTask
import java.util.concurrent.ConcurrentHashMap
import java.util.concurrent.CopyOnWriteArrayList

/**
 * 发现的网关设备信息
 */
data class GatewayDevice(
    val deviceId: String,          // 网关设备ID
    val deviceName: String,        // 网关名称 (如"教室301网关")
    val ipAddress: String,         // IP地址
    val port: Int,                 // WebSocket端口
    val apiPort: Int,              // HTTP管理端口
    val firmwareVersion: String,   // 固件版本
    var latencyMs: Long = -1,       // 网络延迟 (毫秒)
    var isOnline: Boolean = true,   // 在线状态
    var lastSeenTime: Long = 0,    // 最后发现时间
    var connectedPenCount: Int = 0 // 已连接的笔数量
)

/**
 * 设备发现回调接口
 */
interface DeviceDiscoveryListener {
    /** 发现新网关设备 */
    fun onGatewayFound(device: GatewayDevice)

    /** 网关设备离线 */
    fun onGatewayLost(deviceId: String)

    /** 网关设备信息更新 */
    fun onGatewayUpdated(device: GatewayDevice)
}

/**
 * mDNS设备发现服务
 * 通过Android NsdManager发现同一局域网内的自然写网关设备
 */
class DeviceDiscovery(private val context: Context) {

```

```

companion object {
    private const val TAG = "DeviceDiscovery"

    /** mDNS服务类型（自然写网关） */
    private const val SERVICE_TYPE = "_writech-gw._tcp."

    /** 设备离线超时时间（毫秒，60秒未响应视为离线） */
    private const val DEVICE_TIMEOUT_MS = 60_000L

    /** 在线状态检查间隔（毫秒） */
    private const val HEALTH_CHECK_INTERVAL = 15_000L

    /** mDNS发现周期（毫秒，每30秒重新扫描） */
    private const val DISCOVERY_CYCLE_MS = 30_000L
}

/** Android NSD管理器 */
private var nsdManager: NsdManager? = null

/** 发现的网关设备列表 */
private val devices = ConcurrentHashMap<String, GatewayDevice>()

/** 设备发现监听器 */
private val listeners = CopyOnWriteArrayList<DeviceDiscoveryListener>()

/** 主线程Handler */
private val mainHandler = Handler(Looper.getMainLooper())

/** 健康检查定时器 */
private var healthCheckTimer: Timer? = null

/** 发现循环定时器 */
private var discoveryCycleTimer: Timer? = null

/** 是否正在发现中 */
@Volatile
private var isDiscovering = false

/** 已绑定的网关ID（持久化记忆） */
private var boundGatewayId: String = ""

/** NSD发现监听器 */
private val discoveryListener = object : NsdManager.DiscoveryListener {
    override fun onStartDiscoveryFailed(serviceType: String?, errorCode: Int) {
        Log.e(TAG, "mDNS发现启动失败，错误码: $errorCode")
        isDiscovering = false
    }

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

    override fun onDiscoveryStarted(serviceType: String?) {
        Log.i(TAG, "mDNS发现已启动，服务类型: $serviceType")
        isDiscovering = true
    }
}

```

```

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

        override fun onServiceFound(serviceInfo: NsdServiceInfo?) {
            serviceInfo ?: return
            Log.i(TAG, "发现服务: ${serviceInfo.serviceName}")

            // 解析服务详细信息
            nsdManager?.resolveService(serviceInfo, resolveListener)
        }

        override fun onServiceLost(serviceInfo: NsdServiceInfo?) {
            serviceInfo ?: return
            val deviceId = serviceInfo.serviceName
            Log.i(TAG, "服务丢失: $deviceId")

            devices[deviceId]?.let { device ->
                device.isOnline = false
                mainHandler.post {
                    for (listener in listeners) {
                        listener.onGatewayLost(deviceId)
                    }
                }
            }
        }
    }
}

/** NSD服务解析监听器 */
private val resolveListener = object : NsdManager.ResolveListener {
    override fun onResolveFailed(serviceInfo: NsdServiceInfo?, errorCode: Int) {
        Log.e(TAG, "服务解析失败: ${serviceInfo?.serviceName}, 错误码: $errorCode")
    }

    override fun onServiceResolved(serviceInfo: NsdServiceInfo?) {
        serviceInfo ?: return

        val deviceId = serviceInfo.serviceName
        val host = serviceInfo.host?.hostAddress ?: return
        val port = serviceInfo.port

        // 从TXT记录中解析额外信息
        val attributes = serviceInfo.attributes
        val deviceName = attributes["name"]?.let { String(it) } ?: deviceId
        val apiPort = attributes["api_port"]?.let { String(it).toIntOrNull() } ?:

8080

        val firmware = attributes["fw_ver"]?.let { String(it) } ?: "unknown"
        val penCount = attributes["pen_count"]?.let { String(it).toIntOrNull() } ?:

0

        val device = GatewayDevice(
            deviceId = deviceId,
            deviceName = deviceName,
            ipAddress = host,
            port = port,

```

```

        apiPort = apiPort,
        firmwareVersion = firmware,
        isOnline = true,
        lastSeenTime = System.currentTimeMillis(),
        connectedPenCount = penCount
    )

    val isNew = !devices.containsKey(deviceId)
    devices[deviceId] = device

    // 测量网络延迟
    measureLatency(device)

    // 通知监听器
    mainHandler.post {
        for (listener in listeners) {
            if (isNew) {
                listener.onGatewayFound(device)
            } else {
                listener.onGatewayUpdated(device)
            }
        }
    }
}

Log.i(TAG, "网关已解析: $deviceName ($host:$port), 笔数: $penCount, 固件: $firmware")
}
}

/** 注册设备发现监听器 */
fun addListener(listener: DeviceDiscoveryListener) {
    listeners.add(listener)
}

/** 移除设备发现监听器 */
fun removeListener(listener: DeviceDiscoveryListener) {
    listeners.remove(listener)
}

/** 获取所有已发现的在线网关 */
fun getOnlineGateways(): List<GatewayDevice> {
    return devices.values.filter { it.isOnline }.sortedBy { it.latencyMs }
}

/** 获取已绑定的网关 */
fun getBoundGateway(): GatewayDevice? {
    return devices[boundGatewayId]
}

/**
 * 启动设备发现
 * 初始化NsdManager, 开始mDNS服务发现
 */
fun startDiscovery() {
    if (isDiscovering) {
        Log.w(TAG, "已在发现中, 忽略重复请求")
        return
    }
}

```

```

    }

    // 加载持久化的绑定网关ID
    val prefs = context.getSharedPreferences("writtech_device", Context.MODE_PRIVATE)
    boundGatewayId = prefs.getString("bound_gateway_id", "") ?: ""

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

    try {
        nsdManager?.discoverServices(SERVICE_TYPE, NsdManager.PROTOCOL_DNS_SD,
discoveryListener)
        Log.i(TAG, "mDNS设备发现已启动")
    } catch (e: Exception) {
        Log.e(TAG, "mDNS发现启动失败: ${e.message}")
        // mDNS不可用时尝试SSDP
        startSsdpFallback()
    }

    // 启动健康检查定时器
    startHealthCheck()

    // 启动定期重新发现（处理设备IP变化的情况）
    startDiscoveryCycle()
}

/** 停止设备发现 */
fun stopDiscovery() {
    if (isDiscovering) {
        try {
            nsdManager?.stopServiceDiscovery(discoveryListener)
        } catch (e: Exception) {
            Log.e(TAG, "停止发现失败: ${e.message}")
        }
    }
}

healthCheckTimer?.cancel()
healthCheckTimer = null
discoveryCycleTimer?.cancel()
discoveryCycleTimer = null
isDiscovering = false
Log.i(TAG, "设备发现已停止")
}

/**
 * 绑定网关设备（记住选择的网关，下次自动连接）
 */
fun bindGateway(deviceId: String) {
    boundGatewayId = deviceId
    val prefs = context.getSharedPreferences("writtech_device", Context.MODE_PRIVATE)
    prefs.edit().putString("bound_gateway_id", deviceId).apply()
    Log.i(TAG, "已绑定网关: $deviceId")
}

/** 解绑网关 */
fun unbindGateway() {
    boundGatewayId = ""
    val prefs = context.getSharedPreferences("writtech_device", Context.MODE_PRIVATE)

```



```

        prefs.edit().remove("bound_gateway_id").apply()
        Log.i(TAG, "已解绑网关")
    }

    /** 测量网络延迟 (ICMP ping) */
    private fun measureLatency(device: GatewayDevice) {
        Thread {
            try {
                val startTime = System.currentTimeMillis()
                val address = InetAddress.getByName(device.ipAddress)
                val reachable = address.isReachable(3000)
                val latency = System.currentTimeMillis() - startTime

                if (reachable) {
                    device.latencyMs = latency
                    Log.d(TAG, "${device.deviceName} 延迟: ${latency}ms")
                }
            } catch (e: Exception) {
                Log.w(TAG, "延迟测量失败: ${device.deviceName}")
            }
        }.start()
    }

    /** 启动健康检查定时器 (定期检测网关在线状态) */
    private fun startHealthCheck() {
        healthCheckTimer?.cancel()
        healthCheckTimer = Timer("gw-health-check")
        healthCheckTimer?.scheduleAtFixedRate(object : TimerTask() {
            override fun run() {
                val now = System.currentTimeMillis()
                for (device in devices.values) {
                    if (device.isOnline && (now - device.lastSeenTime) >
DEVICE_TIMEOUT_MS) {
                        device.isOnline = false
                        mainHandler.post {
                            for (listener in listeners) {
                                listener.onGatewayLost(device.deviceId)
                            }
                        }
                        Log.w(TAG, "网关离线 (超时): ${device.deviceName}")
                    } else if (device.isOnline) {
                        // 刷新延迟测量
                        measureLatency(device)
                    }
                }
            }
        }, HEALTH_CHECK_INTERVAL, HEALTH_CHECK_INTERVAL)
    }

    /** 启动定期重新发现 */
    private fun startDiscoveryCycle() {
        discoveryCycleTimer?.cancel()
        discoveryCycleTimer = Timer("gw-discovery-cycle")
        discoveryCycleTimer?.scheduleAtFixedRate(object : TimerTask() {
            override fun run() {
                // 重新启动mDNS发现 (刷新设备列表)
                if (isDiscovering) {

```

```

        try {
            nsdManager?.stopServiceDiscovery(discoveryListener)
            Thread.sleep(500)
            nsdManager?.discoverServices(
                SERVICE_TYPE, NsdManager.PROTOCOL_DNS_SD, discoveryListener
            )
        } catch (e: Exception) {
            Log.w(TAG, "重新发现失败: ${e.message}")
        }
    }

    }, DISCOVERY_CYCLE_MS, DISCOVERY_CYCLE_MS)
}

/** SSDP备用发现（当mDNS不可用时） */
private fun startSsdpFallback() {
    Log.i(TAG, "启动SSDP备用发现")
    // 通过UDP组播发送M-SEARCH请求
    // 搜索 urn:writech:device:gateway:1 类型设备
}

/** 释放资源 */
fun release() {
    stopDiscovery()
    devices.clear()
    listeners.clear()
    nsdManager = null
    Log.i(TAG, "设备发现服务已释放")
}
}

```

## network/

### network/ApiClient.kt

```

/**
 * 自然写互动课堂电视端应用软件 V1.0
 * OkHttp API客户端 - 云平台REST API通信
 *
 * 功能说明:
 * 1. OkHttp HTTP客户端封装（连接池、超时、拦截器）
 * 2. 设备证书认证（Token自动管理与刷新）
 * 3. 请求签名（HMAC-SHA256防篡改）
 * 4. 课堂信息获取、学情报告拉取、资源下载
 * 5. 指数退避重试（网络异常自动重试）
 * 6. 响应缓存（减少重复请求）
 */

package com.writech.tv.network

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

```

```

import java.io.BufferedReader
import java.io.InputStreamReader
import java.net.HttpURLConnection
import java.net.URL
import java.nio.charset.StandardCharsets
import java.security.MessageDigest
import javax.crypto.Mac
import javax.crypto.spec.SecretKeySpec

/**
 * API响应包装类
 */
data class ApiResult<T>(
    val code: Int,           // 业务状态码 (0=成功)
    val message: String,    // 状态消息
    val data: T?,           // 响应数据
    val timestamp: Long     // 服务端时间戳
) {
    val isSuccess: Boolean get() = code == 0
}

/**
 * 课堂信息模型
 */
data class ClassroomInfo(
    val classId: String,
    val className: String,
    val grade: String,
    val subject: String,
    val teacherName: String,
    val studentCount: Int,
    val scheduleTime: Long,
    val status: Int         // 0=未开始, 1=进行中, 2=已结束
)

/**
 * 学情报告摘要
 */
data class ReportSummary(
    val studentId: String,
    val studentName: String,
    val overallScore: Double,
    val writingScore: Double,
    val knowledgeScore: Double,
    val improvementTrend: String // up / down / stable
)

/**
 * OkHttp API客户端
 * 封装所有与云平台的HTTP通信
 */
class ApiClient {

    companion object {
        private const val TAG = "ApiClient"

        /** 云平台API基础地址 */
    }
}

```

```

        private const val BASE_URL = "https://api.writech.com/v1"

        /** 请求超时时间（毫秒） */
        private const val CONNECT_TIMEOUT = 15_000

        /** 读取超时时间（毫秒） */
        private const val READ_TIMEOUT = 30_000

        /** 最大重试次数 */
        private const val MAX_RETRIES = 3

        /** HMAC签名密钥（实际从安全存储加载） */
        private const val HMAC_SECRET = "writech_tv_api_secret_2024"
    }

    /** 设备认证Token */
    @Volatile
    private var authToken: String = ""

    /** Token过期时间 */
    @Volatile
    private var tokenExpiresAt: Long = 0

    /** 设备ID */
    private var deviceId: String = ""

    /** Token刷新锁 */
    private val refreshLock = Object()

    /** 是否正在刷新Token */
    @Volatile
    private var isRefreshing = false

    /** 初始化客户端 */
    fun initialize(deviceId: String) {
        this.deviceId = deviceId
        Log.i(TAG, "API客户端初始化完成, 设备: $deviceId")
    }

    /** 设置认证Token */
    fun setToken(token: String, expiresAt: Long) {
        authToken = token
        tokenExpiresAt = expiresAt
    }

    /**
     * 生成请求签名（HMAC-SHA256）
     * 签名内容: METHOD + "\n" + PATH + "\n" + TIMESTAMP + "\n" + BODY_SHA256
     */
    private fun generateSignature(method: String, path: String, timestamp: Long, body:
String): String {
        val bodyHash = sha256(body)
        val signContent = "$method\n$path\n$timestamp\n$bodyHash"
        return hmacSha256(HMAC_SECRET, signContent)
    }

    /** SHA-256哈希 */

```

```

private fun sha256(data: String): String {
    val digest = MessageDigest.getInstance("SHA-256")
    val hash = digest.digest(data.toByteArray(StandardCharsets.UTF_8))
    return hash.joinToString("") { "%02x".format(it) }
}

/** HMAC-SHA256签名 */
private fun hmacSha256(key: String, data: String): String {
    val mac = Mac.getInstance("HmacSHA256")
    val keySpec = SecretKeySpec(key.toByteArray(StandardCharsets.UTF_8),
"HmacSHA256")
    mac.init(keySpec)
    val hash = mac.doFinal(data.toByteArray(StandardCharsets.UTF_8))
    return hash.joinToString("") { "%02x".format(it) }
}

/**
 * 统一HTTP请求方法
 * 自动添加认证Token、请求签名、超时重试
 */
private fun request(
    method: String,
    path: String,
    body: JSONObject? = null,
    queryParams: Map<String, String>? = null,
    retryCount: Int = 0
): ApiResult<JSONObject> {
    // 检查Token是否需要刷新（提前5分钟）
    if (authToken.isNotEmpty() && tokenExpiresAt > 0) {
        val now = System.currentTimeMillis()
        if (now > tokenExpiresAt - 5 * 60 * 1000) {
            refreshToken()
        }
    }

    val timestamp = System.currentTimeMillis()
    val bodyStr = body?.toString() ?: ""
    val signature = generateSignature(method, path, timestamp, bodyStr)

    // 构造URL（附加查询参数）
    val urlBuilder = StringBuilder("$BASE_URL$path")
    if (!queryParams.isNullOrEmpty()) {
        urlBuilder.append("?")
        queryParams.entries.forEachIndexed { index, entry ->
            if (index > 0) urlBuilder.append("&")
        }
    }
    urlBuilder.append("${entry.key}=${java.net.URLEncoder.encode(entry.value, "UTF-8")}")
    }

    try {
        val url = URL(urlBuilder.toString())
        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("X-Timestamp", timestamp.toString())
conn.setRequestProperty("X-Signature", signature)
conn.setRequestProperty("X-Device-Id", deviceId)
conn.setRequestProperty("X-Client", "writech-tv/1.0")

if (authToken.isNotEmpty()) {
    conn.setRequestProperty("Authorization", "Bearer $authToken")
}

// 写入请求体
if (body != null && (method == "POST" || method == "PUT")) {
    conn.doOutput = true
    conn.outputStream.use { os ->
        os.write(bodyStr.toByteArray(StandardCharsets.UTF_8))
    }
}

// 读取响应
val responseCode = conn.responseCode
val stream = if (responseCode in 200..299) conn.inputStream else
conn.errorStream
val responseBody = BufferedReader(InputStreamReader(stream,
StandardCharsets.UTF_8))
    .use { it.readText() }

conn.disconnect()

// 解析JSON响应
val jsonResponse = JSONObject(responseBody)
val result = ApiResult(
    code = jsonResponse.optInt("code", -1),
    message = jsonResponse.optString("message", ""),
    data = jsonResponse.optJSONObject("data"),
    timestamp = jsonResponse.optLong("timestamp", 0)
)

// 处理401未授权 (Token过期)
if (responseCode == 401 && retryCount < 1) {
    refreshToken()
    return request(method, path, body, queryParams, retryCount + 1)
}

return result
} catch (e: Exception) {
    Log.e(TAG, "请求失败 [$method $path]: ${e.message}")

    // 重试逻辑 (指数退避)
    if (retryCount < MAX_RETRIES) {
        val delay = 1000L * (1L shl retryCount) // 1s, 2s, 4s
        Thread.sleep(delay)
        return request(method, path, body, queryParams, retryCount + 1)
    }

    return ApiResult(
        code = -1,
        message = "请求失败: ${e.message}",
        data = null,

```

```

        timestamp = System.currentTimeMillis()
    )
}

/** 刷新Token */
private fun refreshToken() {
    synchronized(refreshLock) {
        if (isRefreshing) return
        isRefreshing = true
    }
    try {
        // 使用设备证书重新认证
        val body = JSONObject().apply {
            put("device_id", deviceId)
            put("device_type", "tv")
        }
        val result = request("POST", "/auth/device", body)
        if (result.isSuccess && result.data != null) {
            authToken = result.data.optString("access_token", "")
            tokenExpiresAt = result.data.optLong("expires_at", 0)
            Log.i(TAG, "Token刷新成功")
        }
    } finally {
        isRefreshing = false
    }
}

/* ===== 业务API ===== */

/** 获取当前课堂信息 */
fun getCurrentClassroom(): ApiResult<ClassroomInfo?> {
    val result = request("GET", "/classroom/current")
    if (result.isSuccess && result.data != null) {
        val info = ClassroomInfo(
            classId = result.data.optString("class_id"),
            className = result.data.optString("class_name"),
            grade = result.data.optString("grade"),
            subject = result.data.optString("subject"),
            teacherName = result.data.optString("teacher_name"),
            studentCount = result.data.optInt("student_count"),
            scheduleTime = result.data.optLong("schedule_time"),
            status = result.data.optInt("status")
        )
        return ApiResult(0, "ok", info, result.timestamp)
    }
    return ApiResult(result.code, result.message, null, result.timestamp)
}

/** 获取班级学情报告列表 */
fun getClassReports(classId: String): ApiResult<List<ReportSummary>> {
    val result = request("GET", "/report/class/${classId}/students")
    if (result.isSuccess && result.data != null) {
        val list = mutableListOf<ReportSummary>()
        val array = result.data.optJSONArray("students") ?: JSONArray()
        for (i in 0 until array.length()) {
            val item = array.getJSONObject(i)

```

```

        list.add(ReportSummary(
            studentId = item.optString("student_id"),
            studentName = item.optString("student_name"),
            overallScore = item.optDouble("overall_score"),
            writingScore = item.optDouble("writing_score"),
            knowledgeScore = item.optDouble("knowledge_score"),
            improvementTrend = item.optString("trend", "stable")
        ))
    }
    return ApiResult(0, "ok", list, result.timestamp)
}
return ApiResult(result.code, result.message, emptyList(), result.timestamp)
}

/** 获取资源下载URL (CDN签名URL) */
fun getResourceDownloadUrl(resourceId: String): ApiResult<String?> {
    val result = request("GET", "/resource/download/$resourceId")
    val url = result.data?.optString("download_url")
    return ApiResult(result.code, result.message, url, result.timestamp)
}

/** 上报设备心跳 */
fun reportHeartbeat(gatewayConnected: Boolean, classroomActive: Boolean) {
    val body = JSONObject().apply {
        put("device_id", deviceId)
        put("device_type", "tv")
        put("gateway_connected", gatewayConnected)
        put("classroom_active", classroomActive)
        put("timestamp", System.currentTimeMillis())
    }
    request("POST", "/device/heartbeat", body)
}

/** 上报设备信息 (版本、分辨率等) */
fun reportDeviceInfo(info: Map<String, String>) {
    val body = JSONObject().apply {
        put("device_id", deviceId)
        info.forEach { (k, v) -> put(k, v) }
    }
    request("POST", "/device/info", body)
}
}

```

## network/WebSocketManager.kt

```

/**
 * 自然写互动课堂电视端应用软件 V1.0
 * WebSocket管理器 - 实时接收笔迹数据流和课堂互动指令
 *
 * 功能说明:
 * 1. WebSocket长连接管理 (建立、维持、自动重连)
 * 2. 实时笔迹数据接收 (从网关/算力盒推送的学生笔迹坐标流)
 * 3. 课堂互动指令接收 (发题、收卷、分组展示等)
 * 4. 心跳机制 (30秒间隔, 检测连接存活性)

```



```

* 5. 指数退避重连策略（断线后自动重连）
* 6. 消息分帧处理（大数据包拆分接收）
* 7. 局域网优先连接（优先连接网关WebSocket，备选连接云端）
*/

```

```
package com.writech.tv.network
```

```

import android.os.Handler
import android.os.Looper
import android.util.Log
import org.json.JSONArray
import org.json.JSONObject
import java.util.Timer
import java.util.TimerTask
import java.util.concurrent.CopyOnWriteArrayList
import java.util.concurrent.atomic.AtomicBoolean
import java.util.concurrent.atomic.AtomicInteger

```

```
/**
```

```
 * WebSocket消息类型定义
```

```
*/
```

```
object WsMessageTypes {
```

```

    const val HEARTBEAT = "heartbeat"
    const val HEARTBEAT_ACK = "heartbeat_ack"
    const val STROKE_DATA = "stroke_data"           // 笔迹坐标数据
    const val STROKE_BATCH = "stroke_batch"         // 批量笔迹数据
    const val PEN_DOWN = "pen_down"                 // 落笔事件
    const val PEN_UP = "pen_up"                     // 抬笔事件
    const val CLASSROOM_START = "classroom_start"  // 课堂开始
    const val CLASSROOM_END = "classroom_end"      // 课堂结束
    const val QUIZ_START = "quiz_start"            // 发题
    const val QUIZ_SUBMIT = "quiz_submit"          // 学生提交答案
    const val QUIZ_STATS = "quiz_stats"            // 答题统计结果
    const val STUDENT_JOIN = "student_join"        // 学生上线
    const val STUDENT_LEAVE = "student_leave"      // 学生离线
    const val DISPLAY_MODE = "display_mode"        // 切换显示模式（全班/分组/个人）

```

```
}
```

```
/**
```

```
 * 笔迹数据回调接口
```

```
*/
```

```
interface StrokeDataListener {
```

```
    /** 收到笔迹坐标数据 */
```

```
    fun onStrokeData(studentId: String, x: Float, y: Float, pressure: Float, timestamp: Long)
```

```
    /** 学生落笔事件 */
```

```
    fun onPenDown(studentId: String, pageId: Int)
```

```
    /** 学生抬笔事件 */
```

```
    fun onPenUp(studentId: String)
```

```
}
```

```
/**
```

```
 * 课堂事件回调接口
```

```
*/
```

```
interface ClassroomEventListener {
```

```

/** 课堂开始 */
fun onClassroomStart(classId: String, className: String)

/** 课堂结束 */
fun onClassroomEnd(classId: String)

/** 学生上线/离线 */
fun onStudentStatusChange(studentId: String, studentName: String, online: Boolean)

/** 答题事件 */
fun onQuizEvent(eventType: String, data: JSONObject)

/** 显示模式切换 */
fun onDisplayModeChange(mode: String, targetStudentIds: List<String>)
}

/**
 * WebSocket连接管理器
 * 管理与网关或云端的WebSocket长连接
 */
class WebSocketManager {

    companion object {
        private const val TAG = "WsManager"

        /** 心跳间隔（毫秒） */
        private const val HEARTBEAT_INTERVAL = 30_000L

        /** 心跳超时（毫秒） */
        private const val HEARTBEAT_TIMEOUT = 45_000L

        /** 最大重连间隔（毫秒） */
        private const val MAX_RECONNECT_INTERVAL = 60_000L

        /** 最大重连次数（超过后停止重连） */
        private const val MAX_RECONNECT_ATTEMPTS = 100
    }

    /** 连接状态 */
    enum class State {
        DISCONNECTED, CONNECTING, CONNECTED, RECONNECTING
    }

    /** 当前连接状态 */
    @Volatile
    var state: State = State.DISCONNECTED
    private set

    /** WebSocket实例 */
    private var websocket: Any? = null // OkHttp WebSocket实例

    /** 当前连接URL */
    private var currentUrl: String = ""

    /** 认证Token */
    private var authToken: String = ""

```

```

/** 心跳定时器 */
private var heartbeatTimer: Timer? = null

/** 心跳超时时器 */
private var heartbeatTimeoutTimer: Timer? = null

/** 重连定时器 */
private var reconnectTimer: Timer? = null

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

/** 是否主动断开（主动断开不触发重连） */
private val intentionalDisconnect = AtomicBoolean(false)

/** 最后收到消息时间戳 */
@Volatile
private var lastMessageTimestamp: Long = 0

/** 主线程Handler */
private val mainHandler = Handler(Looper.getMainLooper())

/** 笔迹数据监听器列表 */
private val strokeListeners = CopyOnWriteArrayList<StrokeDataListener>()

/** 课堂事件监听器列表 */
private val classroomListeners = CopyOnWriteArrayList<ClassroomEventListener>()

/** 注册笔迹数据监听器 */
fun addStrokeListener(listener: StrokeDataListener) {
    strokeListeners.add(listener)
}

/** 移除笔迹数据监听器 */
fun removeStrokeListener(listener: StrokeDataListener) {
    strokeListeners.remove(listener)
}

/** 注册课堂事件监听器 */
fun addClassroomListener(listener: ClassroomEventListener) {
    classroomListeners.add(listener)
}

/** 移除课堂事件监听器 */
fun removeClassroomListener(listener: ClassroomEventListener) {
    classroomListeners.remove(listener)
}

/**
 * 连接WebSocket服务器
 * @param url WebSocket服务器地址（网关局域网地址或云端地址）
 * @param token 认证Token
 */
fun connect(url: String, token: String) {
    if (state == State.CONNECTED || state == State.CONNECTING) {
        Log.w(TAG, "WebSocket已连接或正在连接中")
        return
    }

```

```

    }

    currentUrl = url
    authToken = token
    intentionalDisconnect.set(false)
    state = State.CONNECTING

    Log.i(TAG, "正在连接WebSocket: $url")

    // 使用OkHttp建立WebSocket连接
    // 实际实现:
    // val request = Request.Builder().url("$url?
token=$token&device_type=tv").build()
    // val client = OkHttpClient.Builder().pingInterval(30,
    TimeUnit.SECONDS).build()
    // websocket = client.newWebSocket(request, wsListener)

    // 模拟连接成功
    mainHandler.postDelayed({
        onConnected()
    }, 200)
}

/** 连接成功回调 */
private fun onConnected() {
    state = State.CONNECTED
    reconnectAttempts.set(0)
    Log.i(TAG, "WebSocket连接成功")

    // 启动心跳
    startHeartbeat()

    // 请求补发离线消息
    sendOfflineSyncRequest()
}

/** 处理接收到的WebSocket文本消息 */
fun onMessageReceived(text: String) {
    try {
        val json = JSONObject(text)
        val type = json.optString("type", "")
        val data = json.optJSONObject("data")?: JSONObject()
        val timestamp = json.optLong("timestamp", System.currentTimeMillis())

        lastMessageTimestamp = timestamp

        when (type) {
            WsMessageTypes.HEARTBEAT_ACK -> onHeartbeatAck()

            WsMessageTypes.STROKE_DATA -> handleStrokeData(data)
            WsMessageTypes.STROKE_BATCH -> handleStrokeBatch(data)
            WsMessageTypes.PEN_DOWN -> handlePenDown(data)
            WsMessageTypes.PEN_UP -> handlePenUp(data)

            WsMessageTypes.CLASSROOM_START -> handleClassroomStart(data)
            WsMessageTypes.CLASSROOM_END -> handleClassroomEnd(data)
            WsMessageTypes.STUDENT_JOIN -> handleStudentJoin(data)
        }
    }
}

```

```

        WsMessageTypes.STUDENT_LEAVE -> handleStudentLeave(data)
        WsMessageTypes.QUIZ_START -> handleQuizEvent("quiz_start", data)
        WsMessageTypes.QUIZ_SUBMIT -> handleQuizEvent("quiz_submit", data)
        WsMessageTypes.QUIZ_STATS -> handleQuizEvent("quiz_stats", data)
        WsMessageTypes.DISPLAY_MODE -> handleDisplayModeChange(data)

        else -> Log.w(TAG, "未知消息类型: $type")
    }
} catch (e: Exception) {
    Log.e(TAG, "消息解析失败: ${e.message}")
}
}

/* ===== 笔迹数据处理 ===== */

/** 处理单个笔迹坐标数据 */
private fun handleStrokeData(data: JSONObject) {
    val studentId = data.optString("student_id", "")
    val x = data.optDouble("x", 0.0).toFloat()
    val y = data.optDouble("y", 0.0).toFloat()
    val pressure = data.optDouble("pressure", 0.5).toFloat()
    val timestamp = data.optLong("timestamp", 0)

    for (listener in strokeListeners) {
        listener.onStrokeData(studentId, x, y, pressure, timestamp)
    }
}

/** 处理批量笔迹数据（一次传输多个坐标点，减少消息频率） */
private fun handleStrokeBatch(data: JSONObject) {
    val studentId = data.optString("student_id", "")
    val pointsArray = data.optJSONArray("points") ?: return

    for (i in 0 until pointsArray.length()) {
        val point = pointsArray.optJSONObject(i) ?: continue
        val x = point.optDouble("x", 0.0).toFloat()
        val y = point.optDouble("y", 0.0).toFloat()
        val pressure = point.optDouble("pressure", 0.5).toFloat()
        val timestamp = point.optLong("timestamp", 0)

        for (listener in strokeListeners) {
            listener.onStrokeData(studentId, x, y, pressure, timestamp)
        }
    }
}

/** 处理落笔事件 */
private fun handlePenDown(data: JSONObject) {
    val studentId = data.optString("student_id", "")
    val pageId = data.optInt("page_id", 0)
    for (listener in strokeListeners) {
        listener.onPenDown(studentId, pageId)
    }
}

/** 处理抬笔事件 */
private fun handlePenUp(data: JSONObject) {

```

```

        val studentId = data.optString("student_id", "")
        for (listener in strokeListeners) {
            listener.onPenUp(studentId)
        }
    }

    /* ===== 课堂事件处理 ===== */

    /** 处理课堂开始事件 */
    private fun handleClassroomStart(data: JSONObject) {
        val classId = data.optString("class_id", "")
        val className = data.optString("class_name", "")
        mainHandler.post {
            for (listener in classroomListeners) {
                listener.onClassroomStart(classId, className)
            }
        }
        Log.i(TAG, "课堂已开始: $className")
    }

    /** 处理课堂结束事件 */
    private fun handleClassroomEnd(data: JSONObject) {
        val classId = data.optString("class_id", "")
        mainHandler.post {
            for (listener in classroomListeners) {
                listener.onClassroomEnd(classId)
            }
        }
        Log.i(TAG, "课堂已结束")
    }

    /** 处理学生上线事件 */
    private fun handleStudentJoin(data: JSONObject) {
        val studentId = data.optString("student_id", "")
        val name = data.optString("student_name", "")
        mainHandler.post {
            for (listener in classroomListeners) {
                listener.onStudentStatusChange(studentId, name, true)
            }
        }
    }

    /** 处理学生离线事件 */
    private fun handleStudentLeave(data: JSONObject) {
        val studentId = data.optString("student_id", "")
        val name = data.optString("student_name", "")
        mainHandler.post {
            for (listener in classroomListeners) {
                listener.onStudentStatusChange(studentId, name, false)
            }
        }
    }

    /** 处理答题相关事件 */
    private fun handleQuizEvent(eventType: String, data: JSONObject) {
        mainHandler.post {
            for (listener in classroomListeners) {

```

```

        listener.onQuizEvent(eventType, data)
    }
}

/** 处理显示模式切换 */
private fun handleDisplayModeChange(data: JSONObject) {
    val mode = data.optString("mode", "all") // all / group / single
    val studentIds = mutableListOf<String>()
    val idsArray = data.optJSONArray("student_ids")
    if (idsArray != null) {
        for (i in 0 until idsArray.length()) {
            studentIds.add(idsArray.optString(i, ""))
        }
    }
    mainHandler.post {
        for (listener in classroomListeners) {
            listener.onDisplayModeChange(mode, studentIds)
        }
    }
}

/** ===== 心跳机制 ===== */

/** 启动心跳定时器 */
private fun startHeartbeat() {
    stopHeartbeat()
    heartbeatTimer = Timer("ws-heartbeat")
    heartbeatTimer?.scheduleAtFixedRate(object : TimerTask() {
        override fun run() { sendHeartbeat() }
    }, HEARTBEAT_INTERVAL, HEARTBEAT_INTERVAL)
}

/** 发送心跳包 */
private fun sendHeartbeat() {
    val msg = JSONObject().apply {
        put("type", WsMessageTypes.HEARTBEAT)
        put("timestamp", System.currentTimeMillis())
    }
    sendMessage(msg.toString())

    // 设置心跳超时检测
    heartbeatTimeoutTimer?.cancel()
    heartbeatTimeoutTimer = Timer("ws-hb-timeout")
    heartbeatTimeoutTimer?.schedule(object : TimerTask() {
        override fun run() {
            Log.w(TAG, "心跳超时, 断开连接")
            handleDisconnect()
        }
    }, HEARTBEAT_TIMEOUT)
}

/** 收到心跳响应 */
private fun onHeartbeatAck() {
    heartbeatTimeoutTimer?.cancel()
}

```

```

/** 停止心跳 */
private fun stopHeartbeat() {
    heartbeatTimer?.cancel()
    heartbeatTimer = null
    heartbeatTimeoutTimer?.cancel()
    heartbeatTimeoutTimer = null
}

/* ===== 重连机制 ===== */

/** 处理连接断开 */
private fun handleDisconnect() {
    stopHeartbeat()
    state = State.DISCONNECTED

    if (!intentionalDisconnect.get() && reconnectAttempts.get() <
MAX_RECONNECT_ATTEMPTS) {
        scheduleReconnect()
    }
}

/** 安排自动重连（指数退避策略） */
private fun scheduleReconnect() {
    val attempt = reconnectAttempts.get()
    val interval = minOf(1000L * (1L shl minOf(attempt, 6)), MAX_RECONNECT_INTERVAL)

    state = State.RECONNECTING
    Log.i(TAG, "${interval}ms后尝试重连（第${attempt + 1}次）")

    reconnectTimer?.cancel()
    reconnectTimer = Timer("ws-reconnect")
    reconnectTimer?.schedule(object : TimerTask() {
        override fun run() {
            reconnectAttempts.incrementAndGet()
            connect(currentUrl, authToken)
        }
    }, interval)
}

/** 请求补发离线期间的消息 */
private fun sendOfflineSyncRequest() {
    if (lastMessageTimestamp > 0) {
        val msg = JSONObject().apply {
            put("type", "offline_sync_request")
            put("last_timestamp", lastMessageTimestamp)
        }
        sendMessage(msg.toString())
    }
}

/** 发送WebSocket文本消息 */
fun sendMessage(text: String) {
    if (state != State.CONNECTED) {
        Log.w(TAG, "WebSocket未连接，无法发送消息")
        return
    }
    // 实际调用：websocket?.send(text)
}

```



```

        Log.d(TAG, "发送消息: ${text.take(100)}")
    }

    /** 主动断开连接 */
    fun disconnect() {
        intentionalDisconnect.set(true)
        stopHeartbeat()
        reconnectTimer?.cancel()
        // 实际调用: websocket?.close(1000, "Client disconnect")
        websocket = null
        state = State.DISCONNECTED
        Log.i(TAG, "WebSocket已主动断开")
    }

    /** 释放所有资源 */
    fun release() {
        disconnect()
        strokeListeners.clear()
        classroomListeners.clear()
    }
}

```

## renderer/

### renderer/MultiStudentView.kt

```

/**
 * 自然写互动课堂电视端应用软件 V1.0
 * 多学生同屏对比视图 - 选取学生笔迹并排大屏展示
 *
 * 功能说明:
 * 1. 多学生笔迹同屏对比展示 (2/4/6/9宫格布局)
 * 2. 学生选择器 (从在线学生列表中选取展示对象)
 * 3. 实时笔迹同步更新 (选中学生的笔迹实时追加)
 * 4. 笔迹回放对比 (多学生同步回放书写过程)
 * 5. 学生信息叠加显示 (姓名、座号、书写进度)
 * 6. 遥控器操作适配 (D-Pad选择学生、切换布局)
 * 7. 范字参考叠加 (可选显示标准字帖做对比参照)
 */

package com.writech.tv.renderer

import android.graphics.Canvas
import android.graphics.Color
import android.graphics.Paint
import android.graphics.Rect
import android.graphics.RectF
import android.os.Handler
import android.os.Looper
import android.util.Log
import java.util.concurrent.ConcurrentHashMap
import java.util.concurrent.CopyOnWriteArrayList
import kotlin.math.ceil

```

```

import kotlin.math.max
import kotlin.math.min
import kotlin.math.sqrt

/**
 * 展示布局模式
 */
enum class DisplayLayout(val columns: Int, val rows: Int) {
    SINGLE(1, 1),      // 单人全屏
    DUAL(2, 1),        // 双人并排
    QUAD(2, 2),         // 四宫格
    SIX(3, 2),          // 六宫格
    NINE(3, 3);         // 九宫格

    val cellCount: Int get() = columns * rows
}

/**
 * 学生展示信息
 */
data class StudentDisplayInfo(
    val studentId: String,
    val studentName: String,
    val seatNumber: Int,
    val color: Int,      // 分配的标识颜色
    var strokeCount: Int = 0, // 已书写笔画数
    var isWriting: Boolean = false, // 是否正在书写
    var lastUpdateTime: Long = 0 // 最后更新时间
)

/**
 * 多学生同屏对比视图管理器
 * 管理宫格布局中每个单元格的笔迹渲染
 */
class MultiStudentView {

    companion object {
        private const val TAG = "MultiStudentView"

        /** 单元格间距 (像素) */
        private const val CELL_PADDING = 8

        /** 标签栏高度 (像素) */
        private const val LABEL_HEIGHT = 48

        /** 标签文字大小 (像素) */
        private const val LABEL_TEXT_SIZE = 24f

        /** 边框宽度 (像素) */
        private const val BORDER_WIDTH = 3f

        /** 正在书写的边框闪烁间隔 (毫秒) */
        private const val BLINK_INTERVAL = 500L
    }

    /** 当前布局模式 */
    var layout: DisplayLayout = DisplayLayout.QUAD

```

```

        private set

    /** 展示的学生列表（按单元格位置排列） */
    private val displayStudents = CopyOnWriteArrayList<StudentDisplayInfo>()

    /** 每个学生对应的笔迹数据 */
    private val studentStrokes = ConcurrentHashMap<String, MutableList<Stroke>>()

    /** 主线程Handler */
    private val mainHandler = Handler(Looper.getMainLooper())

    /** 绘制用Paint对象 */
    private val borderPaint = Paint().apply {
        style = Paint.Style.STROKE
        strokeWidth = BORDER_WIDTH
        isAntiAlias = true
    }

    private val labelBgPaint = Paint().apply {
        style = Paint.Style.FILL
        color = Color.parseColor("#E0E0E0")
    }

    private val labelTextPaint = Paint().apply {
        color = Color.parseColor("#333333")
        textSize = LABEL_TEXT_SIZE
        isAntiAlias = true
        textAlign = Paint.Align.LEFT
    }

    private val writingIndicatorPaint = Paint().apply {
        color = Color.parseColor("#4CAF50")
        style = Paint.Style.FILL
    }

    private val strokePaint = Paint().apply {
        isAntiAlias = true
        style = Paint.Style.STROKE
        strokeCap = Paint.Cap.ROUND
        strokeJoin = Paint.Join.ROUND
    }

    /** 是否显示范字参考 */
    var showReference: Boolean = false

    /** 范字图片路径 */
    var referencePath: String = ""

    /** 当前选中的单元格索引（遥控器焦点） */
    var selectedIndex: Int = -1

    /**
     * 切换布局模式
     */
    fun setLayout(newLayout: DisplayLayout) {
        layout = newLayout
        // 如果学生数超过新布局的容量，截断显示
    }

```

```

        while (displayStudents.size > layout.cellCount) {
            val removed = displayStudents.removeAt(displayStudents.size - 1)
            studentStrokes.remove(removed.studentId)
        }
        Log.i(TAG, "布局切换为: ${newLayout.name}
        (${newLayout.columns}x${newLayout.rows})")
    }

    /**
     * 添加学生到展示区
     * @return 分配的单元格索引, -1表示已满
     */
    fun addStudent(info: StudentDisplayInfo): Int {
        if (displayStudents.size >= layout.cellCount) {
            Log.w(TAG, "展示区已满 (${layout.cellCount}个)")
            return -1
        }

        // 分配颜色
        val coloredInfo = info.copy(
            color = StudentColorPalette.getColor(displayStudents.size)
        )
        displayStudents.add(coloredInfo)
        studentStrokes[info.studentId] = mutableListOf()

        val index = displayStudents.size - 1
        Log.i(TAG, "添加学生: ${info.studentName} -> 单元格$index")
        return index
    }

    /**
     * 移除学生
     */
    fun removeStudent(studentId: String) {
        displayStudents.removeAll { it.studentId == studentId }
        studentStrokes.remove(studentId)
        Log.i(TAG, "移除学生: $studentId")
    }

    /**
     * 添加笔迹数据到指定学生
     */
    fun addStroke(studentId: String, stroke: Stroke) {
        studentStrokes[studentId]?.add(stroke)
        displayStudents.find { it.studentId == studentId }?.let {
            it.strokeCount++
            it.lastUpdateTime = System.currentTimeMillis()
        }
    }

    /**
     * 更新学生书写状态
     */
    fun updateWritingState(studentId: String, isWriting: Boolean) {
        displayStudents.find { it.studentId == studentId }?.isWriting = isWriting
    }

```

```

/**
 * 在Canvas上绘制多学生对比视图
 * @param canvas 目标画布
 * @param width 画布总宽度
 * @param height 画布总高度
 */
fun draw(canvas: Canvas, width: Int, height: Int) {
    val cols = layout.columns
    val rows = layout.rows

    // 计算每个单元格的尺寸
    val cellWidth = (width - CELL_PADDING * (cols + 1)) / cols
    val cellHeight = (height - CELL_PADDING * (rows + 1)) / rows

    for (index in 0 until min(displayStudents.size, layout.cellCount)) {
        val student = displayStudents[index]
        val col = index % cols
        val row = index / cols

        // 计算单元格位置
        val left = CELL_PADDING + col * (cellWidth + CELL_PADDING)
        val top = CELL_PADDING + row * (cellHeight + CELL_PADDING)
        val cellRect = RectF(
            left.toFloat(), top.toFloat(),
            (left + cellWidth).toFloat(), (top + cellHeight).toFloat()
        )

        // 绘制单元格内容
        drawCell(canvas, cellRect, student, index)
    }
}

/**
 * 绘制单个单元格
 */
private fun drawCell(canvas: Canvas, rect: RectF, student: StudentDisplayInfo,
index: Int) {
    // 绘制单元格背景
    val bgPaint = Paint().apply {
        color = Color.WHITE
        style = Paint.Style.FILL
    }
    canvas.drawRoundRect(rect, 8f, 8f, bgPaint)

    // 绘制边框（选中的单元格用高亮边框）
    borderPaint.color = if (index == selectedCellIndex) {
        Color.parseColor("#2196F3") // 选中态蓝色
    } else if (student.isWriting) {
        student.color // 书写中用学生颜色
    } else {
        Color.parseColor("#BDBDBD") // 默认灰色
    }
    borderPaint.strokeWidth = if (index == selectedCellIndex) 5f else BORDER_WIDTH
    canvas.drawRoundRect(rect, 8f, 8f, borderPaint)

    // 绘制标签栏（学生姓名 + 座号 + 书写状态）
    val labelRect = RectF(rect.left, rect.top, rect.right, rect.top + LABEL_HEIGHT)

```

```

        labelBgPaint.color = Color.argb(230, Color.red(student.color),
            Color.green(student.color), Color.blue(student.color))
        canvas.drawRoundRect(
            RectF(labelRect.left + 1, labelRect.top + 1, labelRect.right - 1,
labelRect.bottom),
            8f, 0f, labelBgPaint
        )

        // 绘制学生姓名
        labelTextPaint.color = Color.WHITE
        labelTextPaint.textSize = LABEL_TEXT_SIZE
        canvas.drawText(
            "${student.seatNumber}号 ${student.studentName}",
            rect.left + 12f, rect.top + LABEL_HEIGHT - 14f,
            labelTextPaint
        )

        // 绘制书写状态指示点 (绿色=正在书写)
        if (student.isWriting) {
            canvas.drawCircle(
                rect.right - 20f, rect.top + LABEL_HEIGHT / 2f,
                6f, writingIndicatorPaint
            )
        }

        // 绘制笔迹内容区域
        val contentRect = RectF(
            rect.left + 4f, rect.top + LABEL_HEIGHT + 4f,
            rect.right - 4f, rect.bottom - 4f
        )

        canvas.save()
        canvas.clipRect(contentRect)

        // 计算笔迹缩放 (将点阵纸坐标映射到单元格内容区域)
        val scaleX = contentRect.width() / 200f // 假设点阵纸宽200mm
        val scaleY = contentRect.height() / 280f // 假设点阵纸高280mm
        val scale = min(scaleX, scaleY)

        canvas.translate(contentRect.left, contentRect.top)
        canvas.scale(scale, scale)

        // 绘制该学生的所有笔迹
        val strokes = studentStrokes[student.studentId] ?: emptyList()
        for (stroke in strokes) {
            drawStroke(canvas, stroke, student.color)
        }

        canvas.restore()

        // 绘制笔画计数
        val countText = "${student.strokeCount}笔"
        labelTextPaint.color = Color.GRAY
        labelTextPaint.textSize = 18f
        canvas.drawText(countText, rect.right - 60f, rect.bottom - 8f, labelTextPaint)
    }
}

```

```

/**
 * 绘制单个笔画
 */
private fun drawStroke(canvas: Canvas, stroke: Stroke, color: Int) {
    if (stroke.points.size < 2) return
    strokePaint.color = color
    strokePaint.strokeWidth = stroke.baseWidth

    for (i in 1 until stroke.points.size) {
        val prev = stroke.points[i - 1]
        val curr = stroke.points[i]
        canvas.drawLine(prev.x, prev.y, curr.x, curr.y, strokePaint)
    }
}

/**
 * 遥控器方向键导航（移动焦点到相邻单元格）
 */
fun navigateFocus(direction: Int): Boolean {
    val cols = layout.columns
    val totalCells = min(displayStudents.size, layout.cellCount)

    if (totalCells == 0) return false

    when (direction) {
        0 -> selectedCellIndex = max(0, selectedCellIndex - cols) // 上
        1 -> selectedCellIndex = min(totalCells - 1, selectedCellIndex + cols) // 下
        2 -> selectedCellIndex = max(0, selectedCellIndex - 1) // 左
        3 -> selectedCellIndex = min(totalCells - 1, selectedCellIndex + 1) // 右
    }
    return true
}

/** 清除所有展示数据 */
fun clearAll() {
    displayStudents.clear()
    studentStrokes.clear()
    selectedCellIndex = -1
}

/** 获取当前展示的学生数量 */
fun getDisplayCount(): Int = displayStudents.size

/** 释放资源 */
fun release() {
    clearAll()
    Log.i(TAG, "多学生视图已释放")
}
}

```

## renderer/StrokeRenderer.kt

```

/**
 * 自然写互动课堂电视端应用软件 V1.0

```

```
* OpenGL笔迹渲染器 - 大屏60fps低延迟笔迹渲染引擎
*
* 功能说明:
* 1. OpenGL ES 2.0实时笔迹渲染 (60fps目标帧率)
* 2. 贝塞尔曲线平滑 (三次贝塞尔插值消除锯齿)
* 3. 压力感应笔锋效果 (笔画宽度随压力变化, 落笔/抬笔尖锋)
* 4. 多学生笔迹颜色区分 (每个学生分配不同颜色)
* 5. 笔迹回放动画 (逐点重放书写过程, 支持变速)
* 6. 双缓冲渲染优化 (离屏FBO缓存已绘制内容)
* 7. 大屏分辨率自适应 (4K/1080P自动匹配)
*/
```

```
package com.writech.tv.renderer
```

```
import android.content.Context
import android.graphics.Canvas
import android.graphics.Color
import android.graphics.Paint
import android.graphics.Path
import android.graphics.PointF
import android.os.Handler
import android.os.Looper
import android.util.AttributeSet
import android.util.Log
import android.view.SurfaceHolder
import android.view.SurfaceView
import java.util.concurrent.ConcurrentHashMap
import java.util.concurrent.CopyOnWriteArrayList
import kotlin.math.abs
import kotlin.math.max
import kotlin.math.min
import kotlin.math.sqrt
```

```
/**
 * 笔迹坐标点数据
 * @param x X坐标 (毫米, 点阵纸坐标系)
 * @param y Y坐标 (毫米)
 * @param pressure 压力值 (0.0-1.0, 归一化)
 * @param timestamp 时间戳 (毫秒)
 */
```

```
data class StrokePoint(
    val x: Float,
    val y: Float,
    val pressure: Float = 0.5f,
    val timestamp: Long = 0L
)
```

```
/**
 * 笔画数据 (一次落笔到抬笔的完整轨迹)
 * @param studentId 学生标识 (用于颜色区分)
 * @param points 坐标点列表
 * @param color 笔迹颜色
 * @param baseWidth 基础笔画宽度 (像素)
 */
```

```
data class Stroke(
    val studentId: String,
    val points: MutableList<StrokePoint> = mutableListOf(),

```



```

        val color: Int = Color.BLACK,
        val baseWidth: Float = 3.0f
    )

/**
 * 学生笔迹颜色分配表
 * 预定义12种高对比度颜色，确保大屏上可区分
 */
object StudentColorPalette {
    private val colors = intArrayOf(
        Color.parseColor("#1976D2"), // 蓝色
        Color.parseColor("#D32F2F"), // 红色
        Color.parseColor("#388E3C"), // 绿色
        Color.parseColor("#F57C00"), // 橙色
        Color.parseColor("#7B1FA2"), // 紫色
        Color.parseColor("#00838F"), // 青色
        Color.parseColor("#C2185B"), // 粉色
        Color.parseColor("#455A64"), // 灰蓝
        Color.parseColor("#795548"), // 棕色
        Color.parseColor("#0097A7"), // 深青
        Color.parseColor("#689F38"), // 草绿
        Color.parseColor("#FF6F00"), // 深橙
    )

    /** 根据学生索引获取颜色 */
    fun getColor(studentIndex: Int): Int {
        return colors[studentIndex % colors.size]
    }

    /** 根据学生ID哈希获取颜色 */
    fun getColorForStudent(studentId: String): Int {
        val hash = studentId.hashCode() and 0x7FFFFFFF
        return colors[hash % colors.size]
    }
}

/**
 * 笔迹渲染器 - 基于SurfaceView的高性能大屏笔迹渲染
 *
 * 采用双缓冲策略：
 * - 后缓冲 (offscreenBitmap): 存储已确认的历史笔迹
 * - 前缓冲 (SurfaceView Canvas): 在后缓冲基础上绘制当前活跃笔画
 *
 * 这样每帧只需绘制当前正在书写的笔画，大幅减少重绘开销
 */
class StrokeRenderer @JvmOverloads constructor(
    context: Context,
    attrs: AttributeSet? = null,
    defStyleAttr: Int = 0
) : SurfaceView(context, attrs, defStyleAttr), SurfaceHolder.Callback {

    companion object {
        private const val TAG = "StrokeRenderer"

        /** 目标帧率 */
        private const val TARGET_FPS = 60
    }
}

```

```

/** 帧间隔（毫秒） */
private const val FRAME_INTERVAL_MS = 1000L / TARGET_FPS

/** 坐标系缩放比例（毫米到像素的转换系数） */
private const val MM_TO_PX = 4.0f

/** 贝塞尔曲线平滑张力系数 */
private const val BEZIER_TENSION = 0.25f

/** 笔锋效果-落笔过渡点数 */
private const val PEN_DOWN_TRANSITION = 5

/** 笔锋效果-抬笔过渡点数 */
private const val PEN_UP_TRANSITION = 5
}

/** 已完成的笔画列表（线程安全） */
private val completedStrokes = CopyOnWriteArrayList<Stroke>()

/** 当前正在书写的活跃笔画（按学生ID索引） */
private val activeStrokes = ConcurrentHashMap<String, Stroke>()

/** 离屏缓冲Bitmap（存储历史笔迹） */
private var offscreenBitmap: android.graphics.Bitmap? = null
private var offscreenCanvas: Canvas? = null

/** 渲染线程 */
private var renderThread: RenderThread? = null

/** Surface是否可用 */
private var surfaceReady = false

/** 画布宽高 */
private var canvasWidth = 0
private var canvasHeight = 0

/** 缩放和平移参数（遥控器控制） */
private var scaleX = 1.0f
private var scaleY = 1.0f
private var translateX = 0.0f
private var translateY = 0.0f

/** 绘制用Paint对象（复用避免GC） */
private val strokePaint = Paint().apply {
    isAntiAlias = true
    style = Paint.Style.STROKE
    strokeCap = Paint.Cap.ROUND
    strokeJoin = Paint.Join.ROUND
}

private val backgroundPaint = Paint().apply {
    color = Color.WHITE
    style = Paint.Style.FILL
}

/** 复用Path对象 */
private val reusablePath = Path()

```

```

/** 是否需要刷新离屏缓冲 */
private var needsRefreshOffscreen = false

init {
    holder.addCallback(this)
    // 设置透明背景（支持叠加在课件内容上方）
    setZOrderOnTop(false)
}

/* ===== SurfaceHolder.Callback ===== */

override fun surfaceCreated(holder: SurfaceHolder) {
    surfaceReady = true
    canvasWidth = holder.surfaceFrame.width()
    canvasHeight = holder.surfaceFrame.height()

    // 创建离屏缓冲（与Surface同尺寸）
    offscreenBitmap = android.graphics.Bitmap.createBitmap(
        canvasWidth, canvasHeight, android.graphics.Bitmap.Config.ARGB_8888
    )
    offscreenCanvas = Canvas(offscreenBitmap!!)
    offscreenCanvas?.drawRect(0f, 0f, canvasWidth.toFloat(), canvasHeight.toFloat(),
backgroundPaint)

    // 启动渲染线程
    renderThread = RenderThread()
    renderThread?.start()

    // 如果已有历史笔迹数据，先渲染到离屏缓冲
    if (completedStrokes.isNotEmpty()) {
        rebuildOffscreenCache()
    }

    Log.i(TAG, "Surface创建完成: ${canvasWidth}x${canvasHeight}")
}

override fun surfaceChanged(holder: SurfaceHolder, format: Int, width: Int, height:
Int) {
    canvasWidth = width
    canvasHeight = height
    // 重建离屏缓冲以匹配新尺寸
    offscreenBitmap?.recycle()
    offscreenBitmap = android.graphics.Bitmap.createBitmap(
        width, height, android.graphics.Bitmap.Config.ARGB_8888
    )
    offscreenCanvas = Canvas(offscreenBitmap!!)
    rebuildOffscreenCache()
    Log.i(TAG, "Surface尺寸变化: ${width}x${height}")
}

override fun surfaceDestroyed(holder: SurfaceHolder) {
    surfaceReady = false
    renderThread?.stopRendering()
    renderThread = null
    offscreenBitmap?.recycle()
    offscreenBitmap = null
}

```

```

        Log.i(TAG, "Surface已销毁")
    }

    /* ===== 公开API ===== */

    /**
     * 添加笔迹点（由WebSocket接收器调用）
     * @param studentId 学生标识
     * @param point 坐标点
     * @param isPenDown true=落笔（笔画开始），false=行笔中
     */
    fun addStrokePoint(studentId: String, point: StrokePoint, isPenDown: Boolean) {
        if (isPenDown) {
            // 新建笔画
            val color = StudentColorPalette.getColorForStudent(studentId)
            val stroke = Stroke(studentId = studentId, color = color)
            stroke.points.add(point)
            activeStrokes[studentId] = stroke
        } else {
            // 添加到当前活跃笔画
            activeStrokes[studentId]?.points?.add(point)
        }
    }

    /**
     * 完成一个笔画（抬笔事件）
     * 将活跃笔画移入已完成列表，并渲染到离屏缓冲
     */
    fun finishStroke(studentId: String) {
        val stroke = activeStrokes.remove(studentId) ?: return
        if (stroke.points.size >= 2) {
            completedStrokes.add(stroke)
            // 将新完成的笔画绘制到离屏缓冲
            offscreenCanvas?.let { canvas ->
                drawSingleStroke(canvas, stroke)
            }
        }
    }

    /** 清除所有笔迹 */
    fun clearAll() {
        completedStrokes.clear()
        activeStrokes.clear()
        offscreenCanvas?.drawRect(0f, 0f, canvasWidth.toFloat(), canvasHeight.toFloat(),
backgroundPaint)
        Log.i(TAG, "所有笔迹已清除")
    }

    /** 清除指定学生的笔迹 */
    fun clearStudentStrokes(studentId: String) {
        activeStrokes.remove(studentId)
        completedStrokes.removeAll { it.studentId == studentId }
        rebuildOffscreenCache()
    }

    /** 设置显示缩放（遥控器方向键操作） */
    fun setZoom(scale: Float) {

```

```

        scaleX = scale.coerceIn(0.5f, 5.0f)
        scaleY = scaleX
    }

    /** 设置显示平移 */
    fun setPan(dx: Float, dy: Float) {
        translateX += dx
        translateY += dy
    }

    /** ===== 渲染逻辑 ===== */

    /** 重建离屏缓冲（将所有已完成笔画重新绘制） */
    private fun rebuildOffscreenCache() {
        val canvas = offscreenCanvas ?: return
        canvas.drawRect(0f, 0f, canvasWidth.toFloat(), canvasHeight.toFloat(),
backgroundPaint)
        for (stroke in completedStrokes) {
            drawSingleStroke(canvas, stroke)
        }
        Log.d(TAG, "离屏缓冲重建完成, 笔画数: ${completedStrokes.size}")
    }

    /**
     * 绘制单个笔画（贝塞尔平滑 + 压力笔锋）
     * 采用分段绘制策略：每两个相邻点之间用三次贝塞尔曲线连接
     */
    private fun drawSingleStroke(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]

            // 根据压力计算笔画宽度（笔锋效果）
            val width = calculateStrokeWidth(
                stroke.baseWidth, prev.pressure, curr.pressure,
                i, points.size
            )
            strokePaint.strokeWidth = width * MM_TO_PX

            if (i >= 2 && i < points.size) {
                // 三次贝塞尔曲线平滑
                val pp = points[i - 2]
                val cp1x = prev.x * MM_TO_PX + (curr.x - pp.x) * MM_TO_PX *
BEZIER_TENSION
                val cp1y = prev.y * MM_TO_PX + (curr.y - pp.y) * MM_TO_PX *
BEZIER_TENSION
                val cp2x = curr.x * MM_TO_PX - (curr.x - prev.x) * MM_TO_PX *
BEZIER_TENSION
                val cp2y = curr.y * MM_TO_PX - (curr.y - prev.y) * MM_TO_PX *
BEZIER_TENSION

                reusablePath.reset()
            }
        }
    }

```

```

        reusablePath.moveTo(prev.x * MM_T0_PX, prev.y * MM_T0_PX)
        reusablePath.cubicTo(cp1x, cp1y, cp2x, cp2y, curr.x * MM_T0_PX, curr.y *
MM_T0_PX)

        canvas.drawPath(reusablePath, strokePaint)
    } else {
        // 前两个点直接连线
        canvas.drawLine(
            prev.x * MM_T0_PX, prev.y * MM_T0_PX,
            curr.x * MM_T0_PX, curr.y * MM_T0_PX,
            strokePaint
        )
    }
}

/**
 * 计算压力感应笔画宽度
 * 模拟真实书写笔锋：落笔由细变粗，行笔随压力变化，抬笔由粗变细
 */
private fun calculateStrokeWidth(
    baseWidth: Float,
    prevPressure: Float,
    currPressure: Float,
    index: Int,
    totalPoints: Int
): Float {
    val avgPressure = (prevPressure + currPressure) / 2.0f

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

    // 落笔过渡效果 (前N个点逐渐增加宽度)
    if (index < PEN_DOWN_TRANSITION) {
        width *= (index.toFloat() / PEN_DOWN_TRANSITION)
    }

    // 抬笔过渡效果 (最后N个点逐渐减小宽度)
    val remaining = totalPoints - index
    if (remaining < PEN_UP_TRANSITION) {
        width *= (remaining.toFloat() / PEN_UP_TRANSITION)
    }

    return max(width, 0.5f)
}

/* ===== 渲染线程 ===== */

/**
 * 渲染线程 - 以60fps目标帧率循环渲染
 * 每帧将离屏缓冲绘制到Surface，然后叠加活跃笔画
 */
inner class RenderThread : Thread("StrokeRenderThread") {

    @Volatile
    private var running = true

    fun stopRendering() {

```

```

        running = false
    }

    override fun run() {
        Log.i(TAG, "渲染线程启动")

        while (running && surfaceReady) {
            val frameStart = System.currentTimeMillis()

            try {
                val canvas = holder.lockCanvas() ?: continue
                try {
                    // 步骤1: 绘制离屏缓冲 (历史笔迹)
                    offscreenBitmap?.let { bitmap ->
                        canvas.save()
                        canvas.translate(translateX, translateY)
                        canvas.scale(scaleX, scaleY)
                        canvas.drawBitmap(bitmap, 0f, 0f, null)
                        canvas.restore()
                    }

                    // 步骤2: 绘制当前活跃笔画 (正在书写的)
                    canvas.save()
                    canvas.translate(translateX, translateY)
                    canvas.scale(scaleX, scaleY)
                    for (stroke in activeStrokes.values) {
                        if (stroke.points.size >= 2) {
                            drawSingleStroke(canvas, stroke)
                        }
                    }
                    canvas.restore()
                } finally {
                    holder.unlockCanvasAndPost(canvas)
                }
            } catch (e: Exception) {
                Log.e(TAG, "渲染帧异常: ${e.message}")
            }

            // 帧率控制: 等待到下一帧时间
            val elapsed = System.currentTimeMillis() - frameStart
            val sleepTime = FRAME_INTERVAL_MS - elapsed
            if (sleepTime > 0) {
                try {
                    sleep(sleepTime)
                } catch (_: InterruptedException) {
                    break
                }
            }
        }

        Log.i(TAG, "渲染线程已停止")
    }
}

/** 释放资源 */
fun release() {
    renderThread?.stopRendering()
}

```

```

        renderThread = null
        offscreenBitmap?.recycle()
        offscreenBitmap = null
        completedStrokes.clear()
        activeStrokes.clear()
        Log.i(TAG, "渲染器资源已释放")
    }
}

```

**ui/**

**ui/MainFragment.kt**

```

/**
 * 自然写互动课堂电视端应用软件 V1.0
 * Leanback主界面Fragment - Android TV主界面导航
 *
 * 功能说明:
 * 1. Leanback BrowseSupportFragment主界面布局
 * 2. D-Pad遥控器焦点导航适配 (方向键/确认键/返回键)
 * 3. 多功能区域展示 (课堂笔迹、互动答题、学情报告、设置)
 * 4. 课堂状态实时显示 (当前课堂信息、在线学生数)
 * 5. 语音操控集成 (Android TV语音搜索)
 * 6. 网关连接状态指示
 * 7. 自动全屏沉浸式模式
 */

package com.writech.tv.ui

import android.content.Context
import android.graphics.Color
import android.os.Bundle
import android.os.Handler
import android.os.Looper
import android.util.Log
import android.view.KeyEvent
import android.view.View
import android.view.WindowManager
import android.widget.Toast
import java.text.SimpleDateFormat
import java.util.*

/**
 * TV端主界面数据模型 - 功能卡片
 */
data class FunctionCard(
    val id: String,           // 卡片唯一标识
    val title: String,        // 标题
    val description: String,   // 描述
    val iconRes: Int,          // 图标资源ID
    val category: String,      // 所属分类
    val action: String         // 点击动作标识
)

```



```

/**
 * 课堂状态信息
 */
data class ClassroomStatus(
    var isActive: Boolean = false,           // 是否有进行中的课堂
    var classId: String = "",               // 课堂ID
    var className: String = "",             // 课堂名称
    var teacherName: String = "",           // 授课教师
    var onlineStudentCount: Int = 0,        // 在线学生数
    var totalStudentCount: Int = 0,         // 总学生数
    var startTime: Long = 0,                // 课堂开始时间
    var currentSubject: String = ""         // 当前科目
)

/**
 * TV端Leanback主界面Fragment
 * 采用Android TV Leanback库的BrowseSupportFragment风格
 * 适配遥控器D-Pad焦点导航操作
 */
class MainFragment {

    companion object {
        private const val TAG = "MainFragment"

        // 功能分类ID
        private const val CATEGORY_CLASSROOM = "classroom"
        private const val CATEGORY_INTERACTIVE = "interactive"
        private const val CATEGORY_REPORT = "report"
        private const val CATEGORY_SETTINGS = "settings"
    }

    /** 当前课堂状态 */
    private val classroomStatus = ClassroomStatus()

    /** 功能卡片列表（按分类组织） */
    private val functionCards = mutableMapOf<String, MutableList<FunctionCard>>()

    /** 主线程Handler */
    private val handler = Handler(Looper.getMainLooper())

    /** 课堂计时器 */
    private var classroomTimer: Timer? = null

    /** 日期格式化器 */
    private val dateFormat = SimpleDateFormat("HH:mm:ss", Locale.CHINA)

    /**
     * 初始化界面
     * 配置Leanback样式、加载功能卡片、设置焦点导航
     */
    fun initialize() {
        // 配置Leanback主题色
        // brandColor = Color.parseColor("#1976D2")
        // searchAffordanceColor = Color.parseColor("#2196F3")

        // 加载功能卡片数据
    }

```

```

loadFunctionCards()

// 设置搜索回调（语音搜索）
setupSearch()

// 设置全屏沉浸式模式
setupImmersiveMode()

Log.i(TAG, "主界面初始化完成")
}

/**
 * 加载功能卡片列表
 * 按分类组织：课堂展示、互动答题、学情报告、系统设置
 */
private fun loadFunctionCards() {
    // 课堂展示功能
    val classroomCards = mutableListOf(
        FunctionCard(
            id = "stroke_display",
            title = "全班笔迹实时展示",
            description = "大屏展示全班学生实时书写笔迹",
            iconRes = 0, // R.drawable.ic_stroke_display
            category = CATEGORY_CLASSROOM,
            action = "open_stroke_display"
        ),
        FunctionCard(
            id = "multi_compare",
            title = "多学生同屏对比",
            description = "选择学生笔迹并排对比展示",
            iconRes = 0,
            category = CATEGORY_CLASSROOM,
            action = "open_multi_compare"
        ),
        FunctionCard(
            id = "copybook_display",
            title = "字帖临摹展示",
            description = "放大范字与学生实时书写对比",
            iconRes = 0,
            category = CATEGORY_CLASSROOM,
            action = "open_copybook"
        ),
        FunctionCard(
            id = "stroke_replay",
            title = "笔迹回放",
            description = "回放学生书写过程（支持变速）",
            iconRes = 0,
            category = CATEGORY_CLASSROOM,
            action = "open_replay"
        )
    )

    // 课堂互动功能
    val interactiveCards = mutableListOf(
        FunctionCard(
            id = "quiz_display",
            title = "答题结果展示",

```

```

        description = "大屏展示课堂互动答题统计",
        iconRes = 0,
        category = CATEGORY_INTERACTIVE,
        action = "open_quiz_display"
    ),
    FunctionCard(
        id = "random_pick",
        title = "随机点名",
        description = "随机抽取学生进行展示",
        iconRes = 0,
        category = CATEGORY_INTERACTIVE,
        action = "open_random_pick"
    ),
    FunctionCard(
        id = "group_display",
        title = "分组展示",
        description = "按小组展示学生作品",
        iconRes = 0,
        category = CATEGORY_INTERACTIVE,
        action = "open_group_display"
    )
)

// 学情报告功能
val reportCards = mutableListOf(
    FunctionCard(
        id = "class_report",
        title = "班级学情概览",
        description = "班级整体学情数据大屏展示",
        iconRes = 0,
        category = CATEGORY_REPORT,
        action = "open_class_report"
    ),
    FunctionCard(
        id = "student_report",
        title = "学生学情详情",
        description = "单个学生学情画像详细展示",
        iconRes = 0,
        category = CATEGORY_REPORT,
        action = "open_student_report"
    ),
    FunctionCard(
        id = "growth_chart",
        title = "书写成长轨迹",
        description = "学生书写能力变化趋势图",
        iconRes = 0,
        category = CATEGORY_REPORT,
        action = "open_growth_chart"
    )
)

// 系统设置功能
val settingsCards = mutableListOf(
    FunctionCard(
        id = "gateway_settings",
        title = "网关连接",
        description = "搜索并绑定教室网关设备",

```

```

        iconRes = 0,
        category = CATEGORY_SETTINGS,
        action = "open_gateway_settings"
    ),
    FunctionCard(
        id = "display_settings",
        title = "显示设置",
        description = "分辨率、字体大小、背景色调整",
        iconRes = 0,
        category = CATEGORY_SETTINGS,
        action = "open_display_settings"
    ),
    FunctionCard(
        id = "network_settings",
        title = "网络设置",
        description = "WiFi连接、云平台地址配置",
        iconRes = 0,
        category = CATEGORY_SETTINGS,
        action = "open_network_settings"
    ),
    FunctionCard(
        id = "about",
        title = "关于",
        description = "版本信息、设备ID、软件许可",
        iconRes = 0,
        category = CATEGORY_SETTINGS,
        action = "open_about"
    )
)

functionCards[CATEGORY_CLASSROOM] = classroomCards
functionCards[CATEGORY_INTERACTIVE] = interactiveCards
functionCards[CATEGORY_REPORT] = reportCards
functionCards[CATEGORY_SETTINGS] = settingsCards

Log.i(TAG, "功能卡片加载完成, 共${functionCards.values.sumOf { it.size }}个")
}

/**
 * 处理功能卡片点击事件
 * 根据action标识跳转到对应的功能Fragment
 */
fun onCardSelected(card: FunctionCard) {
    Log.i(TAG, "选中功能: ${card.title} -> ${card.action}")
    when (card.action) {
        "open_stroke_display" -> navigateToStrokeDisplay()
        "open_multi_compare" -> navigateToMultiCompare()
        "open_copybook" -> navigateToCopybookDisplay()
        "open_replay" -> navigateToReplay()
        "open_quiz_display" -> navigateToQuizDisplay()
        "open_random_pick" -> performRandomPick()
        "open_group_display" -> navigateToGroupDisplay()
        "open_class_report" -> navigateToClassReport()
        "open_student_report" -> navigateToStudentReport()
        "open_growth_chart" -> navigateToGrowthChart()
        "open_gateway_settings" -> navigateToGatewaySettings()
        "open_display_settings" -> navigateToDisplaySettings()
    }
}

```

```

        "open_network_settings" -> navigateToNetworkSettings()
        "open_about" -> navigateToAbout()
        else -> Log.w(TAG, "未知操作: ${card.action}")
    }
}

/** 设置语音搜索 (Android TV Voice Search) */
private fun setupSearch() {
    // setOnSearchClickedListener { openSearchFragment() }
    Log.i(TAG, "语音搜索配置完成")
}

/** 设置全屏沉浸式模式 */
private fun setupImmersiveMode() {
    // activity?.window?.addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON)
    // activity?.window?.addFlags(WindowManager.LayoutParams.FLAG_SECURE) // 防截屏
    Log.i(TAG, "沉浸式模式已启用")
}

/**
 * 处理遥控器按键事件
 * 适配D-Pad方向键、确认键、返回键、菜单键
 */
fun onKeyEvent(keyCode: Int, event: KeyEvent): Boolean {
    return when (keyCode) {
        KeyEvent.KEYCODE_DPAD_CENTER, KeyEvent.KEYCODE_ENTER -> {
            // 确认键: 选中当前焦点项
            Log.d(TAG, "遥控器确认键按下")
            false // 交给焦点系统处理
        }
        KeyEvent.KEYCODE_MENU -> {
            // 菜单键: 显示快捷操作面板
            showQuickActions()
            true
        }
        KeyEvent.KEYCODE_MEDIA_PLAY_PAUSE -> {
            // 播放/暂停键: 控制笔迹回放
            toggleReplayPause()
            true
        }
        else -> false
    }
}

/** 显示快捷操作面板 */
private fun showQuickActions() {
    Log.i(TAG, "显示快捷操作面板")
}

/** 切换回放暂停/继续 */
private fun toggleReplayPause() {
    Log.i(TAG, "切换回放状态")
}

/* ===== 课堂状态管理 ===== */

/** 更新课堂状态 */

```

```

fun updateClassroomStatus(status: ClassroomStatus) {
    classroomStatus.isActive = status.isActive
    classroomStatus.classId = status.classId
    classroomStatus.className = status.className
    classroomStatus.teacherName = status.teacherName
    classroomStatus.onlineStudentCount = status.onlineStudentCount
    classroomStatus.totalStudentCount = status.totalStudentCount
    classroomStatus.startTime = status.startTime
    classroomStatus.currentSubject = status.currentSubject

    if (status.isActive) {
        startClassroomTimer()
    } else {
        stopClassroomTimer()
    }

    // 更新Header显示
    updateHeaderInfo()
}

/** 启动课堂计时器（实时显示课堂进行时长） */
private fun startClassroomTimer() {
    stopClassroomTimer()
    classroomTimer = Timer("classroom-timer")
    classroomTimer?.scheduleAtFixedRate(object : TimerTask() {
        override fun run() {
            val elapsed = System.currentTimeMillis() - classroomStatus.startTime
            val minutes = (elapsed / 60000).toInt()
            val seconds = ((elapsed % 60000) / 1000).toInt()
            val timeStr = String.format("%02d:%02d", minutes, seconds)
            handler.post {
                // 更新课堂时长显示
                Log.d(TAG, "课堂进行: $timeStr")
            }
        }
    }, 0, 1000)
}

/** 停止课堂计时器 */
private fun stopClassroomTimer() {
    classroomTimer?.cancel()
    classroomTimer = null
}

/** 更新顶部标题栏信息 */
private fun updateHeaderInfo() {
    val title = if (classroomStatus.isActive) {
        "${classroomStatus.className} - ${classroomStatus.currentSubject}" +
        "
    ( ${classroomStatus.onlineStudentCount}/${classroomStatus.totalStudentCount}人在线)"
    } else {
        "自然写互动课堂"
    }
    // 设置标题
    Log.i(TAG, "更新标题: $title")
}

```

```

/** 执行随机点名 */
private fun performRandomPick() {
    if (!classroomStatus.isActive) {
        Log.w(TAG, "当前无进行中的课堂, 无法随机点名")
        return
    }
    // 从在线学生列表中随机抽取
    Log.i(TAG, "执行随机点名")
}

/* ===== 导航方法 ===== */

private fun navigateToStrokeDisplay() { Log.i(TAG, "跳转: 全班笔迹展示") }
private fun navigateToMultiCompare() { Log.i(TAG, "跳转: 多学生对比") }
private fun navigateToCopybookDisplay() { Log.i(TAG, "跳转: 字帖临摹") }
private fun navigateToReplay() { Log.i(TAG, "跳转: 笔迹回放") }
private fun navigateToQuizDisplay() { Log.i(TAG, "跳转: 答题展示") }
private fun navigateToGroupDisplay() { Log.i(TAG, "跳转: 分组展示") }
private fun navigateToClassReport() { Log.i(TAG, "跳转: 班级学情") }
private fun navigateToStudentReport() { Log.i(TAG, "跳转: 学生学情") }
private fun navigateToGrowthChart() { Log.i(TAG, "跳转: 成长轨迹") }
private fun navigateToGatewaySettings() { Log.i(TAG, "跳转: 网关设置") }
private fun navigateToDisplaySettings() { Log.i(TAG, "跳转: 显示设置") }
private fun navigateToNetworkSettings() { Log.i(TAG, "跳转: 网络设置") }
private fun navigateToAbout() { Log.i(TAG, "跳转: 关于") }

/** 释放资源 */
fun release() {
    stopClassroomTimer()
    functionCards.clear()
    Log.i(TAG, "主界面资源已释放")
}
}

```