内容目录

[1 PMS启动流程 1](#__RefHeading__3005_1098183243)

[1.1构造函数 2](#__RefHeading__3007_1098183243)

[1.2 onStart函数 2](#__RefHeading__3009_1098183243)

[1.3 systemReady 3](#__RefHeading__3011_1098183243)

[2 PMS WakeLock使用及流程 4](#__RefHeading__3013_1098183243)

[2.1创建WakeLock 4](#__RefHeading__3015_1098183243)

[2.2 Acquire WakeLock 5](#__RefHeading__4896_1535716620)

[2.2.1监听客户端进程死亡 7](#__RefHeading__4898_1535716620)

[2.2.2特殊处理PARTIAL\_WAKE\_LOCK 7](#__RefHeading__4900_1535716620)

[2.2.3处理WakeLock对应的Flag 8](#__RefHeading__4902_1535716620)

[2.2.4 acquire WakeLock流程图 9](#__RefHeading__4342_1421807576)

[2.2 Release WakeLock 9](#__RefHeading__4344_1421807576)

[2.2.1 release WakeLock code flow 9](#__RefHeading__4346_1421807576)

[2.2.2 Release WakeLock流程图 10](#__RefHeading__4353_1421807576)

[2.3 WakeLock总结 11](#__RefHeading__4355_1421807576)

[3 PMS核心函数updatePowerStateLocked() 11](#__RefHeading__4357_1421807576)

[3.1 mDirty 12](#__RefHeading__4741_1938659111)

[3.2 更新基本状态 12](#__RefHeading__4743_1938659111)

[3.2.1 updateIsPoweredLocked 12](#__RefHeading__4745_1938659111)

[3.2.2 updateStayOnLocked 14](#__RefHeading__4747_1938659111)

[3.2.3 updateScreenBrightnessBoostLocked 14](#__RefHeading__4749_1938659111)

[3.3 更新wakelock和根据用户活动确定屏幕状态 15](#__RefHeading__4751_1938659111)

[3.3.1 updateWakeLockSummaryLocked 15](#__RefHeading__5356_1966077336)

[3.3.2 updateUserActivitySummaryLocked 16](#__RefHeading__5358_1966077336)

[3.3.3 updateWakefulnessLocked 17](#__RefHeading__5440_275729330)

[3.4 更新display power state 18](#__RefHeading__6554_1117857003)

[3.5 更新dream state 20](#__RefHeading__6556_1117857003)

[3.5.1决定是否可以进入屏保状态 20](#__RefHeading__6558_1117857003)

[3.5.2在必要时，进入屏保状态 21](#__RefHeading__6560_1117857003)

[3.5.3更新屏保状态 22](#__RefHeading__6786_1758433336)

[3.6 更新suspend blocker 22](#__RefHeading__6810_1758433336)

[4 PMS Power按键流程 25](#__RefHeading__7832_2135430011)

[4.1 处理按下Power键 - interceptPowerKeyDown 25](#__RefHeading__7834_2135430011)

[4.1.1 判断是否支持长按的行为 26](#__RefHeading__7836_2135430011)

[4.1.2 亮屏时，长按Power键将触发MSG\_POWER\_LONG\_PRESS消息 26](#__RefHeading__7838_2135430011)

[4.1.3 在息屏的状态下按下Power键 27](#__RefHeading__7840_2135430011)

[4.2 处理松开Power键 - interceptPowerKeyUp 27](#__RefHeading__7842_2135430011)

[4.2.1 powerPress 28](#__RefHeading__8129_1044089823)

[4.2.2 finishPowerKeyPress 28](#__RefHeading__8131_1044089823)

[4.3 小结 28](#__RefHeading__8133_1044089823)

[5 PMS之userActivity 29](#__RefHeading__8138_1044089823)

源码位置frameworks/base/services/core/java/com/android/server/power/PowerManagerService.java

**public final class PowerManagerService extends** SystemService

**implements** Watchdog.Monitor {

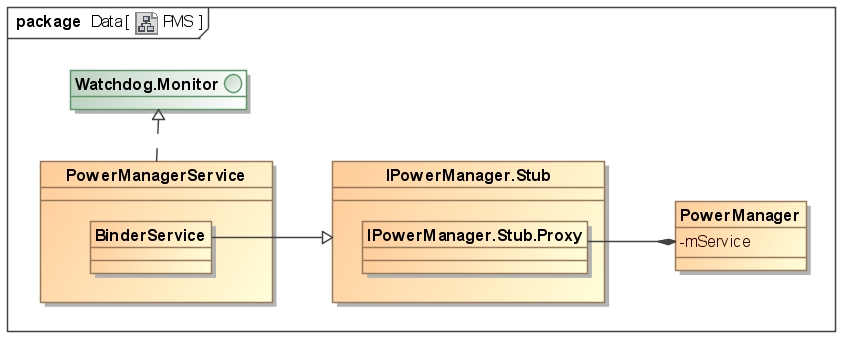
//快6000行了！ -\_-！！！本文就按顺序读代码，然后引出关键变量的话会另起一节

//按照顺序阅读然后再查相关内容，发现阻力很大，还得借助博客去引导

}

**1 PMS启动流程**

第一章侧重于介绍PMS的启动过程，同时对PMS需要打交道的服务建立一个基本的认识



frameworks/base/core/java/android/os/PowerManager.java

frameworks/base/core/java/android/os/IPowerManager.aidl

frameworks/base/services/core/java/com/android/server/power/PowerManagerService.java

frameworks/base/services/core/jni/com\_android\_server\_power\_PowerManagerService.cpp

jni层中方法的具体实现有以下三个独立的部分组成

■ hardware/libhardware\_legacy/power.c这个还在沿用，直接被jni调用，申请/释放锁/system/lib64/libhardware\_legacy.so

■ system/core/libsuspend/ 这个是实现autosuspend\_enable/autosuspend\_disable动态库/system/lib64/libsuspend.so

■ HIDL层，Android O新加入的东东

▪ hardware/interfaces/power/1.0/Ipower.hal **——** 编译出中间文件

out/soong/.intermediates/hardware/interfaces/power/1.0/android.hardware.power@1.0\_genc++/gen/android/hardware/power/1.0/PowerAll.cpp

▪ hardware/interfaces/power/1.0/default/service.cpp

▪ /vendor/bin/hw/android.hardware.power@1.0-service

▪ /vendor/lib64/hw/android.hardware.power@1.0-impl.so

▪ android.hardware.power@1.0-service.rc

■ hardware/interfaces/power/1.0/default/Power.cpp 后面是其继续调用的库文件

这一块是高通独有的实现：

device/qcom/common/power/power.c /vendor/lib64/hw/power.qcom.so

device/qcom/common/power/power-845.c

PMS由SystemServer来启动

frameworks/base/services/java/com/android/server/SystemServer.java

private PowerManagerService mPowerManagerService;

mPowerManagerService = mSystemServiceManager.startService(PowerManagerService.class);//startService主要通过反射调用服务的构造函数，然后再调用服务的onStart函数

mPowerManagerService.systemReady(mActivityManagerService.getAppOpsService());

1.1构造函数

构造函数除了有native和handler的初始化外(老功能架构不变)，还有平台独有的底层实现接口，即HIDL内部方法的具体实现，高通已经将底层实现。

public PowerManagerService(Context context) {

super(context);

mContext = context;

//ServiceThread继承自HandlerThread，专门针对系统服务定义的，应该是优先级更高吧

mHandlerThread = new ServiceThread(TAG, Process.THREAD\_PRIORITY\_DISPLAY, false /\*allowIo\*/);

mHandlerThread.start();

mHandler = new PowerManagerHandler(mHandlerThread.getLooper());

mConstants = new Constants(mHandler);//大概跟了一下，内容提供者和内容观察者，与dumpsys power有关，可以看见内部变量的信息

mAmbientDisplayConfiguration = new AmbientDisplayConfiguration(mContext);//熄屏显示相关

//暂时不分析

mBatterySaverPolicy = new BatterySaverPolicy(mHandler);

mExperienceController = new ExperienceController(mContext);

qcNsrmPowExt = new QCNsrmPowerExtension(this);

mPSMController = new PowerSaveModeController(mContext);

synchronized (mLock) {

//创建一些锁对象，同构acquire和release修改引用数

mWakeLockSuspendBlocker = createSuspendBlockerLocked("PowerManagerService.WakeLocks");

mDisplaySuspendBlocker = createSuspendBlockerLocked("PowerManagerService.Display");

mDisplaySuspendBlocker.acquire();

mHoldingDisplaySuspendBlocker = true;

mHalAutoSuspendModeEnabled = false;

mHalInteractiveModeEnabled = true;

mWakefulness = WAKEFULNESS\_AWAKE;

sQuiescent = SystemProperties.get(SYSTEM\_PROPERTY\_QUIESCENT, "0").equals("1");

nativeInit();//拿出来分析

nativeSetAutoSuspend(false);

nativeSetInteractive(true);

nativeSetFeature(POWER\_FEATURE\_DOUBLE\_TAP\_TO\_WAKE, 0);

}

}

**native\_init**

static void nativeInit(JNIEnv\* env, jobject obj) {

gPowerManagerServiceObj = env->NewGlobalRef(obj);//创建一个全局引用对象，引用PMS

gPowerHalMutex.lock();

getPowerHal();

gPowerHalMutex.unlock();

}

bool getPowerHal() {//获得power hidl的接口，这是给vendor做出来的接口，之前的函数仍旧以动态库的方式编译时做好了链接

if (gPowerHalExists && gPowerHalV1\_0 == nullptr) {

gPowerHalV1\_0 = android::hardware::power::V1\_0::IPower::getService();

if (gPowerHalV1\_0 != nullptr) {

gPowerHalV1\_1 = android::hardware::power::V1\_1::IPower::castFrom(gPowerHalV1\_0);

ALOGI("Loaded power HAL service");

} else {

ALOGI("Couldn't load power HAL service");

gPowerHalExists = false;

}

}

return gPowerHalV1\_0 != nullptr;

}

1.2 onStart函数

@Override

public void onStart() {

publishBinderService(Context.POWER\_SERVICE, new BinderService());//PMS将内部定义的BinderService注册到Service Manager进程，对应的名称为"power"。

publishLocalService(PowerManagerInternal.class, new LocalService());//为PMS注册LocalService

Watchdog.getInstance().addMonitor(this);//看门狗机制

Watchdog.getInstance().addThread(mHandler);

}

**(1) publishBinderService**

publishBinderService定义在PMS的父类SystemService中

//PMS中的BinderService继承自IPowerManager.Stub，实现了IBinder接口，命名PMS的名称为"power"

protected final void publishBinderService(String name, IBinder service) {

this.publishBinderService(name, service, false);

}

//调用ServiceManger的接口，实际上利用Binder通信向Service Manger进程注册服务

protected final void publishBinderService(String name, IBinder service, boolean allowIsolated) {

ServiceManager.addService(name, service, allowIsolated);

}

**(2) publishLocalService**

publishLocalService同样定义在PMS的父类SystemService中：

//参数中的LocalService继承自PowerManagerInternal类

protected final <T> void publishLocalService(Class<T> type, T service) {

LocalServices.addService(type, service);

}

我们来看看LocalServices类的addService函数

public static <T> void addService(Class<T> type, T service) {

synchronized (sLocalServiceObjects) {

if (sLocalServiceObjects.containsKey(type)) {

throw new IllegalStateException("Overriding service registration");

}

sLocalServiceObjects.put(type, service);

}

}

从代码来看，LocalServices的作用与单例模式有些相似，不过更为严格。 单例模式用于保证某个类只能创建出一个对象； LocalServices管理对象时，确保继承某个类或接口的对象只有一个。 目前，还看不出PMS利用LocalServices管理其内部的LocalService的理由，先在此留个悬念。

**(3) watchdog**

//PMS实现了Watchdog.Monitor接口，下面的代码将PMS加入到watchdog的mMonitorChecker中

Watchdog.getInstance().addMonitor(this);

//将PMS的ServiceThread对对应的handler传入watchdog中，watchdog将利用该handler构造一个HandlerChecker

Watchdog.getInstance().addThread(mHandler);

PMS只要能在规定时间内获得mLock锁，watchDog就能确认PMS没有死锁，状态正常

@Override // Watchdog.Monitor implementation

public void monitor() {

// Grab and release lock for watchdog monitor to detect deadlocks.

synchronized (mLock) {

}

}

1.3 systemReady

PMS在systemReady中主要是获得一些成员变量，注册一些广播接收对象、读取一些配置参数等。虽然PMS会与多个服务打交道，使得systemReady内容较为庞杂，但整个过程比较简单。

在systemReady函数的最后，调用了一个比较重要的函数updatePowerStateLocked。当PMS监控到终端发生重大变化时，将利用该函数集中更新所有相关的状态。updatePowerStateLocked涉及的内容较多，等对PMS整体有了初步的了解后，再来分析这个函数。

public void systemReady(IAppOpsService appOps) {

//这里持锁，意味者内部所有函数必须在规定事件内执行完毕，否则watchDog将会检测到错误

synchronized (mLock) {

mSystemReady = true;

mAppOps = appOps;

//注意下面的对象都是从LocalServices取出的，从之前PMS加入LocalServices的流程来看

//下面取出的实际上应该是继承抽象类的实际子类

//因此大概可以理解LocalServices的作用了，以抽象类名为key，保存实际的子类

//以key取出子类对象时，子类退化为抽象类，于是得到了实际对象，但仅能利用父类的接口

//整个设计演变为面向接口的编程

//与doze有关

mDreamManager = getLocalService(DreamManagerInternal.class);

//显示管理相关

mDisplayManagerInternal = getLocalService(DisplayManagerInternal.class);

//Window管理相关

mPolicy = getLocalService(WindowManagerPolicy.class);

//电源管理相关

mBatteryManagerInternal = getLocalService(BatteryManagerInternal.class);

//这里获取PowerManager就是为了方便获取下面三个系统属性

PowerManager pm = (PowerManager) mContext.getSystemService(Context.POWER\_SERVICE);

mScreenBrightnessSettingMinimum = pm.getMinimumScreenBrightnessSetting();

mScreenBrightnessSettingMaximum = pm.getMaximumScreenBrightnessSetting();

mScreenBrightnessSettingDefault = pm.getDefaultScreenBrightnessSetting();

mScreenBrightnessForVrSettingDefault = pm.getDefaultScreenBrightnessForVrSetting();

//Sensor相关

SensorManager sensorManager = new SystemSensorManager(mContext, mHandler.getLooper());

//电量统计有关

mBatteryStats = BatteryStatsService.getService();

//这个notifier运行在system server的main looper上，以不干扰power manager的动画和其他关键功能，好多wake lock的相关方法

mNotifier = new Notifier(Looper.getMainLooper(), mContext, mBatteryStats, mAppOps, createSuspendBlockerLocked("PowerManagerService.Broadcasts"), mPolicy);

//无线充电检测相关

mWirelessChargerDetector = new WirelessChargerDetector(sensorManager, createSuspendBlockerLocked("PowerManagerService.WirelessChargerDetector"), mHandler);

mSettingsObserver = new SettingsObserver(mHandler);

//lightService相关

mLightsManager = getLocalService(LightsManager.class);

mAttentionLight = mLightsManager.getLight(LightsManager.LIGHT\_ID\_ATTENTION);

//Initialize display power management.

//调用DisplayManagerService内部的LocalService的函数

//创建出DisplayBlanker和DisplayPowerController

mDisplayManagerInternal.initPowerManagement(mDisplayPowerCallbacks, mHandler, sensorManager);

//从资源文件中读取大量配置信息

readConfigurationLocked();

//读取数据库字段，保存到本地变量中

//必要时会进行一些实际的操作，例如设置feature到native层的动态库，这个追到底层后没有发现具体实现

updateSettingsLocked();

mDirty |= DIRTY\_BATTERY\_STATE;

//更新全局的电源状态

updatePowerStateLocked();

}

final ContentResolver resolver = mContext.getContentResolver();

mConstants.start(resolver);

mBatterySaverPolicy.start(resolver);

//利用mSettingsObserver监听一堆数据库字段的变化

resolver.registerContentObserver(Settings.Secure.getUriFor(Settings.Secure.SCREENSAVER\_ENABLED), false, mSettingsObserver, UserHandle.USER\_ALL);

resolver.registerContentObserver(Settings.Secure.getUriFor(Settings.Secure.SCREENSAVER\_ACTIVATE\_ON\_SLEEP), false, mSettingsObserver, UserHandle.USER\_ALL);

resolver.registerContentObserver(Settings.Secure.getUriFor(Settings.Secure.SCREENSAVER\_ACTIVATE\_ON\_DOCK), false, mSettingsObserver, UserHandle.USER\_ALL);

resolver.registerContentObserver(Settings.System.getUriFor(Settings.System.SCREEN\_OFF\_TIMEOUT), false, mSettingsObserver, UserHandle.USER\_ALL);

resolver.registerContentObserver(Settings.Secure.getUriFor(Settings.Secure.SLEEP\_TIMEOUT), false, mSettingsObserver, UserHandle.USER\_ALL);

resolver.registerContentObserver(Settings.Global.getUriFor(Settings.Global.STAY\_ON\_WHILE\_PLUGGED\_IN), false, mSettingsObserver, UserHandle.USER\_ALL);

resolver.registerContentObserver(Settings.System.getUriFor(Settings.System.SCREEN\_BRIGHTNESS), false, mSettingsObserver, UserHandle.USER\_ALL);

resolver.registerContentObserver(Settings.System.getUriFor(Settings.System.SCREEN\_BRIGHTNESS\_FOR\_VR), false, mSettingsObserver, UserHandle.USER\_ALL);

resolver.registerContentObserver(Settings.System.getUriFor(Settings.System.SCREEN\_BRIGHTNESS\_MODE), false, mSettingsObserver, UserHandle.USER\_ALL);

resolver.registerContentObserver(Settings.System.getUriFor(Settings.System.SCREEN\_AUTO\_BRIGHTNESS\_ADJ), false, mSettingsObserver, UserHandle.USER\_ALL);

resolver.registerContentObserver(Settings.Global.getUriFor(Settings.Global.LOW\_POWER\_MODE), false, mSettingsObserver, UserHandle.USER\_ALL);

resolver.registerContentObserver(Settings.Global.getUriFor(Settings.Global.LOW\_POWER\_MODE\_TRIGGER\_LEVEL), false, mSettingsObserver, UserHandle.USER\_ALL);

resolver.registerContentObserver(Settings.Global.getUriFor(Settings.Global.THEATER\_MODE\_ON), false, mSettingsObserver, UserHandle.USER\_ALL);

resolver.registerContentObserver(Settings.Secure.getUriFor(Settings.Secure.DOZE\_ALWAYS\_ON), false, mSettingsObserver, UserHandle.USER\_ALL);

resolver.registerContentObserver(Settings.Secure.getUriFor(Settings.Secure.DOUBLE\_TAP\_TO\_WAKE), false, mSettingsObserver, UserHandle.USER\_ALL);

resolver.registerContentObserver(Settings.Global.getUriFor(Settings.Global.DEVICE\_DEMO\_MODE), false, mSettingsObserver, UserHandle.USER\_SYSTEM);

//VR相关

IVrManager vrManager = (IVrManager) getBinderService(Context.VR\_SERVICE);

if (vrManager != null) {

try {

vrManager.registerListener(mVrStateCallbacks);

} catch (RemoteException e) {

Slog.e(TAG, "Failed to register VR mode state listener: " + e);

}

}

// Register for broadcasts from other components of the system.

//定义一堆BroadcastReceiver

IntentFilter filter = new IntentFilter();

filter.addAction(Intent.ACTION\_BATTERY\_CHANGED);

filter.setPriority(IntentFilter.SYSTEM\_HIGH\_PRIORITY);

mContext.registerReceiver(new BatteryReceiver(), filter, null, mHandler);

filter = new IntentFilter();

filter.addAction(Intent.ACTION\_DREAMING\_STARTED);

filter.addAction(Intent.ACTION\_DREAMING\_STOPPED);

mContext.registerReceiver(new DreamReceiver(), filter, null, mHandler);

filter = new IntentFilter();

filter.addAction(Intent.ACTION\_USER\_SWITCHED);

mContext.registerReceiver(new UserSwitchedReceiver(), filter, null, mHandler);

filter = new IntentFilter();

filter.addAction(Intent.ACTION\_DOCK\_EVENT);

mContext.registerReceiver(new DockReceiver(), filter, null, mHandler);

}

2 PMS WakeLock使用及流程

acquire WakeLock将申请信息递交给PMS统一进行处理。

PMS根据WakeLock的level和flag，完成修改一些变量、通知BatteryStatsService等工作后，最终还是依赖于updatePowerStateLocked函数来进行实际的电源状态更新操作。

PMS类中有很多\*\*\*NoUpdateLocked()方法，这些方法都有一些共性，就是仅更新状态，不负责具体的执行。因为PMS中具体的执行逻辑都是updatePowerStateLocked方法中。

2.1创建WakeLock

调用PowerManager的newWakeLock函数，可以创建出WakeLock，用RIL.java具体说明

frameworks/opt/telephony/src/java/com/android/internal/telephony/RIL.java

public RIL(Context context, int preferredNetworkType, int cdmaSubscription, Integer instanceId) {

......

PowerManager pm = (PowerManager)context.getSystemService(Context.POWER\_SERVICE);

//获取WakeLock，第一个参数决定了WakeLock的等级和flag

mWakeLock = pm.newWakeLock(PowerManager.PARTIAL\_WAKE\_LOCK, RILJ\_LOG\_TAG);

//默认WakeLocked会ReferenceCounted，即一次申请对应一次释放

//设为false后，一次释放就可以对应所有的申请

mWakeLock.setReferenceCounted(false);

......

//RIL.java中自己维护了WakeLockCount

mWakeLockCount = 0;

......

}

我们看看PowerManager中的newWakeLock函数定义

public WakeLock newWakeLock(int levelAndFlags, String tag) {

//检查参数有效性，即levelAndFlags必须对应于PowerManager中定义的WakeLock级别和flag，tag不能为空

validateWakeLockParameters(levelAndFlags, tag);

//此WakeLock为PowerManager定义的内部类

return new WakeLock(levelAndFlags, tag, mContext.getOpPackageName());

}

**(1) WakeLock Level & Mask**

public static final int PARTIAL\_WAKE\_LOCK = 0x00000001;

public static final int SCREEN\_DIM\_WAKE\_LOCK = 0x00000006;

public static final int SCREEN\_BRIGHT\_WAKE\_LOCK = 0x0000000a;

public static final int FULL\_WAKE\_LOCK = 0x0000001a;

public static final int PROXIMITY\_SCREEN\_OFF\_WAKE\_LOCK = 0x00000020;

public static final int DOZE\_WAKE\_LOCK = 0x00000040;

public static final int DRAW\_WAKE\_LOCK = 0x00000080;

public static final int WAKE\_LOCK\_LEVEL\_MASK = 0x0000ffff;

**(2) WakeLock Flag**

public static final int ACQUIRE\_CAUSES\_WAKEUP = 0x10000000;

public static final int ON\_AFTER\_RELEASE = 0x20000000;

public static final int UNIMPORTANT\_FOR\_LOGGING = 0x40000000;

public static final int RELEASE\_FLAG\_WAIT\_FOR\_NO\_PROXIMITY = 1 << 0;

public static final int RELEASE\_FLAG\_TIMEOUT = 1 << 16;

WakeLock Level一般与WakeLock Flag组合使用，使用的时候参照一下注释即可

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Level | 保持CPU running | 保持  屏幕亮 | 保持  键盘亮 | 描述 |
| PARTIAL\_WAKE\_LOCK | 是 | 否 | 否 | 用户按下power键，屏幕熄灭后，CPU会保持running，直到该wakelock被释放 |
| SCREEN\_DIM\_WAKE\_LOCK | 是 | Dim | 否 | 用户按下power键，屏幕熄灭后，接着系统隐性释放掉，然后CPU休眠 |
| SCREEN\_BRIGHT\_WAKE\_LOCK | 是 | Bright | 否 | 用户按下power键，屏幕熄灭后，接着系统隐性释放掉，然后CPU休眠 |
| FULL\_WAKE\_LOCK | 是 | Bright | 是 | 用户按下power键，屏幕熄灭后，接着系统隐性释放掉，然后CPU休眠 |
| PROXIMITY\_SCREEN\_OFF\_WAKE\_LOCK | - | - | - | 当系统处于唤醒态时，pensor发现终端离某个物体较近时候，关闭屏幕，远离该物体的时候，点亮屏幕；  虽然无法阻止系统休眠，但是因为pensor一直工作的原因，设备在靠近物体灭屏的时候，也不会休眠 |
| DOZE\_WAKE\_LOCK | - | - | - | 终端处于Dozing state，使得CPU挂起，屏幕处于低电模式，难道是cmd、oled屏幕 |
| DRAW\_WAKE\_LOCK | - | - | - | 终端处于Dozing state，windows manager给予应用足够的时间绘图 |
| Flag |  |  |  | 描述 |
| ACQUIRE\_CAUSES\_WAKEUP | - | - | - | 通常acquire一个wakelock不会点亮屏幕，加上这个标志后，acquire wakelock的同时能点亮屏幕 |
| ON\_AFTER\_RELEASE | - | - | - | 设置了这个标志，当wakelock释放时用户activity计时器会被重置，导致亮屏多持续一段时间 |
| UNIMPORTANT\_FOR\_LOGGING | - | - | - | 系统级别会用到这个，我看AlarmManagerService用到了这个，这个问题先记着 |
| RELEASE\_FLAG\_WAIT\_FOR\_NO\_PROXIMITY | - | - | - | 设置了这个标志，psensor认为终端跟物体距离较远时，释放wakelock |
| RELEASE\_FLAG\_TIMEOUT | - | - | - | 设置了这个标志，超时后释放wakelock |

**(3) WakeLock的构造函数**

WakeLock的构造函数中需要注意的地方是，创建了一个Binder对象，该对象应该是RIL.java所在的Phone进程中。

WakeLock(int flags, String tag, String packageName) {

//level and flag

mFlags = flags;

//创建类对应的打印Tag

mTag = tag;

//创建类的类名

mPackageName = packageName;

//创建一个Binder对象

//PMS将作为该Binder的客户端监听对应进程是否死亡

mToken = new Binder();

mTraceName = "WakeLock (" + mTag + ")";

}

2.2 Acquire WakeLock

进程创建了WakeLock后，需要将WakeLock发送到PMS中，让PMS明白该进程的需求。这种将WakeLock通知到PMS的过程，就被称为acquire WakeLock。

我们还是拿RIL.java来说明

acquireWakeLock(rr, FOR\_WAKELOCK);

RIL跟modem进行通信时，会acquire WakeLock：

private void acquireWakeLock(RILRequest rr, int wakeLockType) {

synchronized (rr) {

............

switch(wakeLockType) {

case FOR\_WAKELOCK:

synchronized (mWakeLock) {

//调用acquire函数

mWakeLock.acquire();

mWakeLockCount++;

mWlSequenceNum++;

............

}

break;

............

}

rr.mWakeLockType = wakeLockType;

}

}

我们跟进一下PowerManager中WakeLock的acquire函数：

public void acquire() {

synchronized (mToken) {

acquireLocked();

}

}

private void acquireLocked() {

//前面已经提过，RIL.java中已经将mRefCounted置为false

//如果不将mRefCounted置为false，意味着acquire和release必须一一对应

//那么每个WakeLock只能acquire一次

if (!mRefCounted || mInternalCount == 1) {

............

try {

mService.acquireWakeLock(mToken, mFlags, mTag, mPackageName, mWorkSource, mHistoryTag);//通过Binder通信传入PMS中，继续看下面分析

} catch (RemoteException e) {

throw e.rethrowFromSystemServer();

}

mHeld = true;

}

}

PMS的acquireWakeLock函数：

public void acquireWakeLock(IBinder lock, int flags, String tag, String packageName,

WorkSource ws, String historyTag) {

//参数和权限检查

............

final int uid = Binder.getCallingUid();

final int pid = Binder.getCallingPid();

final long ident = Binder.clearCallingIdentity();

try {

acquireWakeLockInternal(lock, flags, tag, packageName, ws, historyTag, uid, pid);

} finally {

Binder.restoreCallingIdentity(ident);

}

}

private void acquireWakeLockInternal(IBinder lock, int flags, String tag, String packageName,

WorkSource ws, String historyTag, int uid, int pid) {

synchronized (mLock) {

............

//PMS中也定义了WakeLock内部类，protected final class WakeLock implements Ibinder.DeathRecipient

//findWakeLockIndexLocked查找ArrayList，判断参数对应的WakeLock，是否在之前被申请过

WakeLock wakeLock;

int index = findWakeLockIndexLocked(lock);

boolean notifyAcquire;

if (index >= 0) {

//如果index大于0，说明此时Acquire的是一个旧的WakeLock

//例如RIL会多次调用send函数，于是除第一次外，都会进入这个分支

wakeLock = mWakeLocks.get(index);

if (!wakeLock.hasSameProperties(flags, tag, ws, uid, pid)) {//这是判断WakeLock对应的成员变量是否发生改变

// Update existing wake lock. This shouldn't happen but is harmless. (这不应该发生，但无害)

notifyWakeLockChangingLocked(wakeLock, flags, tag, packageName,

uid, pid, ws, historyTag);

//若wakelock属性发生了变化，更新该属性

wakeLock.updateProperties(flags, tag, packageName, ws, historyTag, uid, pid);

}

notifyAcquire = false;

} else {

............

//创建一个新的WakeLock，例如RIL第一次调用send就会进入该分支

wakeLock = new WakeLock(lock, flags, tag, packageName, ws, historyTag, uid, pid, state);

try {

//2.2.1监控申请WakeLock的进程是否死亡

lock.linkToDeath(wakeLock, 0);

} catch (RemoteException ex) {

throw new IllegalArgumentException("Wake lock is already dead.");

}

//添加到wakelock列表

mWakeLocks.add(wakeLock);

//2.2.2特殊处理PARTIAL\_WAKE\_LOCK

//实际上，根据Doze模式的白名单更新wakelock的disabled变量

setWakeLockDisabledStateLocked(wakeLock);

qcNsrmPowExt.checkPmsBlockedWakelocks(uid, pid, flags, tag, wakeLock);

notifyAcquire = true;

}

//2.2.3处理WakeLock对应的Flag

//实际上判断WakeLock是否有ACQUIRE\_CAUSES\_WAKEUP，在必要时唤醒屏幕

applyWakeLockFlagsOnAcquireLocked(wakeLock, uid);

mDirty |= DIRTY\_WAKE\_LOCKS;

//更新电源状态，以后单独分析

updatePowerStateLocked();

if (notifyAcquire) {

//这需要最后一次完成，所以我们确信我们已经获得了内核唤醒锁。

//否则，我们会有一场比赛，在我们开始在电池统计中进行计时的时候，系统可能会进入睡眠状态，而我们实际上是在告诉内核保持清醒。

//通知wakeLock发生变化

//电量统计服务做相关统计

notifyWakeLockAcquiredLocked(wakeLock);

}

}

}

如上代码中标注的注释，acquireWakeLockInternal中有几处比较重要的地方，我们一起来分析一下

**2.2.1监听客户端进程死亡**

上面的代码中，第一次创建WakeLock后，调用了linkToDeath， 如果binder死亡，将通知PMS。

.........

lock.linkToDeath(wakeLock, 0);

.........

我们看看PMS中WakeLock与此相关的定义：

protected final class WakeLock implements IBinder.DeathRecipient {

.........

@Override

public void binderDied() {

//发现客户端进程死亡后，调用PMS的handleWakeLockDeath进行处理，传入的参数为WakeLock自己

PowerManagerService.this.handleWakeLockDeath(this);

}

.........

}

private void handleWakeLockDeath(WakeLock wakeLock) {

synchronized (mLock) {

.........

int index = mWakeLocks.indexOf(wakeLock);

if (index < 0) {

return;

}

removeWakeLockLocked(wakeLock, index);

}

}

private void removeWakeLockLocked(WakeLock wakeLock, int index) {

mWakeLocks.remove(index);

UidState state = wakeLock.mUidState;

state.mNumWakeLocks--;

if (state.mNumWakeLocks <= 0 &&

state.mProcState == ActivityManager.PROCESS\_STATE\_NONEXISTENT) {

mUidState.remove(state.mUid);

}

//通知到BatteryStatsService

notifyWakeLockReleasedLocked(wakeLock);

//处理WakeLock对应的flag，与后文applyWakeLockFlagsOnAcquireLocked一起分析

//实际上是判断是否需要立即息屏

applyWakeLockFlagsOnReleaseLocked(wakeLock);

mDirty |= DIRTY\_WAKE\_LOCKS;

//锁移除后，还是利用updatePowerStateLocked更新电源状态

updatePowerStateLocked();

}

**2.2.2特殊处理PARTIAL\_WAKE\_LOCK**

PMS处理第一次创建的WakeLock时，还会调用setWakeLockDisabledStateLocked函数进行处理：

在Android Doze模式下，当终端处于device Idle Mode时，对于一个非系统应用而言，如果该应用不在系统定义的白名单中， 并且该应用所在进程的类型优先级不高，说明进程对事件处理的时效性要求不高， 那么即使该应用申请了PARTIAL\_WAKE\_LOCK，也不能阻止系统进入休眠状态。

有些设备商，为了优化系统的功耗，就修改了这个地方。

例如，有些系统应用其实也很耗电，因此可以去掉该函数中对非系统应用的限制判断条件，这样对系统应用也可以进行管控。

private boolean setWakeLockDisabledStateLocked(WakeLock wakeLock) {

//仅会特殊处理PARTIAL\_WAKE\_LOCK，毕竟PARTIAL\_WAKE\_LOCK要求按Power键后CPU依然可以工作

if ((wakeLock.mFlags & PowerManager.WAKE\_LOCK\_LEVEL\_MASK)

== PowerManager.PARTIAL\_WAKE\_LOCK) {

boolean disabled = false;

final int appid = UserHandle.getAppId(wakeLock.mOwnerUid);

if (appid >= Process.FIRST\_APPLICATION\_UID) {//如果是非系统应用不允许持有wakelock

//Cached inactive processes are never allowed to hold wake locks. -- 缓存的非活动进程？？？难道是cached memory中的进程？？？？？

if (mConstants.NO\_CACHED\_WAKE\_LOCKS) {

disabled = !wakeLock.mUidState.mActive &&

wakeLock.mUidState.mProcState != ActivityManager.PROCESS\_STATE\_NONEXISTENT && //!=进程不存在

wakeLock.mUidState.mProcState > ActivityManager.PROCESS\_STATE\_RECEIVER; //>后台进程，且正在运行receiver

}

//设备处于Doze定义的device idle模式时

if (mDeviceIdleMode) {

//在idle模式下，我们会忽略没有在白名单的应用中所申请partial wake lock

final UidState state = wakeLock.mUidState;

//白名单search

if (Arrays.binarySearch(mDeviceIdleWhitelist, appid) < 0 &&

Arrays.binarySearch(mDeviceIdleTempWhitelist, appid) < 0 &&

state.mProcState != ActivityManager.PROCESS\_STATE\_NONEXISTENT &&

state.mProcState > ActivityManager.PROCESS\_STATE\_FOREGROUND\_SERVICE) {//大于前台service，数字越大优先级越低

disabled = true;//不在白名单中，优先级又不高，需要忽略它的wakelock

}

}

}

if (wakeLock.mDisabled != disabled) {

wakeLock.mDisabled = disabled;

return true;

}

}

return false;

}

2.2.3处理WakeLock对应的Flag

当acquire WakeLock时，将调用applyWakeLockFlagsOnAcquireLocked处理WakeLock对应的flag；

当由于进程死亡，释放WakeLock时，会调用applyWakeLockFlagsOnReleaseLocked处理WakeLock对应的flag。

**(1) 申请wakelock-applyWakeLockFlagsOnAcquireLocked**

是否需要点亮屏幕

private void applyWakeLockFlagsOnAcquireLocked(WakeLock wakeLock, int uid) {

//仅处理ACQUIRE\_CAUSES\_WAKEUP flag，同时要求WakeLock的level是与screen有关的，

//即FULL\_WAKE\_LOCK、SCREEN\_BRIGHT\_WAKE\_LOCK和SCREEN\_DIM\_WAKE\_LOCK

if ((wakeLock.mFlags & PowerManager.ACQUIRE\_CAUSES\_WAKEUP) != 0

&& isScreenLock(wakeLock)) {

String opPackageName;//包名

int opUid;//uid

if (wakeLock.mWorkSource != null && wakeLock.mWorkSource.getName(0) != null) {

opPackageName = wakeLock.mWorkSource.getName(0);

opUid = wakeLock.mWorkSource.get(0);

} else {

opPackageName = wakeLock.mPackageName;

opUid = wakeLock.mWorkSource != null ? wakeLock.mWorkSource.get(0) : wakeLock.mOwnerUid;

}

wakeUpNoUpdateLocked(SystemClock.uptimeMillis(), wakeLock.mTag, opUid, opPackageName, opUid);

}

}

private boolean wakeUpNoUpdateLocked(long eventTime, String reason, int reasonUid,

String opPackageName, int opUid) {

//不满足以下条件，没有唤醒屏幕的必要

if (eventTime < mLastSleepTime || mWakefulness == WAKEFULNESS\_AWAKE

|| !mBootCompleted || !mSystemReady) {

return false;

}

try {

switch (mWakefulness) {

case WAKEFULNESS\_ASLEEP://waking up from 休眠

Slog.i(TAG, "Waking up from sleep (uid=" + reasonUid + " reason=" + reason + ")..."); break;

case WAKEFULNESS\_DREAMING://waking up from 屏保

Slog.i(TAG, "Waking up from dream (uid=" + reasonUid + " reason=" + reason + ")..."); break;

case WAKEFULNESS\_DOZING://waking up from doze

Slog.i(TAG, "Waking up from dozing (uid=" + reasonUid + " reason=" + reason + ")..."); break;

}

mLastWakeTime = eventTime;//记录本次eventTime为最近一次唤醒的时间

//修改PMS的一些成员变量，并进行通知

//其中主要的是将mDirty变量的DIRTY\_WAKEFULNESS位置为了1

//PMS根据mDirty的位信息管理电源状态，同时唤醒屏幕

//Notifier调用onWakefulnessChangeStarted发送亮屏广播，还涉及AMS和WindowsManager

setWakefulnessLocked(WAKEFULNESS\_AWAKE, 0);//亮屏前的初始化工作，然后回调到PowerManager的wakeUp函数，整个过程还是比较复杂的，以后再分析

//通知给电源统计服务

mNotifier.onWakeUp(reason, reasonUid, opPackageName, opUid);

//调用userActivityNoUpdateLocked函数

userActivityNoUpdateLocked(eventTime, PowerManager.USER\_ACTIVITY\_EVENT\_OTHER, 0, reasonUid);//更新最后一次用户事件的时间

}......

return true;

}

**(2) 释放wakelock-applyWakeLockFlagsOnReleaseLocked**

private void applyWakeLockFlagsOnReleaseLocked(WakeLock wakeLock) {

if ((wakeLock.mFlags & PowerManager.ON\_AFTER\_RELEASE) != 0

&& isScreenLock(wakeLock)) {

userActivityNoUpdateLocked(SystemClock.uptimeMillis(),

PowerManager.USER\_ACTIVITY\_EVENT\_OTHER,

PowerManager.USER\_ACTIVITY\_FLAG\_NO\_CHANGE\_LIGHTS,

wakeLock.mOwnerUid);

}

}

applyWakeLockFlagsOnAcquireLocked和applyWakeLockFlagsOnReleaseLocked最后均会调用userActivityNoUpdateLocked函数，只是参数不同。

**(3) 申请/释放wakelock都调用userActivityNoUpdateLocked**

private boolean userActivityNoUpdateLocked(long eventTime, int event, int flags, int uid) {

......

//过时的事件不需要处理

if (eventTime < mLastSleepTime || eventTime < mLastWakeTime || !mBootCompleted || !mSystemReady) {

return false;

}

Trace.traceBegin(Trace.TRACE\_TAG\_POWER, "userActivity");

try {

if (eventTime > mLastInteractivePowerHintTime) {

powerHintInternal(PowerHint.INTERACTION, 0);//调用native加载的动态库的powerHint函数，底层是高通实现，调频相关，CPU4\_MIN\_FREQ

mLastInteractivePowerHintTime = eventTime;

}

mNotifier.onUserActivity(event, uid);//调用BatteryStatsService的noteUserActivity函数，做event对应的type计数

if (mUserInactiveOverrideFromWindowManager) {

mUserInactiveOverrideFromWindowManager = false;

mOverriddenTimeout = -1;

}

if (mWakefulness == WAKEFULNESS\_ASLEEP

|| mWakefulness == WAKEFULNESS\_DOZING

|| (flags & PowerManager.USER\_ACTIVITY\_FLAG\_INDIRECT) != 0) {

return false;

}

//根据参数信息修改mDirty的一些变量

if ((flags & PowerManager.USER\_ACTIVITY\_FLAG\_NO\_CHANGE\_LIGHTS) != 0) {

if (eventTime > mLastUserActivityTimeNoChangeLights

&& eventTime > mLastUserActivityTime) {

mLastUserActivityTimeNoChangeLights = eventTime;

mDirty |= DIRTY\_USER\_ACTIVITY;

if (event == PowerManager.USER\_ACTIVITY\_EVENT\_BUTTON) {

mDirty |= DIRTY\_QUIESCENT;

}

return true;

}

} else {

if (eventTime > mLastUserActivityTime) {

mLastUserActivityTime = eventTime;

mDirty |= DIRTY\_USER\_ACTIVITY;

if (event == PowerManager.USER\_ACTIVITY\_EVENT\_BUTTON) {

mDirty |= DIRTY\_QUIESCENT;

}

return true;

}

}

} finally {

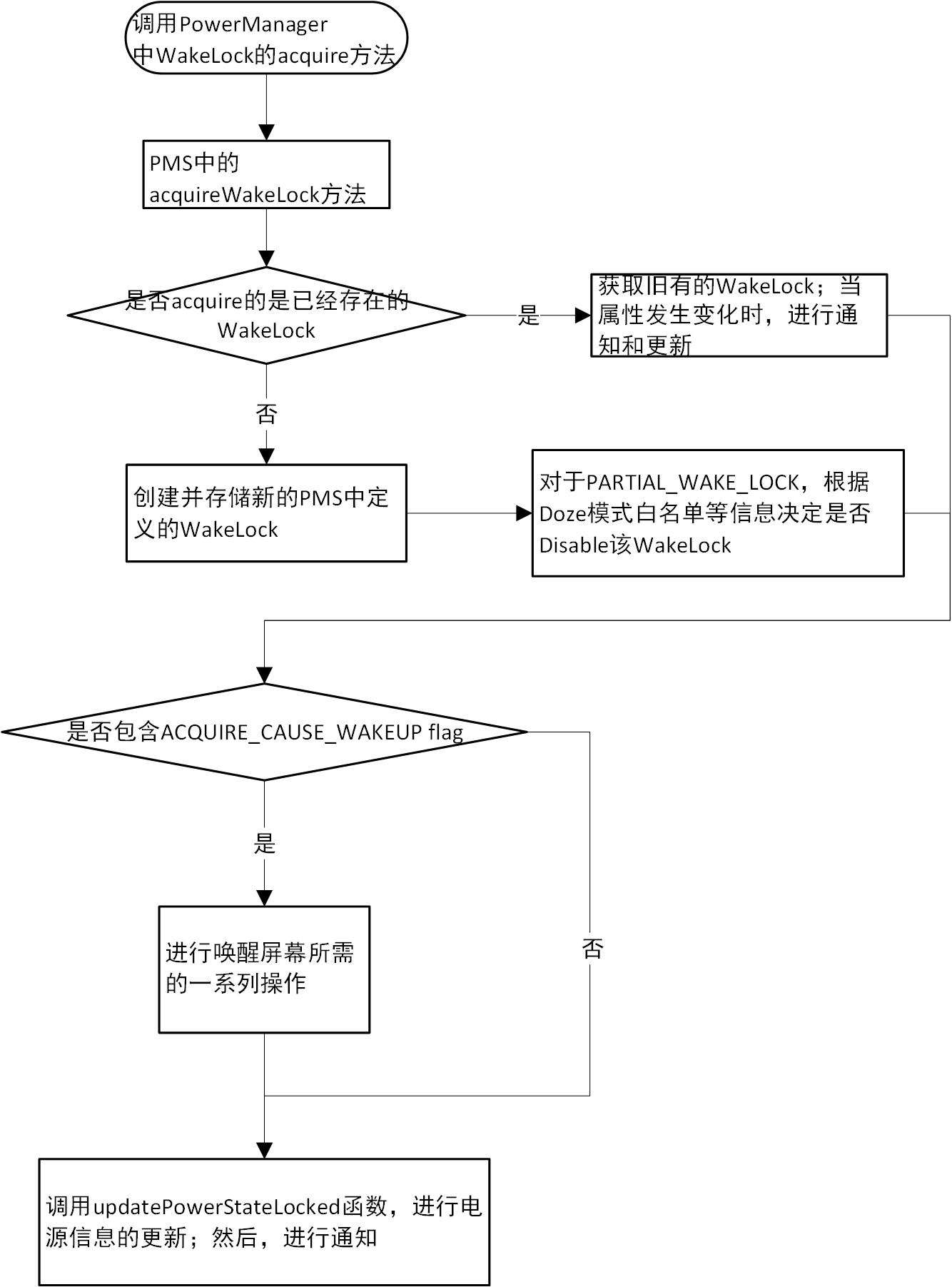
Trace.traceEnd(Trace.TRACE\_TAG\_POWER);

}

return false;

}

2.2.4 acquire WakeLock流程图

2.2 Release WakeLock

2.2.1 release WakeLock code flow

当进程完成工作后，需要释放之前申请的WakeLock。我们同样以RIL.java中的操作为例

protected RILRequest processResponse(RadioResponseInfo responseInfo) {

…...

decrementWakeLock(rr);

…...

}

private void decrementWakeLock(RILRequest rr) {

synchronized (rr) {

switch(rr.mWakeLockType) {

case FOR\_WAKELOCK:

synchronized (mWakeLock) {

…...

//前面已经提到过，RIL.java多个请求复用同一个WakeLock

//并且利用mWakeLockCount记录复用的次数

//这么设计的目的是：RIL发送请求的数量非常多，复用WakeLock可以避免多次构造释放

//同时减少与PMS之间Binder通信的次数

if (mWakeLockCount > 1) {

mWakeLockCount--;

} else {

mWakeLockCount = 0;

//所有请求均得到了处理，调用PowerManager中WakeLock的release函数

mWakeLock.release();

}

}

break;

…...

}

…...

}

}

现在我们跟进PowerManager中WakeLock定义的release函数

public void release(int flags) {

…...

mService.releaseWakeLock(mToken, flags);//PMS的release wakelock

…...

}

PMS中释放WakeLock函数

public void releaseWakeLock(IBinder lock, int flags) {

…//参数和权限检查

final long ident = Binder.clearCallingIdentity();

try {

releaseWakeLockInternal(lock, flags);

} finally {

Binder.restoreCallingIdentity(ident);

}

}

private void releaseWakeLockInternal(IBinder lock, int flags) {

synchronized (mLock) {

//根据Binder代理，从存储的ArrayList中找到对应WakeLock的序号

int index = findWakeLockIndexLocked(lock);

…...

WakeLock wakeLock = mWakeLocks.get(index);

…...

//RELEASE\_FLAG\_WAIT\_FOR\_NO\_PROXIMITY，表示当sensor判断终端离物体较远时，

//才真正释放PROXIMITY\_SCREEN\_OFF\_WAKE\_LOCK等级的WakeLock

if ((flags & PowerManager.RELEASE\_FLAG\_WAIT\_FOR\_NO\_PROXIMITY) != 0) {

mRequestWaitForNegativeProximity = true;

}

//PMS不再关注客户端进程是否死亡

wakeLock.mLock.unlinkToDeath(wakeLock, 0);

removeWakeLockLocked(wakeLock, index);

}

}

private void removeWakeLockLocked(WakeLock wakeLock, int index) {

mWakeLocks.remove(index);

UidState state = wakeLock.mUidState;

state.mNumWakeLocks--;

if (state.mNumWakeLocks <= 0 &&

state.mProcState == ActivityManager.PROCESS\_STATE\_NONEXISTENT) {

mUidState.remove(state.mUid);

}

//通知BatteryStatsService

notifyWakeLockReleasedLocked(wakeLock);

//上文分析过，做一些记录，修改mDirty等

applyWakeLockFlagsOnReleaseLocked(wakeLock);

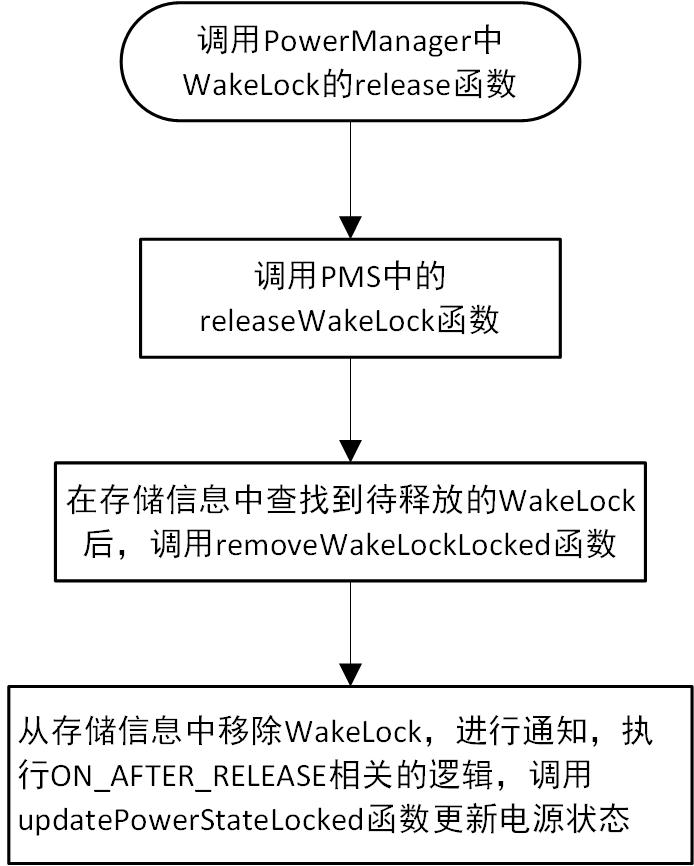
mDirty |= DIRTY\_WAKE\_LOCKS;

//依然靠updatePowerStateLocked函数更新终端的电源状态

updatePowerStateLocked();

}

2.2.2 Release WakeLock流程图

2.3 WakeLock总结

(1) 创建

PowerManager pm = (PowerManager)context.getSystemService(Context.POWER\_SERVICE);

mWakeLock = pm.newWakeLock(PowerManager.PARTIAL\_WAKE\_LOCK, RILJ\_LOG\_TAG);

(2) acquire

mWakeLock.acquire();

(3) release

mWakeLock.release();

当申请发送到PMS后，PMS将针对WakeLock的level和flag信息进行一些处理。

无论是acquire还是release WakeLock，PMS最终将利用updatePowerStateLocked函数对终端的电源状态进行调整。

我们将单独分析一下PMS核心的updatePowerStateLocked函数。

3 PMS核心函数updatePowerStateLocked()

当Android中的进程要使用电量时，需要向PMS申请WakeLock；当进程完成工作后，需要释放对应的WakeLock。PMS收到申请和释放WakeLock的请求后，均需要调用updatePowerStateLocked来更新电源的状态，该函数是PMS的核心方法。

**protected void** updatePowerStateLocked() {

**if** (!mSystemReady || mDirty == 0) {//mDrity没有变化，不更新

**return**;

}

**if** (!**Thread**.holdsLock(mLock)) {//当前进程如果没有持有mLock，What a Terrible Failure，报告一个永远不可能发生的错误！

Slog.wtf(TAG, "Power manager lock was not held when calling updatePowerStateLocked");

}

Trace.traceBegin(Trace.TRACE\_TAG\_POWER, "updatePowerState");

**try** {

//Basic state updates.

//1、更新基本状态

updateIsPoweredLocked(mDirty);//充电、低电等处理

updateStayOnLocked(mDirty);//更新变量mStayOn的值， 如果mStayOn如果为true，则屏幕保持长亮的状态

updateScreenBrightnessBoostLocked(mDirty);//更新终端可处于最大屏幕亮度的时间

//Update wakefulness.

//Loop because the wake lock and user activity computations are influenced by changes in wakefulness.

//2、更新wakelock和根据用户活动确定屏幕状态

**final long** now = **SystemClock**.uptimeMillis();

**int** dirtyPhase2 = 0;

**for** (;;) {

**int** dirtyPhase1 = mDirty;//dp1记录mDrity

dirtyPhase2 |= dirtyPhase1;//dp2再次记录mDirity

mDirty = 0;//清0

updateWakeLockSummaryLocked(dirtyPhase1);//更新wakelock

updateUserActivitySummaryLocked(now, dirtyPhase1);//根据用户活动确定屏幕状态

mExperienceController.updateUserActivity(now, dirtyPhase1, mWakefulness,

mLastWakeTime, mLastUserActivityTime, mLastUserActivityTimeNoChangeLights);

**if** (!updateWakefulnessLocked(dirtyPhase1)) {//第二阶段的电源状态更新是否结束

**break**;

}

}

//Update display power state.

//3、更新display power state

**boolean** displayBecameReady = updateDisplayPowerStateLocked(dirtyPhase2);

//Update dream state (depends on display ready signal).

//4、更新dream state

updateDreamLocked(dirtyPhase2, displayBecameReady);

//Send notifications, if needed.

finishWakefulnessChangeIfNeededLocked();

//Update suspend blocker.

//Because we might release the last suspend blocker here, we need to make sure

//we finished everything else first!

//5、更新suspend blocker

updateSuspendBlockerLocked();

} **finally** {

Trace.traceEnd(Trace.TRACE\_TAG\_POWER);

}

}

根据Android源码中的注释，可以看出updatePowerStateLocked的工作主要分为几个步骤，接下来我们一个一个步骤的来进行分析。

3.1 mDirty

在PMS中有个很重要的变量mDirty，该变量按位存储PMS中的各种变化状态。

例如，之前介绍PMS的acquire WakeLock流程时，就进行了以下操作：

mDirty |= DIRTY\_WAKE\_LOCKS；

每当PMS检测到一些重要事件发生时，就会更新mDirty的相应的位。从updatePowerStateLocked的代码可以看出，它将根据mDirty的信息，来更新手机电源状态。

// Dirty bit: mWakeLocks changed

**protected static final int** DIRTY\_WAKE\_LOCKS = 1 << 0;

// Dirty bit: mWakefulness changed

**private static final int** DIRTY\_WAKEFULNESS = 1 << 1;

// Dirty bit: user activity was poked or may have timed out

**private static final int** DIRTY\_USER\_ACTIVITY = 1 << 2;

// Dirty bit: actual display power state was updated asynchronously

**private static final int** DIRTY\_ACTUAL\_DISPLAY\_POWER\_STATE\_UPDATED = 1 << 3;

// Dirty bit: mBootCompleted changed

**private static final int** DIRTY\_BOOT\_COMPLETED = 1 << 4;

// Dirty bit: settings changed

**private static final int** DIRTY\_SETTINGS = 1 << 5;

// Dirty bit: mIsPowered changed

**private static final int** DIRTY\_IS\_POWERED = 1 << 6;

// Dirty bit: mStayOn changed

**private static final int** DIRTY\_STAY\_ON = 1 << 7;

// Dirty bit: battery state changed

**private static final int** DIRTY\_BATTERY\_STATE = 1 << 8;

// Dirty bit: proximity state changed

**private static final int** DIRTY\_PROXIMITY\_POSITIVE = 1 << 9;

// Dirty bit: dock state changed

**private static final int** DIRTY\_DOCK\_STATE = 1 << 10;

// Dirty bit: brightness boost changed

**private static final int** DIRTY\_SCREEN\_BRIGHTNESS\_BOOST = 1 << 11;

// Dirty bit: sQuiescent changed

**private static final int** DIRTY\_QUIESCENT = 1 << 12;

// Dirty bit: VR Mode enabled changed

**private static final int** DIRTY\_VR\_MODE\_CHANGED = 1 << 13;

// Dirty bit: auto-brightness changed

**private static final int** DIRTY\_AUTO\_BRIGHTNESS = 1 << 14;

...

// A bitfield that indicates what parts of the power state have

// changed and need to be recalculated.

**protected int** mDirty;

**3.2 更新基本状态**

**3.2.1 updateIsPoweredLocked**

updateIsPoweredLocked主要用于：

(1)更新PMS中的一些变量，包括记录终端是否在充电、充电的类型、电池的电量及电池电量是否处于低电状态；

(2)当电源的充电状态，或者充电类型发生变化，判断出现插拔充电器等操作时，是否需要点亮或熄灭屏幕；

(3)当电源充电状态发生变化，或者终端是否处于低电量的标志发生变化的时候，终端调用updateLowPowerModeLocked()更新低电模式相关的操作。

private void updateIsPoweredLocked(int dirty) {//充电状态、省电模式的判断和处理

//DIRTY\_BATTERY\_STATE位置1时，表示终端的电源状态发生了改变

if ((dirty & DIRTY\_BATTERY\_STATE) != 0) {

//过去状态记录

final boolean wasPowered = mIsPowered;//充电判断old

final int oldPlugType = mPlugType;//充电类型old

final boolean oldLevelLow = mBatteryLevelLow;//低电量判断old

//从BatteryService获取现在的记录

mIsPowered = mBatteryManagerInternal.isPowered(BatteryManager.BATTERY\_PLUGGED\_ANY);//得到终端现在是否在充电

mPlugType = mBatteryManagerInternal.getPlugType();//得到充电的类型

mBatteryLevel = mBatteryManagerInternal.getBatteryLevel();//得到当前的电量

mBatteryLevelLow = mBatteryManagerInternal.getBatteryLevelLow();//得到是否为低电量

if (DEBUG\_SPEW) {

//打印变量的log

}

if (wasPowered != mIsPowered || oldPlugType != mPlugType) {//是否充电的状态发生改变，或者充电类型发生改变

mDirty |= DIRTY\_IS\_POWERED;

//无线充电相关，暂时不用管

final boolean dockedOnWirelessCharger = mWirelessChargerDetector.update(mIsPowered, mPlugType, mBatteryLevel);

final long now = SystemClock.uptimeMillis();

//判断插拔充电器或者USB是否需要唤醒屏幕

if (shouldWakeUpWhenPluggedOrUnpluggedLocked(wasPowered, oldPlugType, dockedOnWirelessCharger)) {

//做好唤醒终端屏幕前的准备工作，2.2.3节中分析了wakeUpNoUpdateLocked ，传递的参数：

eventTime = now

reason = "android.server.power:POWER"

reasonUid = Process.SYSTEM\_UID,

opPackageName = mContext.getOpPackageName()

opUid = Process.SYSTEM\_UID

wakeUpNoUpdateLocked(now, "android.server.power:POWER", Process.SYSTEM\_UID, mContext.getOpPackageName(), Process.SYSTEM\_UID);

}

//触发一次用户活动，修改PMS中记录用户活动事件的时间，同时通知BatteryStatsService等

userActivityNoUpdateLocked(now, PowerManager.USER\_ACTIVITY\_EVENT\_OTHER, 0, Process.SYSTEM\_UID);//更新userActivity最新时间点和mDirty

if (dockedOnWirelessCharger) {//无线充电相关消息通知，暂时可以不管

mNotifier.onWirelessChargingStarted();

}

}

if (wasPowered != mIsPowered || oldLevelLow != mBatteryLevelLow) {//比如不充电->充电，那么判断条件成立

//结束低电的状态

if (oldLevelLow != mBatteryLevelLow && !mBatteryLevelLow) {//如果是低电量，true!=0条件成立；如果不是低电量，false==0条件失败

if (DEBUG\_SPEW) {

Slog.d(TAG, "updateIsPoweredLocked: resetting low power snooze");

}

//当电量低并且允许自动设置省电模式时，如果此时省电模式是开启的，就要设置mAutoLowPowerModeSnoozing的值为false；如果省电模式是关闭的，就要设置mAutoLowPowerModeSnoozing的值为true。就是要求mAutoLowPowerModeSnoozing变量与省电模式的逻辑保持一致

mAutoLowPowerModeSnoozing = false;

}

updateLowPowerModeLocked();//更新省电模式相关的操作

}

}

}

updateLowPowerModeLocked函数：

(1)首先判断手机是否在充电，如果手机在充电，退出LowPowerMode模式，同时更新数据库；

(2)当手机的低电量模式发生了变化，就发送广播进行通知，并回调关于监听该模式变化的观察者的接口。

例如：UI对应的APK收到低电量省电模式的广播，就会弹出低电量省电模式的提醒界面。

可以看出这一部分除了更新PMS中的一些变量外，关注的重点还是集中在：

(1)充电状态是否改变；

(2)充电状态的改变，将引出对充电器插拔是否需要亮屏的考虑；

(3)同样，充电状态的改变，将引出对终端的低电模式是否发生改变的考虑。

从这个角度来看，updateIsPoweredLocked函数的命名是实至名归的。

private void updateLowPowerModeLocked() {

if ((mIsPowered || !mBatteryLevelLow && !mBootCompleted) && mLowPowerModeSetting) {//如果在充电或者 非低电量且 mLowPowerModeSetting=true

if (DEBUG\_SPEW) {//log...}

// Turn setting off if powered

//更新数据库，关闭低电模式

Settings.Global.putInt(mContext.getContentResolver(), Settings.Global.LOW\_POWER\_MODE, 0);

mLowPowerModeSetting = false;//退出省电模式，且 mLowPowerModeSetting置为false

}

//是否使能自动进入省电模式，需要满足如下条件 ：

//mIsPowered=false，mAutoLowPowerModeConfigured=true， mAutoLowPowerModeSnoozing=false， mBatteryLevelLow=true

// 未充电 && 进行了自动省电的配置 && 没有设置低电“打盹” && 电池电量低

final boolean autoLowPowerModeEnabled = !mIsPowered && mAutoLowPowerModeConfigured && !mAutoLowPowerModeSnoozing && mBatteryLevelLow;

//使能省电模式，需要满足如下条件 mLowPowerModeSetting=false，autoLowPowerModeEnabled=true

final boolean lowPowerModeEnabled = mLowPowerModeSetting || autoLowPowerModeEnabled;

if (mLowPowerModeEnabled != lowPowerModeEnabled) {//状态发生变化

mLowPowerModeEnabled = lowPowerModeEnabled;

powerHintInternal(PowerHint.LOW\_POWER, lowPowerModeEnabled ? 1 : 0);//这个底层没有查到case POWER\_HINT\_LOW\_POWER的实现

postAfterBootCompleted(new Runnable() {//开机完成后才能执行的Runnable对象

@Override

public void run() {

//发送低电模式CHANGING的广播

Intent intent = new Intent(PowerManager.ACTION\_POWER\_SAVE\_MODE\_CHANGING)

.putExtra(PowerManager.EXTRA\_POWER\_SAVE\_MODE, mLowPowerModeEnabled)

.addFlags(Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY);

mContext.sendBroadcast(intent);//广播

//PMS提供了registerLowPowerModeObserver的接口

//其它进程可以调用该接口，注册观察者

ArrayList<PowerManagerInternal.LowPowerModeListener> listeners;

synchronized (mLock) {

listeners = new ArrayList<PowerManagerInternal.LowPowerModeListener>(mLowPowerModeListeners);

}

for (int i = 0; i < listeners.size(); i++) {

final PowerManagerInternal.LowPowerModeListener listener = listeners.get(i);

final PowerSaveState result = mBatterySaverPolicy.getBatterySaverPolicy(listener.getServiceType(), lowPowerModeEnabled);

listener.onLowPowerModeChanged(result);//调用回调接口的onLowPowerModeChanged函数，通知其它进程低电模式发生改变

}

//再次发送CHANGED广播

intent = new Intent(PowerManager.ACTION\_POWER\_SAVE\_MODE\_CHANGED);

intent.addFlags(Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY);

mContext.sendBroadcastAsUser(intent, UserHandle.ALL);

// Send internal version that requires signature permission.

intent = new Intent(PowerManager.ACTION\_POWER\_SAVE\_MODE\_CHANGED\_INTERNAL);

intent.addFlags(Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY);

mContext.sendBroadcastAsUser(intent, UserHandle.ALL, Manifest.permission.DEVICE\_POWER);

}

});

}

}

**3.2.2 updateStayOnLocked**

这一部分的代码功能比较单一，主要用于更新变量mStayOn的值。 如果mStayOn如果为true，则屏幕保持长亮的状态。

private void updateStayOnLocked(int dirty) {//主要用于更新变量mStayOn的值。 如果mStayOn如果为true，则屏幕保持长亮的状态

if ((dirty & (DIRTY\_BATTERY\_STATE | DIRTY\_SETTINGS)) != 0) {//电池状态或者电源设置发生变化

final boolean wasStayOn = mStayOn;

//设置了充电器插入时亮屏(分为AC充电亮屏、USB充电亮屏或无线充电亮屏)

if (mStayOnWhilePluggedInSetting != 0

//判断mMaximumScreenOffTimeoutFromDeviceAdmin的是否处于0与Integer.MAX\_VALUE之间

//Android给出的注释是：

//The maximum allowable screen off timeout according to the device administration policy

//初始为Integer.MAX\_VALUE，因此这里是要求其它进程没有设置这个值，应该对应于强制息屏时间

&& !isMaximumScreenOffTimeoutFromDeviceAdminEnforcedLocked()) {

//判断是否充电亮屏，定义于BatteryService.java中

//从代码来看，只要mStayOnWhilePluggedInSetting设置了，就会亮屏

mStayOn = mBatteryManagerInternal.isPowered(mStayOnWhilePluggedInSetting);

} else {

mStayOn = false;

}

if (mStayOn != wasStayOn) {

mDirty |= DIRTY\_STAY\_ON;

}

}

}

3.2.3 updateScreenBrightnessBoostLocked

Android手机定义了一个最大屏幕亮度，用户可以手动或者让终端自动确定最大的屏幕亮度。updateScreenBrightnessBoostLocked函数主要用于：更新终端可处于最大屏幕亮度的时间。

为了比较好的理解updateScreenBrightnessBoostLocked函数，我们可以先分析一下与之相关的，PMS提供的对外的接口boostScreenBrightness。该方法的作用是让屏幕在一段时间内保持最大的亮度，使屏幕在强光下有更好的可读性。

@Override // Binder call

public void boostScreenBrightness(long eventTime) {

if (eventTime > SystemClock.uptimeMillis()) {

throw new IllegalArgumentException("event time must not be in the future");

}

mContext.enforceCallingOrSelfPermission(android.Manifest.permission.DEVICE\_POWER, null);

final int uid = Binder.getCallingUid();

final long ident = Binder.clearCallingIdentity();

try {

boostScreenBrightnessInternal(eventTime, uid);

} finally {

Binder.restoreCallingIdentity(ident);

}

}

private void boostScreenBrightnessInternal(long eventTime, int uid) {

synchronized (mLock) {

//系统没有准备好或者当前为Asleep状态或者该事件过时了，不处理

if (!mSystemReady || mWakefulness == WAKEFULNESS\_ASLEEP || eventTime < mLastScreenBrightnessBoostTime) {

return;

}

Slog.i(TAG, "Brightness boost activated (uid " + uid +")...");

mLastScreenBrightnessBoostTime = eventTime;//记录事件到来的时间，也可以认为是终端处于最亮屏幕状态的起始时间

if (!mScreenBrightnessBoostInProgress) {//设置最亮屏幕的标志位true

mScreenBrightnessBoostInProgress = true;

mNotifier.onScreenBrightnessBoostChanged();//发送广播

}

mDirty |= DIRTY\_SCREEN\_BRIGHTNESS\_BOOST;//修改mDirty的值，表示最大屏幕亮度的状态发生了变化

userActivityNoUpdateLocked(eventTime, PowerManager.USER\_ACTIVITY\_EVENT\_OTHER, 0, uid);//记录

updatePowerStateLocked();//更新电源状态信息

}

}

从上面的代码可以看出，该函数：

首先，使用mLastScreenBrightnessBoostTime变量记录了终端处于最大屏幕亮度的起始时间；

然后，将最大屏幕亮度的标志位置为true，并修改mDirty标志位，以表示最大屏幕亮度的状态发生了变化；

最后，调用updatePowerStateLocked方法更新电源状态信息。

我们已经知道，updatePowerStateLocked将会调用到updateScreenBrightnessBoostLocked

private void updateScreenBrightnessBoostLocked(int dirty) {

if ((dirty & DIRTY\_SCREEN\_BRIGHTNESS\_BOOST) != 0) {//根据mDirty的标志位来判断终端屏幕最大可用亮度的状态是否发生了变化

if (mScreenBrightnessBoostInProgress) {//上面的代码已经提到过，当boostScreenBrightness接口被调用时，mScreenBrightnessBoostInProgress置为true

//移除旧的超时事件

final long now = SystemClock.uptimeMillis();

mHandler.removeMessages(MSG\_SCREEN\_BRIGHTNESS\_BOOST\_TIMEOUT);

//终端处于最大屏幕亮度的时间，在sleep的时间之后，说明终端还未息屏

if (mLastScreenBrightnessBoostTime > mLastSleepTime) {

final long boostTimeout = mLastScreenBrightnessBoostTime + SCREEN\_BRIGHTNESS\_BOOST\_TIMEOUT;

if (boostTimeout > now) {

Message msg = mHandler.obtainMessage(MSG\_SCREEN\_BRIGHTNESS\_BOOST\_TIMEOUT);

msg.setAsynchronous(true);

//发送延迟的超时事件

//当屏幕离开最大亮度状态时，该事件将被发送

//当该事件被处理时，会再次进入到updateScreenBrightnessBoostLocked函数

mHandler.sendMessageAtTime(msg, boostTimeout);

return;

}

}

//进入到这个分支时，说明屏幕处于最大亮度状态的时间已经超时了

//将该标志置为false

mScreenBrightnessBoostInProgress = false;

//发送广播

mNotifier.onScreenBrightnessBoostChanged();

//触发一次用户活动，写入mDirty标志位，同时做一些其它记录

userActivityNoUpdateLocked(now, PowerManager.USER\_ACTIVITY\_EVENT\_OTHER, 0, Process.SYSTEM\_UID);

}

}

}

至此，PMS第一阶段更新基本状态信息的流程结束。

**3.3 更新wakelock和根据用户活动确定屏幕状态**

final long now = SystemClock.uptimeMillis();

int dirtyPhase2 = 0;

for (;;) {

int dirtyPhase1 = mDirty;

dirtyPhase2 |= dirtyPhase1;

mDirty = 0;

updateWakeLockSummaryLocked(dirtyPhase1);//将所有wakelock进行集中处理，更新mWakeLockSummary，来确定终端状态

updateUserActivitySummaryLocked(now, dirtyPhase1);//根据用户活动判断屏幕状态

if (!updateWakefulnessLocked(dirtyPhase1)) {

break;

}

}

3.3.1 updateWakeLockSummaryLocked

updateWakeLockSummaryLocked函数根据PMS当前持有的所有WakeLock，得到当前终端整体的信息，保存到mWakeLockSummary变量中。

这里唯一需要说明的是，Android定义一个mWakeLockSummary变量的原因是：

PMS将WakeLock定义为不同进程的请求信息，这些请求信息对CPU、屏幕和键盘有不同的需求。

对于每一种资源而言，只要有一个申请满足获取条件，PMS就需要为终端分配该申请对应的资源。

例如：假设PMS有20个WakeLock，只有1个申请亮屏，另外19个只申请CPU唤醒，PMS仍然需要保持终端亮屏。

因此，mWakeLockSummary就提供了一种整合多个WakeLock请求的功能，方便PMS进行集中的控制。

private void updateWakeLockSummaryLocked(int dirty) {

//PMS持有的WakeLock发生变化，或者唤醒状态发生变化时，才重新进行更新mWakeLockSummary

//例如：调用PMS的acquireWakeLock时，就会将dirty的DIRTY\_WAKE\_LOCKS位置1

if ((dirty & (DIRTY\_WAKE\_LOCKS | DIRTY\_WAKEFULNESS)) != 0) {

mWakeLockSummary = 0;

final int numWakeLocks = mWakeLocks.size();

for (int i = 0; i < numWakeLocks; i++) {

final WakeLock wakeLock = mWakeLocks.get(i);

//这里只关注的是wakelock的level，这些level的作用见2.1节，mWakeLockSummary对应该level置位对应的bit位

switch (wakeLock.mFlags & PowerManager.WAKE\_LOCK\_LEVEL\_MASK) {

case PowerManager.PARTIAL\_WAKE\_LOCK:

//在2.2.4节中的分析，在doze模式下，不在白名单内的非系统应用申请PARTIAL\_WAKE\_LOCK时，将被disabled

if (!wakeLock.mDisabled) {

//We only respect this if the wake lock is not disabled.

mWakeLockSummary |= WAKE\_LOCK\_CPU;

}

break;

case PowerManager.FULL\_WAKE\_LOCK:

mWakeLockSummary |= WAKE\_LOCK\_SCREEN\_BRIGHT | WAKE\_LOCK\_BUTTON\_BRIGHT;

break;

case PowerManager.SCREEN\_BRIGHT\_WAKE\_LOCK:

mWakeLockSummary |= WAKE\_LOCK\_SCREEN\_BRIGHT;

break;

case PowerManager.SCREEN\_DIM\_WAKE\_LOCK:

mWakeLockSummary |= WAKE\_LOCK\_SCREEN\_DIM;

break;

case PowerManager.PROXIMITY\_SCREEN\_OFF\_WAKE\_LOCK:

mWakeLockSummary |= WAKE\_LOCK\_PROXIMITY\_SCREEN\_OFF;

break;

case PowerManager.DOZE\_WAKE\_LOCK:

mWakeLockSummary |= WAKE\_LOCK\_DOZE;

break;

case PowerManager.DRAW\_WAKE\_LOCK:

mWakeLockSummary |= WAKE\_LOCK\_DRAW;

break;

}

}

//Cancel wake locks that make no sense based on the current state.

//从下面的代码可以看出，PMS中的mWakefulness变量记录了终端当前的状态，下面就是移除在特定状态下，没有意义的WakeLock

if (mWakefulness != WAKEFULNESS\_DOZING) {

//如果不是Dozing状态，移除相应的wakeLock标志位

mWakeLockSummary &= ~(WAKE\_LOCK\_DOZE | WAKE\_LOCK\_DRAW);

}

if (mWakefulness == WAKEFULNESS\_ASLEEP

|| (mWakeLockSummary & WAKE\_LOCK\_DOZE) != 0) {

//如果当前为Asleep或者有Doze的wakeLock锁的时候，应该移除掉屏幕亮度相关的wakeLock锁

mWakeLockSummary &= ~(WAKE\_LOCK\_SCREEN\_BRIGHT | WAKE\_LOCK\_SCREEN\_DIM

| WAKE\_LOCK\_BUTTON\_BRIGHT);

if (mWakefulness == WAKEFULNESS\_ASLEEP) {

//休眠时，sensor不再需要监听终端是否靠近物体，以触发亮灭屏

mWakeLockSummary &= ~WAKE\_LOCK\_PROXIMITY\_SCREEN\_OFF;

}

}

// Infer implied wake locks where necessary based on the current state.

// 根据当前的状态，及PMS持有的WakeLock，推断出隐含的持锁需求

// 例如：当PMS持有亮屏锁WAKE\_LOCK\_SCREEN\_BRIGHT时，若当前终端为唤醒态，那么CPU显然也需要处于唤醒态

if ((mWakeLockSummary & (WAKE\_LOCK\_SCREEN\_BRIGHT | WAKE\_LOCK\_SCREEN\_DIM)) != 0) {

if (mWakefulness == WAKEFULNESS\_AWAKE) {

mWakeLockSummary |= WAKE\_LOCK\_CPU | WAKE\_LOCK\_STAY\_AWAKE;

} else if (mWakefulness == WAKEFULNESS\_DREAMING) {

mWakeLockSummary |= WAKE\_LOCK\_CPU;

}

}

if ((mWakeLockSummary & WAKE\_LOCK\_DRAW) != 0) {

mWakeLockSummary |= WAKE\_LOCK\_CPU;

}

if (DEBUG\_SPEW) {...}

}

}

3.3.2 updateUserActivitySummaryLocked

updateUserActivitySummaryLocked主要根据用户最后的活动来决定当前屏幕的状态

从下面的代码可以看出，在该函数中用mUserActivitySummary变量存储当前屏幕的状态

一共有3中基本状态：

\* USER\_ACTIVITY\_SCREEN\_BRIGHT 点亮屏幕

\* USER\_ACTIVITY\_SCREEN\_DIM 屏幕变暗

\* USER\_ACTIVITY\_SCREEN\_DREAM 屏保状态

从代码可以看出，屏幕变化和userActivity活动有关，它根据最后的userActivity活动的时间决定点亮屏幕、调暗屏幕或熄灭屏幕。

之前的很多方法中都会调用userActivityNoUpdateLocked方法。该方法将触发一次用户活动，以更新用户活动的时间，这样屏幕变暗和熄灭时间就会重新进行计算。 这也就是为什么用户一直操作手机，屏幕不会熄灭或者变暗的原因。

流程图可以见updateUserActivitySummaryLocked.jepg，这些时间计算还是挺绕的。

private void updateUserActivitySummaryLocked(long now, int dirty) {

// Update the status of the user activity timeout timer.

if ((dirty & (DIRTY\_WAKE\_LOCKS | DIRTY\_USER\_ACTIVITY

| DIRTY\_WAKEFULNESS | DIRTY\_SETTINGS)) != 0) {

mHandler.removeMessages(MSG\_USER\_ACTIVITY\_TIMEOUT);

long nextTimeout = 0;

if (mWakefulness == WAKEFULNESS\_AWAKE

|| mWakefulness == WAKEFULNESS\_DREAMING

|| mWakefulness == WAKEFULNESS\_DOZING) {

//获取进入休眠状态的sleepTimeout，getSleepTimeoutLocked中会判断休眠timeout和屏幕熄灭timeout的关系，如果休眠timeout小于屏幕熄灭timeout，

//则休眠timeout被调整为屏幕熄灭timeout，因为屏幕亮屏状态下，终端不能进入休眠

final int sleepTimeout = getSleepTimeoutLocked();

//获取屏幕熄灭screenOffTimeout

final int screenOffTimeout = getScreenOffTimeoutLocked(sleepTimeout);

//获取屏幕变暗的持续时间

final int screenDimDuration = getScreenDimDurationLocked(screenOffTimeout);

//当Window Manager判定用户inactive时，将此标志userInactiveOverride 置为true

final boolean userInactiveOverride = mUserInactiveOverrideFromWindowManager;

//类似于之前的mWakeLockSummary，将当前的用户事件，转化为PMS可以处理的屏幕状态

mUserActivitySummary = 0;

//用户最近activity时间大于 mLastWakeTime，即在唤醒的状态下，发生过用户事件

if (mLastUserActivityTime >= mLastWakeTime) {

//重新计算出屏幕需要变暗的时间

nextTimeout = mLastUserActivityTime + screenOffTimeout - screenDimDuration;

if (now < nextTimeout) {

//如果没有到达需要变暗的时间，那么当前屏幕的状态为USER\_ACTIVITY\_SCREEN\_BRIGHT(亮屏)

mUserActivitySummary = USER\_ACTIVITY\_SCREEN\_BRIGHT;

} else {

//到达变暗的时间，则计算出屏幕熄灭的时间，最近的user活动时间+屏幕熄灭timeout

nextTimeout = mLastUserActivityTime + screenOffTimeout;

if (now < nextTimeout) {

//还没到熄灭的时间，则当前屏幕的状态为USER\_ACTIVITY\_SCREEN\_DIM(暗屏)

mUserActivitySummary = USER\_ACTIVITY\_SCREEN\_DIM;

}

}

}

//注意mUserActivitySummary为0才会进入下面的分支

//即上面改变mUserActivitySummary的条件不满足时，才会进入这个分支(例如：唤醒状态下，没发生过改变屏幕状态的UserActivity)

if (mUserActivitySummary == 0

//mLastUserActivityTimeNoChangeLights表示用户最后的活动不会改变屏幕当前的状态

&& mLastUserActivityTimeNoChangeLights >= mLastWakeTime) {

//计算下次屏幕熄灭的时间

nextTimeout = mLastUserActivityTimeNoChangeLights + screenOffTimeout;

//还未到达息屏时间

if (now < nextTimeout) {

if (mDisplayPowerRequest.policy == DisplayPowerRequest.POLICY\_BRIGHT

|| mDisplayPowerRequest.policy == DisplayPowerRequest.POLICY\_VR) {

//当前屏幕是亮屏，仍然设置为亮屏

mUserActivitySummary = USER\_ACTIVITY\_SCREEN\_BRIGHT;

} else if (mDisplayPowerRequest.policy == DisplayPowerRequest.POLICY\_DIM) {

//当前屏幕是变暗，仍然设置为变暗

mUserActivitySummary = USER\_ACTIVITY\_SCREEN\_DIM;

}

}

}

if (mUserActivitySummary == 0) {

//若定义了有效的休眠时间

if (sleepTimeout >= 0) {

final long anyUserActivity = Math.max(mLastUserActivityTime, mLastUserActivityTimeNoChangeLights);

//最近的一次用户活动时间大于最近休眠的时间，重新更新休眠时间，此时，应该是有过用户活动，但过了息屏时间了(黑屏下的用户活动？)

if (anyUserActivity >= mLastWakeTime) {

nextTimeout = anyUserActivity + sleepTimeout;

if (now < nextTimeout) {  
 //走到这个分支，应该是屏幕已经熄灭，但还未到达休眠状态，先进入dream态

mUserActivitySummary = USER\_ACTIVITY\_SCREEN\_DREAM;

}

}

} else {

//直接进入dream态，后续的updateWakefulnessLocked将判断是否休眠

mUserActivitySummary = USER\_ACTIVITY\_SCREEN\_DREAM;

nextTimeout = -1;

}

}

//如果屏幕未进入dream态，但Window Manager判定用户inactive，则进入下面分支

if (mUserActivitySummary != USER\_ACTIVITY\_SCREEN\_DREAM && userInactiveOverride) {

//如果屏幕未熄灭

if ((mUserActivitySummary & (USER\_ACTIVITY\_SCREEN\_BRIGHT | USER\_ACTIVITY\_SCREEN\_DIM)) != 0) {

// Device is being kept awake by recent user activity

if (nextTimeout >= now && mOverriddenTimeout == -1) {

// Save when the next timeout would have occurred

mOverriddenTimeout = nextTimeout;

}

}

//Window Manager的权限很大，如果它判断用户inactive，直接进入dream态

mUserActivitySummary = USER\_ACTIVITY\_SCREEN\_DREAM;

nextTimeout = -1;

}

//根据nextTimeout延迟发送信息，信息被处理后，将重新调用updatePowerStateLocked，于是再次进入到该方法

//通过不断进入该方法，不断评估是否根据用户动作亮、熄屏等

if (mUserActivitySummary != 0 && nextTimeout >= 0) {

Message msg = mHandler.obtainMessage(MSG\_USER\_ACTIVITY\_TIMEOUT);

msg.setAsynchronous(true);

mHandler.sendMessageAtTime(msg, nextTimeout);

}

} else {

mUserActivitySummary = 0;

}

if (DEBUG\_SPEW) {...}

}

}

3.3.3 updateWakefulnessLocked

updateWakefulnessLocked将决定第二阶段的电源状态更新是否结束，从下面的代码可以看出，如果终端可以一直保持唤醒状态，或一开始就是非唤醒态， 那么mWakefulness不会发生改变，第二阶段的for循环将会break；

如果终端要从唤醒态变为非唤醒态，那么for循环将再运行一次，即重新计算一次mWakeLockSummary和mUserActivitySummary。这么做的原因是：updateWakeLockSummaryLocked和updateUserActivitySummaryLocked函数的一些计算，与终端是否处于唤醒状态，即mWakefulness的值有关。由于这两个函数并不会修改mWakefulness，因此在这一次运行时，updateWakefulnessLocked将返回false，即第二阶段结束。

因此，我们可以得出结论：更新电源状态的第二阶段，正常情况下最多运行两次

private boolean updateWakefulnessLocked(int dirty) {

boolean changed = false;

//下面的条件还是比较容易满足的，基本上只要之前的流程更改过mDirty就会进入分支

if ((dirty & (DIRTY\_WAKE\_LOCKS | DIRTY\_USER\_ACTIVITY | DIRTY\_BOOT\_COMPLETED

| DIRTY\_WAKEFULNESS | DIRTY\_STAY\_ON | DIRTY\_PROXIMITY\_POSITIVE

| DIRTY\_DOCK\_STATE)) != 0) {

//如果当前的状态是唤醒的，isItBedTimeYetLocked()->isBeingKeptAwakeLocked()根据状态，判断终端是否应该处于唤醒状态

if (mWakefulness == WAKEFULNESS\_AWAKE && isItBedTimeYetLocked()) {

if (DEBUG\_SPEW) {

Slog.d(TAG, "updateWakefulnessLocked: Bed time...");

}

final long time = SystemClock.uptimeMillis();

//主要根据设置信息，判断是否满足进入Dream状态的条件

if (shouldNapAtBedTimeLocked()) {

//将mWakefullness的值置为WAKEFULNESS\_DREAMING，修改mDirty变量，并进行通知等

changed = napNoUpdateLocked(time, Process.SYSTEM\_UID);

} else {

//将mWakefullness的值置为WAKEFULNESS\_DOZING

//如果系统设置了跳过Dozing态，则将mWakefullness置为WAKEFULNESS\_ASLEEP

//同时修改mDirty变量，并进行通知等

changed = goToSleepNoUpdateLocked(time,

PowerManager.GO\_TO\_SLEEP\_REASON\_TIMEOUT, 0, Process.SYSTEM\_UID);

}

//注意：napNoUpdateLocked和goToSleepNoUpdateLocked函数正常执行后，都会将mSandmanSummoned(被"睡魔"眷顾了)置为true

}

}

return changed;

}

**3.4 更新display power state**

第三阶段将负责更新屏幕的显示状态。

private boolean updateDisplayPowerStateLocked(int dirty) {

final boolean oldDisplayReady = mDisplayReady;

//mDirty满足条件时，进入以下分支

if ((dirty & (DIRTY\_WAKE\_LOCKS | DIRTY\_USER\_ACTIVITY | DIRTY\_WAKEFULNESS

| DIRTY\_ACTUAL\_DISPLAY\_POWER\_STATE\_UPDATED | DIRTY\_BOOT\_COMPLETED

| DIRTY\_SETTINGS | DIRTY\_SCREEN\_BRIGHTNESS\_BOOST | DIRTY\_VR\_MODE\_CHANGED |

DIRTY\_QUIESCENT)) != 0) {

//根据mWakefullness、mWakeLockSummary、mUserActivitySummary等，决定屏幕的policy

//policy定义为DisplayPowerRequest.(POLICY\_OFF、POLICY\_DOZE、POLICY\_BRIGHT和POLICY\_DIM)

mDisplayPowerRequest.policy = getDesiredScreenPolicyLocked();

// Determine appropriate screen brightness and auto-brightness adjustments.

//决定屏幕的亮度

//......

// Update display power request.

// 更新mDisplayPowerRequest的参数

//......

//实际上调用DisplayPowerController的requestPowerState函数

//在初始时，PMS注册了mDisplayPowerCallbacks到DisplayPowerController中，

//当更新完成后，会回调定义的接口，重新updatePowerStateLocked

mDisplayReady = mDisplayManagerInternal.requestPowerState(mDisplayPowerRequest,

mRequestWaitForNegativeProximity);

mRequestWaitForNegativeProximity = false;

//......

return mDisplayReady && !oldDisplayReady;

}

我们简单看下注册流程，在PMS的systemReady中提到过

mDisplayManagerInternal = getLocalService(DisplayManagerInternal.class);

mDisplayManagerInternal.initPowerManagement(mDisplayPowerCallbacks, mHandler, sensorManager);

我们进入frameworks/base/services/core/java/com/android/server/display/DisplayManagerService.java中

private final class LocalService extends DisplayManagerInternal {

public void initPowerManagement(final DisplayPowerCallbacks callbacks, Handler handler, SensorManager sensorManager) {

...

mDisplayPowerController = new DisplayPowerController(mContext, callbacks, handler, sensorManager, blanker);

…

}

}

PMS中注册的callbacks

private final DisplayManagerInternal.DisplayPowerCallbacks mDisplayPowerCallbacks =

new DisplayManagerInternal.DisplayPowerCallbacks() {

private int mDisplayState = Display.STATE\_UNKNOWN;

@Override

public void onStateChanged() {...updatePowerStateLocked();}

@Override

public void onProximityPositive() {...updatePowerStateLocked();}

@Override

public void onProximityNegative() {... updatePowerStateLocked();}

@Override

public void onDisplayStateChange(int state) {...}

@Override

public void acquireSuspendBlocker() {...}

@Override

public void releaseSuspendBlocker() {...}

@Override

public String toString() {...}

};

所以当DisplayPowerController的requestPowerState函数执行时，会回调PMS注册的callbacks

public boolean requestPowerState(DisplayPowerRequest request, boolean waitForNegativeProximity) {

if (DEBUG) {//pensor debug log

Slog.d(TAG, "requestPowerState: " + request + ", waitForNegativeProximity=" + waitForNegativeProximity);

}

synchronized (mLock) {

boolean changed = false;

//proximity sensor需要检测距离 && 没有等待检测

if (waitForNegativeProximity && !mPendingWaitForNegativeProximityLocked) {

mPendingWaitForNegativeProximityLocked = true;

changed = true;

}

//以下表示，参数中的Request对于DisplayPowerController而言，是一个新的需求

if (mPendingRequestLocked == null) {

mPendingRequestLocked = new DisplayPowerRequest(request);

changed = true;

} else if (!mPendingRequestLocked.equals(request)) {

mPendingRequestLocked.copyFrom(request);

changed = true;

}

if (changed) {

//一但有新的需求，mDisplayReadyLocked就是false，表示屏幕有待调整

mDisplayReadyLocked = false;

}

//有新需求，同时有对应的request

if (changed && !mPendingRequestChangedLocked) {

mPendingRequestChangedLocked = true;

//发送消息，更新屏幕状态

//最终通过DisplayPowerController的updatePowerState函数，进行屏幕状态更新

//这部分代码也极其复杂，暂时不在这里展开分析

//更新屏幕状态后，将回调PMS的接口

sendUpdatePowerStateLocked();

}

return mDisplayReadyLocked;

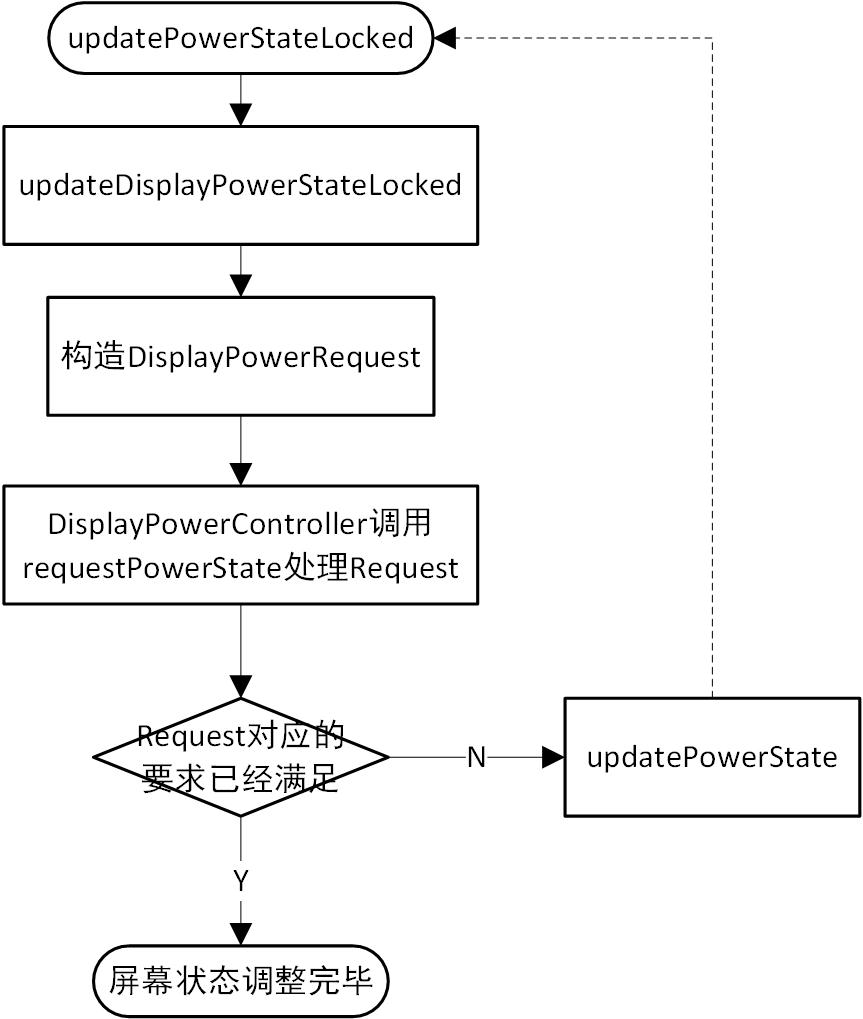
}

}

根据requestPowerState的代码，我们知道：

当PMS传入一个新的mDisplayPowerRequest时，requestPowerState应该返回为false；当DisplayPowerController按照mDisplayPowerRequest修改完屏幕状态，再次进入回到updateDisplayPowerStateLocked函数，调用requestPowerState时才会返回true。

整体的代码流程大概可以抽象成下图：

这一阶段的代码，我们只是分析了整个过程的冰山一角，并没有分析更新屏幕状态的实际操作。 

但从现有的代码可以看出，PMS的作用仅仅是维护终端电源相关状态，实际的工作还是通过类似发送Request的方式，让其它的服务协助完成。

例如：在整个阶段，PMS根据之前得到信息，构造出DisplayPowerRequest，然后发送给DisplayPowerController进行实际的处理。

当DisplayPowerController完成实际的工作(部分工作还依赖于PhoneWindowManager)后，再通知PMS进行复查。

因此PMS的定位，确实可以用一个”Manager”来形容；

负责整个终端信息的搜集和维护，然后将相应的工作指派给具体的“员工”执行；

“员工”执行完毕后，向”Manager”汇报；

“Manager”检查工作的完成情况后，然后做出下一步的指示。

**3.5 更新dream state**

updateDreamLocked函数主要用于更新屏保状态，当设备进入或者退出屏保的时候都会触发这个方法：

private void updateDreamLocked(int dirty, boolean displayBecameReady) {

if ((dirty & (DIRTY\_WAKEFULNESS

| DIRTY\_USER\_ACTIVITY

| DIRTY\_WAKE\_LOCKS

| DIRTY\_BOOT\_COMPLETED

| DIRTY\_SETTINGS

| DIRTY\_IS\_POWERED

| DIRTY\_STAY\_ON

| DIRTY\_PROXIMITY\_POSITIVE

| DIRTY\_BATTERY\_STATE)) != 0 || displayBecameReady) {

if (mDisplayReady) {

scheduleSandmanLocked();

}

}

}

handleSandman函数比较复杂，主要用于决定设备是否应该停留在dreaming或dozing状态。

private void scheduleSandmanLocked() {

if (!mSandmanScheduled) {

//mSandmanScheduled的作用就是让MessageQueue中仅保留一个MSG\_SANDMAN???

mSandmanScheduled = true;

Message msg = mHandler.obtainMessage(MSG\_SANDMAN);//由handleSandman处理

msg.setAsynchronous(true);

mHandler.sendMessage(msg);

}

}

我们分段介绍该函数的功能

▪决定是否可以进入屏保状态

▪在必要时，进入屏保状态

▪更新屏保状态

3.5.1决定是否可以进入屏保状态

private void handleSandman() {

//Handle preconditions.

final boolean startDreaming;

final int wakefulness;

synchronized (mLock) {

mSandmanScheduled = false;

wakefulness = mWakefulness;

//前面提到过，当updateWakefulnessLocked判断进入dozing或sleep状态时，会将mSandmanSummoned置为true，mDisplayReady主要确保前面屏幕状态更新完毕

if (mSandmanSummoned && mDisplayReady) {

//判断device是否可以dream或dozing

startDreaming = canDreamLocked() || canDozeLocked();

mSandmanSummoned = false;

} else {

startDreaming = false;

}

}

......

除去前置条件的限制外，此处的结果主要由canDreamLocked和canDozeLocked决定。

private boolean canDreamLocked() {//dreaming除了对终端当前的状态、配置项有关外，在非唤醒状态下还与当前的电池电量有关系

//mWakefulness不等于WAKEFULNESS\_DREAMING

if (mWakefulness != WAKEFULNESS\_DREAMING

|| !mDreamsSupportedConfig//设备不支持dreaming

|| !mDreamsEnabledSetting//设置开关未开启

|| !mDisplayPowerRequest.isBrightOrDim()//屏幕未熄灭

|| mDisplayPowerRequest.isVr()//是Vr?

|| (mUserActivitySummary & (USER\_ACTIVITY\_SCREEN\_BRIGHT

| USER\_ACTIVITY\_SCREEN\_DIM | USER\_ACTIVITY\_SCREEN\_DREAM)) == 0

|| !mBootCompleted) {//初始化未完成

return false;

}//以上条件只要有一项就会return false

if (!isBeingKeptAwakeLocked()) {//不处于唤醒态

//没充电，电源选项也未配置，不可dreaming

if (!mIsPowered && !mDreamsEnabledOnBatteryConfig) {

return false;

}

if (!mIsPowered//没充电，且电池电量过低，不可dreaming

&& mDreamsBatteryLevelMinimumWhenNotPoweredConfig >= 0

&& mBatteryLevel < mDreamsBatteryLevelMinimumWhenNotPoweredConfig) {

return false;

}

if (mIsPowered//充电，但电池电量过低，不可dreaming

&& mDreamsBatteryLevelMinimumWhenPoweredConfig >= 0

&& mBatteryLevel < mDreamsBatteryLevelMinimumWhenPoweredConfig) {

return false;

}

//充电和未充电分别有一个最低的dreaming电量门限

}

return true;

}

private boolean canDozeLocked() {//canDozeLocked函数相对简单

return mWakefulness == WAKEFULNESS\_DOZING;

}

3.5.2在必要时，进入屏保状态

决定了是否可以进入屏保状态后，这一部分就开始进行实际的工作。

//start dreaming if needed.

final boolean isDreaming;

if (mDreamManager != null) {

if (startDreaming) {

mDreamManager.stopDream(false /\*immediate\*/);//结束旧梦

mDreamManager.startDream(wakefulness == WAKEFULNESS\_DOZING);//开启新梦

}

//startDream成功后，一般isDreaming就会返回true

isDreaming = mDreamManager.isDreaming();

} else {

isDreaming = false;

}

我们重点看看DreamManagerService的startDream函数，stopDream的工作内容与startDream相反，不做细致分析

//定义于DreamManagerService的内部类中

public void startDream(boolean doze) {

startDreamInternal(doze);

}

//定义于DreamManagerService

private void startDreamInternal(boolean doze) {

final int userId = ActivityManager.getCurrentUser();

//个人觉得这里应该是获取屏保对象

final ComponentName dream = chooseDreamForUser(doze, userId);

if (dream != null) {

synchronized (mLock) {

startDreamLocked(dream, false /\*isTest\*/, doze, userId);

}

}

}

private void startDreamLocked(final ComponentName name,

final boolean isTest, final boolean canDoze, final int userId) {

if (Objects.equal(mCurrentDreamName, name)//申请的屏保与当前的一致，不用进行修改

&& mCurrentDreamIsTest == isTest

&& mCurrentDreamCanDoze == canDoze

&& mCurrentDreamUserId == userId) {

Slog.i(TAG, "Already in target dream.");

return;

}

//立即停止当前的屏保

stopDreamLocked(true /\*immediate\*/);

Slog.i(TAG, "Entering dreamland.");

final Binder newToken = new Binder();

mCurrentDreamToken = newToken;

mCurrentDreamName = name;

mCurrentDreamIsTest = isTest;

mCurrentDreamCanDoze = canDoze;

mCurrentDreamUserId = userId;

PowerManager.WakeLock wakeLock = mPowerManager.newWakeLock(PowerManager.PARTIAL\_WAKE\_LOCK, "startDream");

mHandler.post(wakeLock.wrap(

() -> mController.startDream(newToken, name, isTest, canDoze, userId, wakeLock)));//调用DreamController的startDream函数

}

public void startDream(Binder token, ComponentName name,

boolean isTest, boolean canDoze, int userId, PowerManager.WakeLock wakeLock) {//定义于DreamController中

//移除当前屏保并回调通知

stopDream(true /\*immediate\*/);

Trace.traceBegin(Trace.TRACE\_TAG\_POWER, "startDream");

try {

// Close the notification shade. No need to send to all, but better to be explicit.

mContext.sendBroadcastAsUser(mCloseNotificationShadeIntent, UserHandle.ALL);

...

//记录dream

mCurrentDream = new DreamRecord(token, name, isTest, canDoze, userId, wakeLock);

mDreamStartTime = SystemClock.elapsedRealtime();

MetricsLogger.visible(mContext,

mCurrentDream.mCanDoze ? MetricsEvent.DOZING : MetricsEvent.DREAMING);

try {//做好屏幕相关的准备工作

mIWindowManager.addWindowToken(token, TYPE\_DREAM, DEFAULT\_DISPLAY);

} catch (RemoteException ex) {

Slog.e(TAG, "Unable to add window token for dream.", ex);

stopDream(true /\*immediate\*/);

return;

}

Intent intent = new Intent(DreamService.SERVICE\_INTERFACE);

intent.setComponent(name);

intent.addFlags(Intent.FLAG\_ACTIVITY\_EXCLUDE\_FROM\_RECENTS);

try {

//拉起屏保服务

if (!mContext.bindServiceAsUser(intent, mCurrentDream,

Context.BIND\_AUTO\_CREATE | Context.BIND\_FOREGROUND\_SERVICE,

new UserHandle(userId))) {

Slog.e(TAG, "Unable to bind dream service: " + intent);

stopDream(true /\*immediate\*/);

return;

}

} catch (SecurityException ex) {

Slog.e(TAG, "Unable to bind dream service: " + intent, ex);

stopDream(true /\*immediate\*/);

return;

}

mCurrentDream.mBound = true;

//在DREAM\_CONNECTION\_TIMEOUT到期时，bind服务还未成功，runnable就负责结束dream

mHandler.postDelayed(mStopUnconnectedDreamRunnable, DREAM\_CONNECTION\_TIMEOUT);

} finally {

Trace.traceEnd(Trace.TRACE\_TAG\_POWER);

}

}

从这部分代码我们知道了，所谓的屏保其实就是拉起一个特殊的服务。

3.5.3更新屏保状态

// Update dream state.

synchronized (mLock) {

// Remember the initial battery level when the dream started.

if (startDreaming && isDreaming) {

mBatteryLevelWhenDreamStarted = mBatteryLevel;

//......

}

// If preconditions changed, wait for the next iteration to determine

// whether the dream should continue (or be restarted).

if (mSandmanSummoned || mWakefulness != wakefulness) {

return; // wait for next cycle

}

// Determine whether the dream should continue.

if (wakefulness == WAKEFULNESS\_DREAMING) {

if (isDreaming && canDreamLocked()) {

if (mDreamsBatteryLevelDrainCutoffConfig >= 0

&& mBatteryLevel < mBatteryLevelWhenDreamStarted //在dream开始的时候，电池电量的百分比。

- mDreamsBatteryLevelDrainCutoffConfig //电流下降到该百分点，当用户活动超时后不进入屏保，默认5，这个mXxxXxx回来要出个章节整理一下

&& !isBeingKeptAwakeLocked()) {//且没有处于keep awake状态

// If the user activity timeout expired and the battery appears

// to be draining faster than it is charging then stop dreaming

// and go to sleep.

} else {

return; // continue dreaming

}

}

// Dream has ended or will be stopped. Update the power state.

if (isItBedTimeYetLocked()) {//根据状态，判断终端是否应该处于唤醒状态，当我们离开dream时，我们也会决定是否应该完全清醒，还是睡个好觉。

//休眠

goToSleepNoUpdateLocked(SystemClock.uptimeMillis(), PowerManager.GO\_TO\_SLEEP\_REASON\_TIMEOUT, 0, Process.SYSTEM\_UID);

updatePowerStateLocked();

} else {

//唤醒

wakeUpNoUpdateLocked(SystemClock.uptimeMillis(), "android.server.power:DREAM", Process.SYSTEM\_UID, mContext.getOpPackageName(), Process.SYSTEM\_UID);

updatePowerStateLocked();

}

} else if (wakefulness == WAKEFULNESS\_DOZING) {

if (isDreaming) {

return; // continue dozing

}

// Doze has ended or will be stopped. Update the power state.

reallyGoToSleepNoUpdateLocked(SystemClock.uptimeMillis(), Process.SYSTEM\_UID);

updatePowerStateLocked();

}

}

// Stop dream.

// 执行到这里说明退出了dreaming状态，如果之前拉起过屏保服务，此时应该停止它

if (isDreaming) {

mDreamManager.stopDream(false /\*immediate\*/);

}

以上是PMS更新屏保状态的基本流程，这部分代码最后太乱 ，整体来看相当的繁琐 ，每次更新状态后都会重新调用updatePowerStateLocked，然后再次进入到handleSandman函数中。 这种反复地递归调用，比较难以把控，我们还是用一个图来整体整理一下。见PMS更新屏保状态基本流程图.jepg

**3.6 更新suspend blocker**

updateSuspendBlockerLocked函数主要根据之前流程的执行结果，持有或者释放CPU和屏幕的锁。我们一起来看看对应的函数：

private void updateSuspendBlockerLocked() {

//根据是否有CPU的wakelock，来决定cpu是保持否唤醒

final boolean needWakeLockSuspendBlocker = ((mWakeLockSummary & WAKE\_LOCK\_CPU) != 0);

//根据前面屏幕相关的状态，来决定是否需要持有屏幕的锁

final boolean needDisplaySuspendBlocker = needDisplaySuspendBlockerLocked();

//屏幕如果不需要保持开启状态，那么使能autoSuspend，最后/system/core/libsuspend/

final boolean autoSuspend = !needDisplaySuspendBlocker;

//高通底层实现是亮灭屏，会设置对应当前governor的相关节点，但是发现是一个半成品，没有用上，而且interactive也淘汰不用了

final boolean interactive = mDisplayPowerRequest.isBrightOrDim();

// Disable auto-suspend if needed.

// FIXME We should consider just leaving auto-suspend enabled forever since we already hold the necessary wakelocks.

if (!autoSuspend && mDecoupleHalAutoSuspendModeFromDisplayConfig) {//autoSuspend为false，说明屏幕还需要点亮

setHalAutoSuspendModeLocked(false);//通过native函数，调用底层的autosuspend\_disable

}

// First acquire suspend blockers if needed.

// 在需要的情况下，获取CPU和屏幕的锁

if (needWakeLockSuspendBlocker && !mHoldingWakeLockSuspendBlocker) {

mWakeLockSuspendBlocker.acquire();

mHoldingWakeLockSuspendBlocker = true;

}

if (needDisplaySuspendBlocker && !mHoldingDisplaySuspendBlocker) {

mDisplaySuspendBlocker.acquire();

mHoldingDisplaySuspendBlocker = true;

}

// Inform the power HAL about interactive mode.

// Although we could set interactive strictly based on the wakefulness as reported by isInteractive(), it is actually more desirable to track the display policy state instead so that the

//interactive state observed by the HAL more accurately tracks transitions between AWAKE and DOZING.

// Refer to getDesiredScreenPolicyLocked() for details.

if (mDecoupleHalInteractiveModeFromDisplayConfig) {

// When becoming non-interactive, we want to defer sending this signal until the display is actually ready so that all transitions have completed.

// This is probably a good sign that things have gotten too tangled over here...

if (interactive || mDisplayReady) {

setHalInteractiveModeLocked(interactive);

}

}

// Then release suspend blockers if needed.

if (!needWakeLockSuspendBlocker && mHoldingWakeLockSuspendBlocker) {

mWakeLockSuspendBlocker.release();

mHoldingWakeLockSuspendBlocker = false;

}

if (!needDisplaySuspendBlocker && mHoldingDisplaySuspendBlocker) {

mDisplaySuspendBlocker.release();

mHoldingDisplaySuspendBlocker = false;

}

// Enable auto-suspend if needed.

if (autoSuspend && mDecoupleHalAutoSuspendModeFromDisplayConfig) {

setHalAutoSuspendModeLocked(true);

}

}

从上面的代码可以看出PMS是非常依赖于native层的，真实的持锁、释放锁、interactive等工作，均是移交到native层进行操作。

我们以mWakeLockSuspendBlocker的处理流程为例，看看native的调用过程：

mWakeLockSuspendBlocker = createSuspendBlockerLocked("PowerManagerService.WakeLocks");

之前的博客也提到过，PMS在其构造函数中调用createSuspendBlockerLocked函数，创建出了mWakeLockSuspendBlocker：

private SuspendBlocker createSuspendBlockerLocked(String name) {

SuspendBlocker suspendBlocker = new SuspendBlockerImpl(name);

mSuspendBlockers.add(suspendBlocker);

return suspendBlocker;

}

从上面的代码，我们知道当PMS需要获取底层锁时，调用的是SuspendBlockerImpl的acquire函数

public void acquire() {

synchronized (this) {

mReferenceCount += 1;

if (mReferenceCount == 1) {

//......

nativeAcquireSuspendBlocker(mName);

}

}

}

static void nativeAcquireSuspendBlocker(JNIEnv \*env, jclass /\* clazz \*/, jstring nameStr) {

ScopedUtfChars name(env, nameStr);

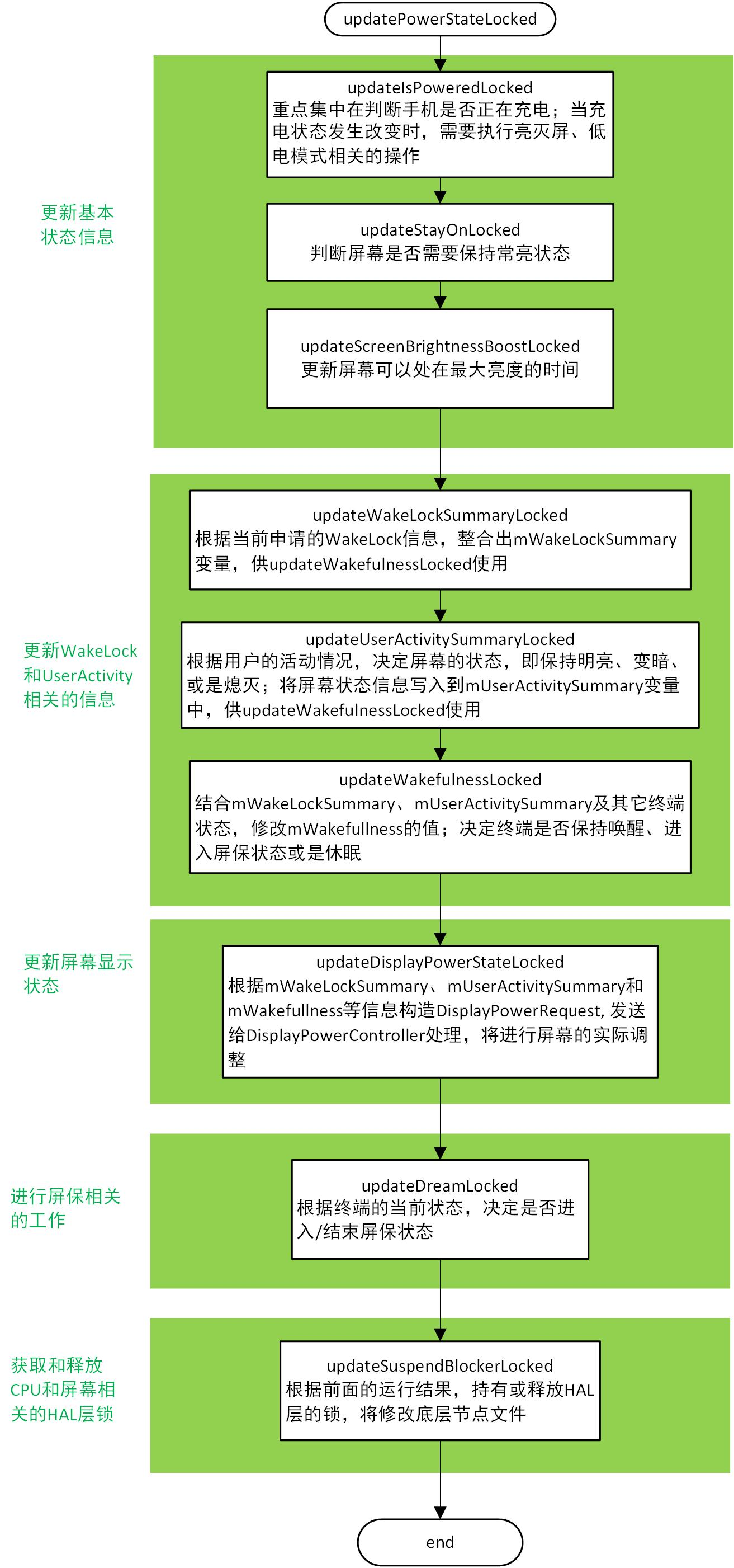
//获取的是PARTIAL\_WAKE\_LOCK的类型，即保持CPU唤醒的

acquire\_wake\_lock(PARTIAL\_WAKE\_LOCK, name.c\_str());

}

HAL层函数acquire\_wake\_lock，最后会向/sys/power/wake\_lock节点进行write操作。

总结，至此，updatePowerStateLocked的基本流程介绍完毕，大体上如下图所示：



**4 PMS Power按键流程**

按键的处理主要由InputManagerService负责，属于Android输入系统的流程。在这篇博客里，我们只关注与Power键相关的内容。InputManagerService处理的按键事件，最终将会传递到PhoneWindowManager的interceptKeyBeforeQueueing函数。

我们就从这个函数开始，逐步进行分析。

public int interceptKeyBeforeQueueing(KeyEvent event, int policyFlags) {

if (!mSystemBooted) {

// If we have not yet booted, don't let key events do anything.

return 0;

}

//表示屏幕是否点亮

final boolean interactive = (policyFlags & FLAG\_INTERACTIVE) != 0;

final boolean down = event.getAction() == KeyEvent.ACTION\_DOWN;

final boolean canceled = event.isCanceled();

//按键对应的编码

final int keyCode = event.getKeyCode();

.................

switch (keyCode) {

.................

case KeyEvent.KEYCODE\_POWER: {

// Any activity on the power button stops the accessibility shortcut

cancelPendingAccessibilityShortcutAction();

result &= ~ACTION\_PASS\_TO\_USER;

isWakeKey = false; // wake-up will be handled separately

if (down) {

interceptPowerKeyDown(event, interactive);//处理按下Power键

} else {

interceptPowerKeyUp(event, interactive, canceled);//处理松开Power键

}

break;

}

}

.................

if (isWakeKey) {

//按power键时，isWakeKey置为false，于是不会调用wakeUp函数

wakeUp(event.getEventTime(), mAllowTheaterModeWakeFromKey, "android.policy:KEY");

}

return result;

}

接下来，我们分别看一下interceptPowerKeyDown和interceptPowerKeyUp函数

**4.1 处理按下Power键 - interceptPowerKeyDown**

interceptPowerKeyDown用于处理按下Power键(还未松手释放)对应的事件。

private void interceptPowerKeyDown(KeyEvent event, boolean interactive) {

// Hold a wake lock until the power key is released.

// mPowerKeyWakeLock为PARTIAL\_WAKE\_LOCK级别的锁

if (!mPowerKeyWakeLock.isHeld()) {

mPowerKeyWakeLock.acquire();// 将调用到PMS的acquire WakeLock流程

}

// Cancel multi-press detection timeout.

//处理多次按power键的场景

//每次power up时，发送MSG\_POWER\_DELAYED\_PRESS的延迟消息

//如果延迟消息被处理，说明一次完整的Power键处理结束（按下去，弹起来）

//在延迟消息被处理前，再次按power键，就检测到多次点击了

if (mPowerKeyPressCounter != 0) {

mHandler.removeMessages(MSG\_POWER\_DELAYED\_PRESS);

}

// Detect user pressing the power button in panic when an application has taken over the whole screen.

//从注释来看及mHiddenNavPanic的代码，觉得像是处理“误触”的

boolean panic = mImmersiveModeConfirmation.onPowerKeyDown(interactive,

SystemClock.elapsedRealtime(), isImmersiveMode(mLastSystemUiFlags),

isNavBarEmpty(mLastSystemUiFlags));

if (panic) {

mHandler.post(mHiddenNavPanic);

}

// Latch power key state to detect screenshot chord.

//如果当前是亮屏状态，且满足触发截屏的条件，触发截屏功能

if (interactive && !mScreenshotChordPowerKeyTriggered

&& (event.getFlags() & KeyEvent.FLAG\_FALLBACK) == 0) {

mScreenshotChordPowerKeyTriggered = true;

mScreenshotChordPowerKeyTime = event.getDownTime();

interceptScreenshotChord();

}

// Stop ringing or end call if configured to do so when power is pressed.

TelecomManager telecomManager = getTelecommService();

boolean hungUp = false;

if (telecomManager != null) {

if (telecomManager.isRinging()) {

// Pressing Power while there's a ringing incoming call should silence the ringer.

// 如果有电话拨入，且电话铃声响起，按Power键，设置电话响铃静音

telecomManager.silenceRinger();

} else if ((mIncallPowerBehavior

& Settings.Secure.INCALL\_POWER\_BUTTON\_BEHAVIOR\_HANGUP) != 0

&& telecomManager.isInCall() && interactive) {

// Otherwise, if "Power button ends call" is enabled, the Power button will hang up any current active call.

// 如果正在接听电话，且配置了Power键挂断电话的话，按Power按键挂断正在接听的电话

hungUp = telecomManager.endCall();

}

}

GestureLauncherService gestureService = LocalServices.getService(

GestureLauncherService.class);

boolean gesturedServiceIntercepted = false;

if (gestureService != null) {

//手势对应的服务，尝试拦截处理Power键动作事件

gesturedServiceIntercepted = gestureService.interceptPowerKeyDown(event, interactive,

mTmpBoolean);

if (mTmpBoolean.value && mRequestedOrGoingToSleep) {

mCameraGestureTriggeredDuringGoingToSleep = true;

}

}

// Inform the StatusBar; but do not allow it to consume the event.

sendSystemKeyToStatusBarAsync(event.getKeyCode());

// If the power key has still not yet been handled, then detect short press, long press, or multi press and decide what to do.

// Power键事件的处理，就像处理屏幕上的点击事件一样，也依赖于事件分发机制，如果已经被消耗掉了，就不会再被继续处理

mPowerKeyHandled = hungUp || mScreenshotChordVolumeDownKeyTriggered

|| mA11yShortcutChordVolumeUpKeyTriggered || gesturedServiceIntercepted;

// Power键事件未被消耗掉

if (!mPowerKeyHandled) {

// 屏幕还是亮的

if (interactive) {

// When interactive, we're already awake.

// Wait for a long press or for the button to be released to decide what to do.

// 1、判断是否支持长按的行为

if (hasLongPressOnPowerBehavior()) {

// 2、亮屏时，长按Power键将触发MSG\_POWER\_LONG\_PRESS消息

Message msg = mHandler.obtainMessage(MSG\_POWER\_LONG\_PRESS);

msg.setAsynchronous(true);

mHandler.sendMessageDelayed(msg,

ViewConfiguration.get(mContext).getDeviceGlobalActionKeyTimeout());

}

} else {

// 此时屏幕是熄灭状态

// 3、先唤醒系统，这个会调用到PMS的wakeUp

wakeUpFromPowerKey(event.getDownTime());

// 支持熄屏长按，mSupportLongPressPowerWhenNonInteractive读资源文件得到，默认为false

if (mSupportLongPressPowerWhenNonInteractive && hasLongPressOnPowerBehavior()) {

Message msg = mHandler.obtainMessage(MSG\_POWER\_LONG\_PRESS);

msg.setAsynchronous(true);

mHandler.sendMessageDelayed(msg,

ViewConfiguration.get(mContext).getDeviceGlobalActionKeyTimeout());

mBeganFromNonInteractive = true;

} else {

// 默认返回1

final int maxCount = getMaxMultiPressPowerCount();

if (maxCount <= 1) {

// 息屏时，按下power键(不弹起)，仅消耗掉该事件

mPowerKeyHandled = true;

} else {

mBeganFromNonInteractive = true;

}

}

}

}

}

4.1.1 判断是否支持长按的行为

hasLongPressOnPowerBehavior负责判断终端是否支持长按的行为：

private boolean hasLongPressOnPowerBehavior() {

return getResolvedLongPressOnPowerBehavior() != LONG\_PRESS\_POWER\_NOTHING;

}

private int getResolvedLongPressOnPowerBehavior() {

//取决与系统属性"factory.long\_press\_power\_off"，此处默认为false

if (FactoryTest.isLongPressOnPowerOffEnabled()) {

return LONG\_PRESS\_POWER\_SHUT\_OFF\_NO\_CONFIRM;

}

return mLongPressOnPowerBehavior;

}

从上面的代码可以看出，终端是否支持长按行为，最终将由mLongPressOnPowerBehavior决定。

......

mLongPressOnPowerBehavior = mContext.getResources().getInteger(com.android.internal.R.integer.config\_longPressOnPowerBehavior);

......

mLongPressOnPowerBehavior将在PhoneWindowManager初始化时，通过读取资源文件得到，一般情况下应该为1。于是，hasLongPressOnPowerBehavior的值返回true，即终端支持Power键长按。

4.1.2 亮屏时，长按Power键将触发MSG\_POWER\_LONG\_PRESS消息

从上面的代码，我们知道亮屏时按Power键，会触发延迟的MSG\_POWER\_LONG\_PRESS消息。

如果在MSG\_POWER\_LONG\_PRESS超时前，Power键未被释放掉，那么此次操作被定义为长按Power键。

MSG\_POWER\_LONG\_PRESS对应的处理函数为powerLongPress

private void powerLongPress() {

//也是由资源文件得到，默认为1，即LONG\_PRESS\_POWER\_GLOBAL\_ACTIONS

final int behavior = getResolvedLongPressOnPowerBehavior();

switch (behavior) {

case LONG\_PRESS\_POWER\_NOTHING:

break;

case LONG\_PRESS\_POWER\_GLOBAL\_ACTIONS:

mPowerKeyHandled = true;

//感觉这里是：终端对接收的事件处理后，给用户一个反馈信息

//performHapticFeedbackLw主要进行震动反馈，例如按键后，终端震动一下

//不同的事件，定义了不同的震动模式

performHapticFeedbackLw(null, HapticFeedbackConstants.LONG\_PRESS, false);

//弹出选择关机还是重启的对话框

showGlobalActionsInternal();

break;

case LONG\_PRESS\_POWER\_SHUT\_OFF:

case LONG\_PRESS\_POWER\_SHUT\_OFF\_NO\_CONFIRM:

mPowerKeyHandled = true;

performHapticFeedbackLw(null, HapticFeedbackConstants.LONG\_PRESS, false);

sendCloseSystemWindows(SYSTEM\_DIALOG\_REASON\_GLOBAL\_ACTIONS);

mWindowManagerFuncs.shutdown(behavior == LONG\_PRESS\_POWER\_SHUT\_OFF);

break;

}

}

4.1.3 在息屏的状态下按下Power键

在息屏的状态下按下Power键，将调用wakeUpFromPowerKey函数唤醒系统：

private void wakeUpFromPowerKey(long eventTime) {

//从config.xml来看，mAllowTheaterModeWakeFromPowerKey默认为true

wakeUp(eventTime, mAllowTheaterModeWakeFromPowerKey, "android.policy:POWER");

}

private boolean wakeUp(long wakeTime, boolean wakeInTheaterMode, String reason) {

//取数据库的值

final boolean theaterModeEnabled = isTheaterModeEnabled();

//按Power键时，条件返回false

if (!wakeInTheaterMode && theaterModeEnabled) {

return false;

}

//看来唤醒时，将退出剧院模式

if (theaterModeEnabled) {

Settings.Global.putInt(mContext.getContentResolver(), Settings.Global.THEATER\_MODE\_ON, 0);

}

//最终将调用到PMS的wakeUp函数

mPowerManager.wakeUp(wakeTime, reason);

return true;

}

我们跟进一下PMS的wakeUp函数：

public void wakeUp(long eventTime, String reason, String opPackageName) {

..........

try {

wakeUpInternal(eventTime, reason, uid, opPackageName, uid);

} finally {

..........

}

}

private void wakeUpInternal(long eventTime, String reason, int uid, String opPackageName, int opUid) {

synchronized (mLock) {

//更新Wakefullness的状态为WAKEFULNESS\_AWAKE，记录一次UserActivity

if (wakeUpNoUpdateLocked(eventTime, reason, uid, opPackageName, opUid)) {

updatePowerStateLocked(); //如之前博客所述，对整个电源状态进行一次调整，将在需要时点亮屏幕

}

}

}

**4.2 处理松开Power键 - interceptPowerKeyUp**

interceptPowerKeyUp处理松开Power键后的流程：

private void interceptPowerKeyUp(KeyEvent event, boolean interactive, boolean canceled) {

//事件被取消，或者在按下Power键时，该事件已被消耗掉，那么就不用继续处理

final boolean handled = canceled || mPowerKeyHandled;

mScreenshotChordPowerKeyTriggered = false;

//退出截屏

cancelPendingScreenshotChordAction();

//取消MSG\_POWER\_LONG\_PRESS事件，即在一定时间内Power键弹起，则表示这一次不是长按Power键

cancelPendingPowerKeyAction();//mHandler.removeMessages(MSG\_POWER\_LONG\_PRESS);

//从之前的代码，我们知道除了特殊功能外，灭屏按Power键或亮屏长按时，均会消耗掉Power事件，因此，只有亮屏短按Power键需要进行处理

if (!handled) {

// Figure out how to handle the key now that it has been released.

// 记录短按的次数

mPowerKeyPressCounter += 1;

final int maxCount = getMaxMultiPressPowerCount();

final long eventTime = event.getDownTime();

if (mPowerKeyPressCounter < maxCount) {

// This could be a multi-press. Wait a little bit longer to confirm. Continue holding the wake lock.

// 与之前interceptPowerKeyDown，处理Power键被多次按下场景对应，每次被按下，均发送MSG\_POWER\_DELAYED\_PRESS消息

// 实际上maxCount为1，不会进入该分支

Message msg = mHandler.obtainMessage(MSG\_POWER\_DELAYED\_PRESS,

interactive ? 1 : 0, mPowerKeyPressCounter, eventTime);

msg.setAsynchronous(true);

mHandler.sendMessageDelayed(msg, ViewConfiguration.getMultiPressTimeout());

return;

}

// 1、No other actions. Handle it immediately.

powerPress(eventTime, interactive, mPowerKeyPressCounter);

}

// 2、Done. Reset our state.

finishPowerKeyPress();

}

4.2.1 powerPress

我们跟进一下powerPress函数：

private void powerPress(long eventTime, boolean interactive, int count) {

if (mScreenOnEarly && !mScreenOnFully) {

Slog.i(TAG, "Suppressed redundant power key press while "

+ "already in the process of turning the screen on.");

return;

}

if (count == 2) {

//原生不进入

powerMultiPressAction(eventTime, interactive, mDoublePressOnPowerBehavior);

} else if (count == 3) {

//原生不进入

powerMultiPressAction(eventTime, interactive, mTriplePressOnPowerBehavior);

} else if (interactive && !mBeganFromNonInteractive) {

//亮屏时，将进入该分支

//mShortPressOnPowerBehavior被配置为1

switch (mShortPressOnPowerBehavior) {

case SHORT\_PRESS\_POWER\_NOTHING:

break;

case SHORT\_PRESS\_POWER\_GO\_TO\_SLEEP:

//最终调用到PMS的goToSleep函数

goToSleep(eventTime, PowerManager.GO\_TO\_SLEEP\_REASON\_POWER\_BUTTON, 0);

break;

.........

}

}

}

从上面的代码可以看出，在亮屏状态下，短按一下Power键，最终将调用到PMS的goToSleep函数，使终端进入到休眠状态，与实际情况一致。

我们跟进一下PMS的goToSleep函数：

public void goToSleep(long eventTime, int reason, int flags) {

.........

try {

goToSleepInternal(eventTime, reason, flags, uid);

} finally {

.........

}

}

private void goToSleepInternal(long eventTime, int reason, int flags, int uid) {

synchronized (mLock) {

//没有触发用户事件，将mWakefullness置为WAKEFULNESS\_DOZING

if (goToSleepNoUpdateLocked(eventTime, reason, flags, uid)) {

//执行整体的电源状态更新，将熄灭屏幕

updatePowerStateLocked();

}

}

}

4.2.2 finishPowerKeyPress

每当处理一次完整的Power键按下、弹出操作后，interceptPowerKeyUp调用finishPowerKeyPress进行最后的状态复位操作：

private void finishPowerKeyPress() {

mBeganFromNonInteractive = false;

mPowerKeyPressCounter = 0;

if (mPowerKeyWakeLock.isHeld()) {

mPowerKeyWakeLock.release();

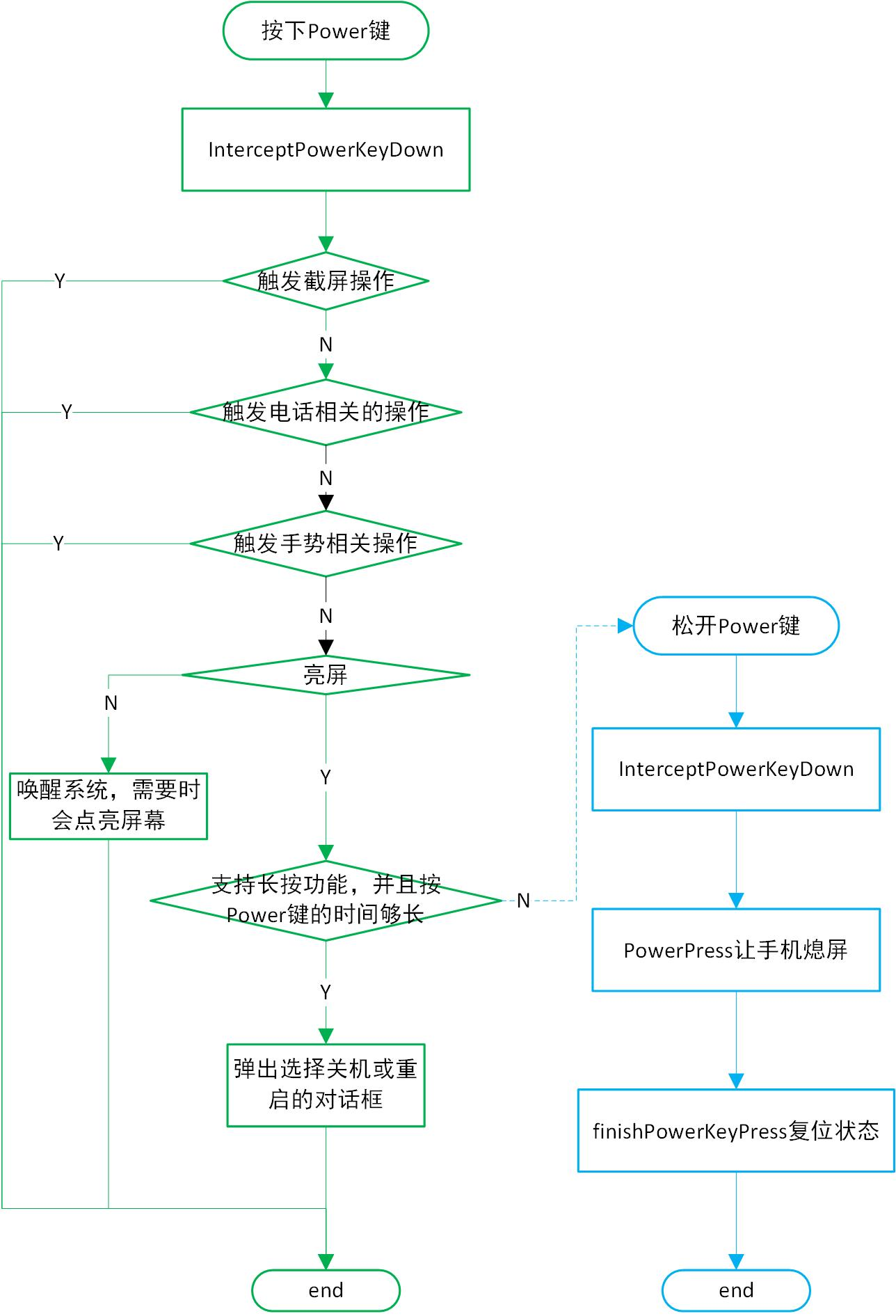
}

}

从代码可以看出，主要的工作其实就是将状态变量恢复为初始值，同时释放掉最初申请的锁。

**4.3 小结**

整个Power按键的主要处理流程如下图所示。结合手机的实际情况，进行整体逻辑的理解。



**5 PMS之userActivity**

用户活动，顾名思义，当有对手机操作时，该方法会被调用。

如果我们在Settings中设置sleep时间为15s，那么15秒内如果没有任何操作，屏幕就会熄灭(当然，没有WakeLock未被释放是前提)。如果在这个时间内用户有操作：touch屏幕或者按下菜单键、返回键等，那么这时就会调用PowerManagerService的UserActivity方法，这样会重置那个timeout的时间。

// Message: Sent when a user activity timeout occurs to update the power state.

**private static final int** MSG\_USER\_ACTIVITY\_TIMEOUT = 1;

**private final class BinderService extends** IPowerManager.Stub {

@Override // Binder call

**public void userActivity**(**long** eventTime, **int** event, **int** flags) {

**final long** now = **SystemClock**.uptimeMillis();//从开机到现在的毫秒数(手机睡眠的时间不包括在内)；

**if** (mContext.checkCallingOrSelfPermission(**android.Manifest.permission**.DEVICE\_POWER)//检查上下文是否允许访问底层PM

!= PackageManager.PERMISSION\_GRANTED

&& mContext.checkCallingOrSelfPermission(android.Manifest.permission.USER\_ACTIVITY)//是否具有userActivity权限

!= PackageManager.PERMISSION\_GRANTED) {

**synchronized** (mLock) {

//权限没有通过打印log后，返回

}

**return**;

}

**if** (eventTime > now) {//确保这次事件的发生不应该是将来发生的，否则抛出异常

**throw new** IllegalArgumentException("event time must not be in the future");

}

//A通过IPC调用B，然后B又调用自己的接口所以需要将当前的UID、PID设置为B的，调用完毕后再恢复成A的

**final int** uid = **Binder**.getCallingUid();//获得调用者A的UID

**final long** ident = **Binder**.clearCallingIdentity();//清空调用者A的UID，使用服务端B的UID、PID

**try** {

**userActivityInternal**(eventTime, event, flags, uid);//注意这个uid是A的UID

} **finally** {

**Binder**.restoreCallingIdentity(ident);//恢复调用者A的UID和PID

}

}

}

**private void userActivityInternal**(**long** eventTime, **int** event, **int** flags, **int** uid) {

**synchronized** (mLock) {

**if** (**userActivityNoUpdateLocked**(eventTime, event, flags, uid)) {

updatePowerStateLocked();//参见关于updatePowerStateLocked 章节

}

}

}

userActivityNoUpdateLocked(...)触发一次用户活动，更新mDirty和最近一次userActivity的时间

**private boolean** **userActivityNoUpdateLocked**(**long** eventTime, **int** event, **int** flags, **int** uid) {

**if** (DEBUG\_SPEW) {

**Slog**.d(TAG, "userActivityNoUpdateLocked: eventTime=" + eventTime

+ ", event=" + event + ", flags=0x" + **Integer**.toHexString(flags) + ", uid=" + uid);

}

**if** (eventTime < mLastSleepTime || eventTime < mLastWakeTime || !mBootCompleted || !mSystemReady) {//时间检查等

**return false**;

}

Trace.traceBegin(Trace.TRACE\_TAG\_POWER, "userActivity");//systrace

**try** {

**if** (eventTime > mLastInteractivePowerHintTime) {

powerHintInternal(PowerHint.INTERACTION, 0);//提频操作device/qcom/common/power/power-845.c

mLastInteractivePowerHintTime = eventTime;//记录本次eventTime

}

mNotifier.onUserActivity(event, uid);//通过Notifier发送通知、广播

**if** (mUserInactiveOverrideFromWindowManager) {//来自WindowManager的用户交互覆盖？，这个问题先记着

mUserInactiveOverrideFromWindowManager = **false**;

mOverriddenTimeout = -1;

}

**if** (mWakefulness == WAKEFULNESS\_ASLEEP//表示系统当前处于休眠状态，只能被wakeUp()调用唤醒。

|| mWakefulness == WAKEFULNESS\_DOZING//doze状态

|| (flags & PowerManager.USER\_ACTIVITY\_FLAG\_INDIRECT) != 0) {//判断flag

**return false**;//直接返回false，这样power state不会更新

}

**if** ((flags & PowerManager.USER\_ACTIVITY\_FLAG\_NO\_CHANGE\_LIGHTS) != 0) {//记录相应最近发生的时间点和状态变化mDirty

**if** (eventTime > mLastUserActivityTimeNoChangeLights && eventTime > mLastUserActivityTime) {

mLastUserActivityTimeNoChangeLights = eventTime;

mDirty |= DIRTY\_USER\_ACTIVITY;

**if** (event == PowerManager.USER\_ACTIVITY\_EVENT\_BUTTON) {

mDirty |= DIRTY\_QUIESCENT;

}

**return true**;

}

} **else** {

**if** (eventTime > mLastUserActivityTime) {

mLastUserActivityTime = eventTime;

mDirty |= DIRTY\_USER\_ACTIVITY;//用全局变量mDirty来记录状态变化

**if** (event == PowerManager.USER\_ACTIVITY\_EVENT\_BUTTON) {

mDirty |= DIRTY\_QUIESCENT;//静止状态发生改变

}

**return true**;

}

}

} **finally** {

Trace.traceEnd(Trace.TRACE\_TAG\_POWER);//systrace结束

}

**return false**;

}

参数int event

frameworks/base/core/java/android/os/PowerManager.java

public static final int USER\_ACTIVITY\_EVENT\_OTHER = 0; //BackupManagerService代码里有使用

public static final int USER\_ACTIVITY\_EVENT\_BUTTON = 1; //button事件

public static final int USER\_ACTIVITY\_EVENT\_TOUCH = 2; //touch事件

public static final int USER\_ACTIVITY\_EVENT\_ACCESSIBILITY = 3; //AccessibilityManagerServicev代码里有使用，比如跟View相关的监听

参数int flags

frameworks/base/core/java/android/os/PowerManager.java

public static final int USER\_ACTIVITY\_FLAG\_NO\_CHANGE\_LIGHTS = 1 << 0;//如果已经是dim亮度，则延长dim亮度的timeout时间且不让屏幕更亮。该标记可以用于仅仅需要再延长一下亮屏时间而不要更改亮度，比如按下power键虽然延长了亮度，但是会让亮度变化

public static final int USER\_ACTIVITY\_FLAG\_INDIRECT = 1 << 1;//照常记录用户活动，但不重置用户活动超时。该标记可以用于当手机跟辅助屏幕交互，允许主屏幕休眠