

```
+++
date = '2025-09-29T10:22:54+08:00'
draft = true
title = 'SplitScreenController flow'
+++
```

Android Window Brightness Override 技术实现细节深度解析

本文将深入剖析 Android 系统中 **Window Brightness Override**（窗口亮度覆盖）机制的实现原理。基于提供的系统源码文件（`RootWindowContainer.java`、`DisplayPowerController.java` 等），我们将详细还原从应用层发起请求到底层硬件执行的全链路流程，特别是针对包含 **Voyah Porting** 定制逻辑的实现进行重点分析。

1. 机制概述

Window Brightness Override 允许当前前台窗口（Window）通过设置 `WindowManager.LayoutParams.screenBrightness` 来临时接管屏幕亮度控制权（例如视频播放器或二维码展示页面）。

在该定制实现中，整个流程表现为**“WMS 决策、DPC 仲裁、双通道下发”**的特征。

2. 核心调用流程详解

第一阶段：WMS 层的布局与决策

一切始于应用层调用 `Window.setAttributes()`，这会触发 `WindowManagerService` 的 `reLayoutWindow`。真正的亮度决策逻辑发生在 WMS 的布局（Layout）阶段。

1. 触发布局：WMS 主循环调用 `RootWindowContainer.performSurfacePlacementNoTrace()` 开始遍历窗口层级。

2. 数据收集：

在 `performSurfacePlacementNoTrace` 中，代码会清理上一轮的缓存 `mDisplayBrightnessOverrides.clear()`，然后调用 `applySurfaceChangesTransaction()`。

在此过程中，系统遍历所有窗口，调用 `handleNotObscuredLocked` 方法。该方法会检查窗口是否可见、是否被遮挡，并提取 `w.mAttrs.screenBrightness`。如果该值在有效范围内（0.0-1.0），则将其存入 `mDisplayBrightnessOverrides` 稀疏数组中。

3. 异步发送：

为了避免持有 WMS 全局锁时调用外部服务导致死锁，`RootWindowContainer` 并没有直接调用 DMS，而是通过内部的 `MyHandler` 发送消息：

```
mHandler.obtainMessage(SET_SCREEN_BRIGHTNESS_OVERRIDE, mDisplayBrightnessOverrides.clone()).sendToTarget();
```

4. 跨服务调用：

`MyHandler` 处理该消息，最终调用

```
mWmService.mDisplayManagerInternal.setScreenBrightnessOverrideFromWindowManager(brightnessOverrides)。
```

第二阶段：DMS 到 DPC 的异步分发

`DisplayManagerService` (DMS) 接收到覆盖请求后，会将其转发给对应的 `DisplayPowerController` (DPC)。

1. **DPC 接收消息**：DPC 的 `setBrightnessOverrideRequest` 方法被调用，它同样不直接处理，而是发送 `MSG_SET_WINDOW_MANAGER_BRIGHTNESS_OVERRIDE` 到 DPC 的 Handler 线程。

2. **策略更新**：

在 DPC 的 Handler 中，调用 `mDisplayBrightnessController.updateWindowManagerBrightnessOverride` 更新内部策略。如果覆盖值发生变化，则触发核心方法 `updatePowerState()`。

第三阶段：DPC 策略仲裁 (`updatePowerState`)

`updatePowerStateInternal()` 是 DPC 的心脏。在这里，系统决定最终使用哪个亮度值。

1. PMS 请求 vs WMS 覆盖:

通常 PowerManagerService 会发送一个 `DisplayPowerRequest`。但在 Window Override 场景下，PMS 发送的 `screenBrightnessOverride` 通常为 NaN (无效)。

2. 策略合并:

DPC 内部逻辑会发现 PMS 的请求无效，转而使用从 WMS 接收并缓存的 `mWindowManagerBrightnessOverride` (例如 0.8)。

3. 计算最终值:

经过自动亮度策略 (此时被禁用)、HBM (高亮模式) 限制、热缓解限制等计算后，得出最终的 `animateValue` (目标亮度)。

第四阶段：执行与双通道下发 (定制逻辑)

在标准的 AOSP 实现中，这里会启动一个 `RampAnimator` 进行平滑的亮度渐变。但在提供的代码中，我们观察到了显著的 **Voyah Porting** 定制修改。

代码调用了 `animateScreenBrightness`，但内部逻辑被修改为直接调用 `directSetScreenBrightness`。

```
// DisplayPowerController.java 伪代码分析
private boolean directSetScreenBrightness(float targetBrightness, float sdrTarget, float rate) {
    // 1. 拦截检查 (如熄屏状态下不设置)
    if (shouldInterceptBrightnessSet(targetBrightness)) { return false; }

    // 2. 通道 A: 直接下发硬件配置 (Real Hardware)
    boolean ret = BrightnessUtils.setBrightnessToConfig(mDisplayId, targetBrightness, reason);

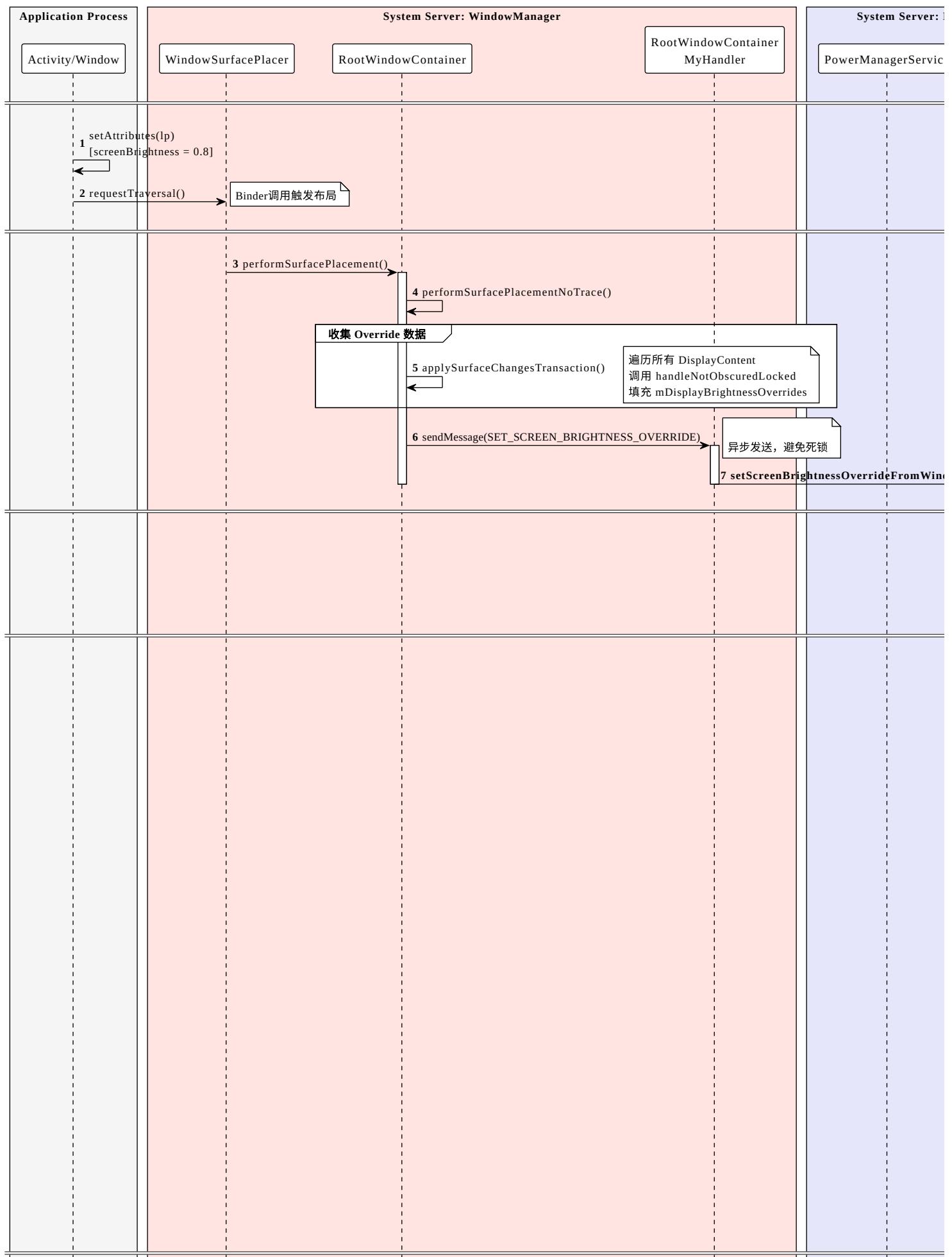
    // 3. 通道 B: 同步框架状态 (Framework State)
    if (mPowerState != null) {
        mPowerState.setScreenBrightness(targetBrightness);
        mPowerState.setSdrScreenBrightness(sdrTarget);
    }
    return true;
}
```

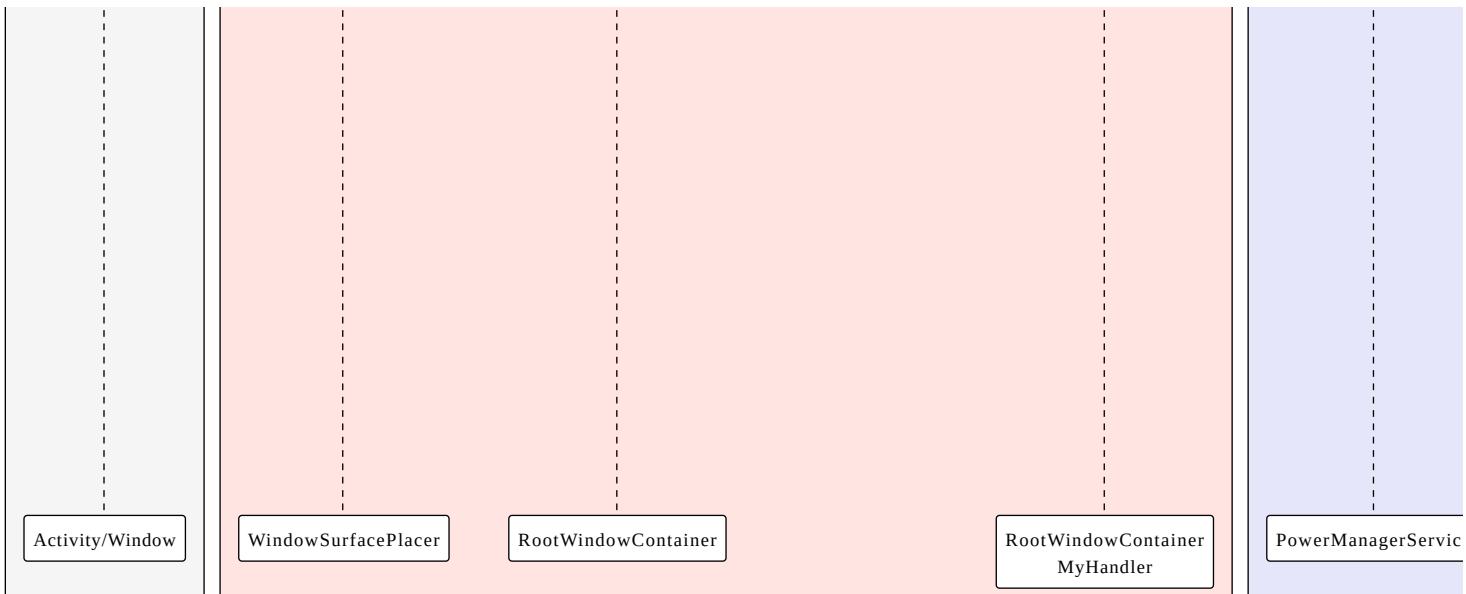
这一步实现了**“双通道下发”**：

- **通道 A (Hardware - 实):** 调用 `BrightnessUtils.setBrightnessToConfig`。这通常通过 Binder 调用底层的 CarService 或专用 HAL 接口，**直接驱动屏幕背光变化**。这种方式绕过了传统的 LightsService。
- **通道 B (Framework - 虚):** 调用 `mPowerState.setScreenBrightness`。这非常重要，它会触发 PhotonicModulator 线程，最终调用 `SurfaceControl.setDisplayBrightness`。虽然此时它不再负责驱动背光，但它**通知了 SurfaceFlinger 当前的亮度值**。这对于 HDR 色调映射、屏幕截图亮度和系统状态同步至关重要。

3. 完整实现时序图

以下是基于上述代码分析生成的精确时序图：





4. 总结

该实现方案展示了典型的 Android 系统深度定制：

1. **解耦与异步**: WMS 通过 MyHandler 异步通知亮度变化，避免了繁重的锁竞争。
2. **绕过与直通**: DPC 劫持了标准的动画流程，通过 BrightnessUtils 实现了对硬件的直接、同步控制。
3. **状态一致性**: 尽管硬件控制被接管，代码依然保留了对 DisplayPowerState 的更新，确保了 Android 上层框架（SurfaceFlinger）的数据一致性。