01.功能概述

- **功能ID**：`FEAT-20250709-001`
- **功能名称**：
- **目标版本**：
- **提交人**：@panruiqi
- **状态**：
  - [x] ⏳ 设计中 /
  - [ ] ⏳ 开发中 /
  - [ ] ☑ 已完成 /
  - [ ] ❌ 已取消
- **价值评估**：
  - [ ] ★★★★★ 核心业务功能
  - [x] ★★★★ 用户体验优化
  - [ ] ★★★ 辅助功能增强
  - [ ] ★★ 技术债务清理
- **功能描述**
  - 扫码枪一直扫码，不符合我们的期望，这种频率太高。

- 我们希望：
  - 对于相同的码，降低检测频率（智能防抖）
  - 对于码的切换，保持灵敏（快速响应）
  - 防止误操作和重复加购

02.需求分析

2.1 用户场景

- **主要场景**：
  - 用户用扫码枪快速扫码，最开始一直扫描一件商品，一直滴滴滴响
  - 中间换成另一件商品时，希望立即响应
  - 同一件商品连续扫码时，希望降低频率，避免重复加购
- **边界场景**：
  - 扫码枪故障导致的连续触发
  - 用户故意快速切换不同商品
  - 网络延迟导致的响应慢但用户连续扫码
  - 支付过程中误扫码干扰

2.2 功能范围

- ☑ 包含：
  - 同一商品码的智能防抖（指数退避）
  - 不同商品码的快速切换检测
  - 扫码频率自适应调节
  - 支付状态下的扫码拦截
- ❌ 不包含：
  - 扫码枪硬件配置
  - 商品数据验证逻辑
  - 网络重试机制

03.技术方案

3.1 方案一：时间窗口 + 内容去重（简单有效）

- 实现思路：
  - 维护最近扫码记录（内容+时间戳）
  - 相同内容在时间窗口内只处理一次
  - 不同内容立即处理，重置时间窗口
  - ```
    class SimpleScanDebouncer {
        private var lastScanContent = ""
        private var lastScanTime = 0L
        private val debounceWindow = 1000L // 1秒防抖窗口

        fun shouldProcess(content: String): Boolean {
            val now = System.currentTimeMillis()
    ```
    ```

```
        return if (content == lastScanContent) {
            // 相同内容：检查时间间隔
            if (now - lastScanTime > debounceWindow) {
                lastScanTime = now
                true
            } else false
        } else {
            // 不同内容：立即处理
            lastScanContent = content
            lastScanTime = now
            true
        }
    }
}
```

- 优点： 简单直观，代码量少，性能好
- 缺点： 固定时间窗口，不够智能

3.2 方案二：指数退避 + 智能频率调节（推荐）

- 实现思路：
  - 使用指数退避算法动态调整同一商品的扫码间隔
  - 连续扫码同一商品时，间隔逐渐增加：100ms → 200ms → 400ms → 800ms
  - 切换到不同商品时立即重置，保持灵敏度
  - ```
    class IntelligentScanDebouncer {
        private data class ScanRecord(
            val content: String,
            val lastTime: Long,
            val consecutiveCount: Int,
            val currentInterval: Long
        )

        private var scanRecord: ScanRecord? = null
        private val baseInterval = 100L // 基础间隔100ms
        private val maxInterval = 2000L  // 最大间隔2秒
        private val multiplier = 2       // 指数倍数

        fun shouldProcess(content: String): Boolean {
            val now = System.currentTimeMillis()
            val current = scanRecord

            return if (current == null || current.content != content) {
                // 首次扫码或切换商品：立即处理
                scanRecord = ScanRecord(content, now, 1, baseInterval)
                true
            } else {
                // 相同商品：检查指数退避间隔
                if (now - current.lastTime >= current.currentInterval) {
                    val newInterval = minOf(
                        current.currentInterval * multiplier,
                        maxInterval
                    )
                    scanRecord = current.copy(
                        lastTime = now,
                        consecutiveCount = current.consecutiveCount + 1,
    ```
    ```

```
                    currentInterval = newInterval
                )
                true
            } else false
        }
    }

    fun reset() {
        scanRecord = null
    }
}
```

- 优点：智能自适应，用户体验好，符合需求
- 缺点：逻辑稍复杂，内存占用略高

3.3 方案三：状态机 + 多层防抖

- 实现思路：
  - 定义扫码状态：空闲、单商品连扫、快速切换、冷却等
  - 每个状态有不同的防抖策略
  - 支持复杂业务场景的扩展
  - ```
    sealed class ScanState {
        // 空闲状态：没有扫码活动
        object Idle : ScanState()

        // 连续扫码状态：同一商品重复扫码
        data class ContinuousScanning(
            val content: String,           // 当前扫码内容
            val count: Int,                // 连续扫码次数
            val interval: Long,            // 当前防抖间隔
            val lastScanTime: Long         // 上次扫码时间
        ) : ScanState()

        // 快速切换状态：短时间内扫码不同商品
        data class RapidSwitching(
            val switchHistory: List<String>, // 最近切换的商品历史
            val lastSwitchTime: Long,        // 上次切换时间
            val currentContent: String       // 当前商品
        ) : ScanState()

        // 冷却状态：强制等待，防止过度扫码
        data class Cooldown(
            val until: Long,               // 冷却结束时间
            val reason: CooldownReason     // 冷却原因
        ) : ScanState()

        // 错误状态：检测到异常扫码行为
        data class Error(
            val errorType: ErrorType,      // 错误类型
            val recoverTime: Long          // 恢复时间
        ) : ScanState()
    }

    enum class CooldownReason {
        TOO_FREQUENT,       // 扫码过于频繁
    ```
    ```

```
    RAPID_SWITCHING,    // 快速切换过多
    SYSTEM_BUSY         // 系统繁忙
}

enum class ErrorType {
    INVALID_CONTENT,    // 无效内容
    HARDWARE_ERROR,     // 硬件错误
    TIMEOUT            // 超时
}
```

```
    [Idle]
       ↓ (首次扫码)
[ContinuousScanning] ↔ [RapidSwitching]
       ↓ (过于频繁)         ↓ (切换过多)
   [Cooldown] ←----------[Cooldown]
       ↓ (超时/错误)
    [Error]
       ↓ (恢复)
    [Idle]
```

- 优点：功能强大，扩展性好，支持复杂场景
- 缺点：代码复杂度高，维护成本高

3.4 方案四：协程 + Flow 响应式防

实现思路：

- 使用Kotlin Flow的防抖操作符
- 结合协程实现异步处理
- 支持取消和背压处理

```
class FlowBasedScanDebouncer {
    private val scanEvents = MutableSharedFlow<ScanEvent>(
        extraBufferCapacity = 100
    )

    data class ScanEvent(val content: String, val timestamp: Long)

    init {
        // 设置Flow处理管道
        scanEvents
            .groupBy { it.content } // 按内容分组
            .forEach { (content, flow) ->
                flow
                    .debounce { event ->
                        calculateDebounceTime(content, event.timestamp)
                    }
                    .onEach { event ->
                        processScanEvent(event)
                    }
                    .launchIn(GlobalScope)
            }
    }
```

```
    fun submitScan(content: String) {
        scanEvents.tryEmit(ScanEvent(content, System.currentTimeMillis()))
    }

    private fun calculateDebounceTime(content: String, timestamp: Long):
Long {
        // 动态计算防抖时间
        return getAdaptiveInterval(content)
    }
}
```

- 优点：代码优雅，异步处理，符合响应式编程
- 缺点：学习成本高，调试复杂

04.实现规划

4.1 技术选型

4.2 任务拆解

4.3 代码路径

05.兼容性设计

5.1 设备适配

5.2 冲突检查

06.测试方案

6.1 核心用例

6.2 性能指标

07.发布计划

7.1 阶段发布

7.2 回滚方案

08.文档记录

8.1 技术文档

8.2 用户文档

8.3 监控埋点