Android (/tags/#Android) PowerManager (/tags/#PowerManager) Wakelock (/tags/#Wakelock)

Android电源管理之申请电源锁

介绍Android系统中电源锁的基础概念和申请流程

Posted by Cheson on February 23, 2017

适用平台

Android Version： 6.0 Platform： MTK6580/MTK6735/MTK6753

1. 基础介绍

Android新加入的WakeLock是一种锁的机制,只要拿着这个锁,系统就无法进入休眠,可以被用户态进程和内核线程获得。这个锁可以是有超时的或者是没有超时的,超时的锁会在时间过去以后自动解锁。如果没有锁了或者超时了, 内核就会启动标准Linux的那套休眠机制机制来进入休眠。

WakeLock机制是Android电源管理中的重要部分，本文将根据自上而下的顺序来介绍获取电源锁的获取流程，基于Android6.0代码。

2. APP层使用

在一个App中想要使用电源锁来hold住背光或者是CPU，那么可以这么做。1）获取PowerManager接口：pm = (PowerManager) getSystemService(Context.POWER_SERVICE)；2）新建电源锁：PowerManager.WakeLock wakelock = pm.newWakeLock(wakelock的类型，wakelock的tag)；3）获取电源锁：wakelock.acquire()；4）在不使用的时候释放电源锁：wakelock.release()；5）在AndroidManifest中加入权限

根据以上几步就可以在一个app应用中取得电源锁了，在使用电源锁中需要注意以下几点：

1）获取的电源锁类型，也就是newWakelock方法中的第一个参数。电源锁类型需要根据应用自身需求和特点来申请，具体的类型定义在PowerManager中：

PARTIAL_WAKE_LOCK—-保证CPU运行，屏幕和键盘灯允许关闭。用户按power键之后，屏幕和键盘灯会关闭，CPU keep on，直到所有该类型所被释放

SCREEN_DIM_WAKE_LOCK—-保证屏幕亮（可能是dim状态），键盘灯允许关闭。用户按power键之后，CPU和屏幕都会被关闭

SCREEN_BRIGHT_WAKE_LOCK—-保证屏幕亮（at full brightness），键盘灯允许关闭。用户按power键之后，CPU和屏幕都会被关闭

FULL_WAKE_LOCK—-保证屏幕和键盘灯亮（at full brightness）。用户按power键之后，CPU和屏幕键盘灯都会被关闭

PROXIMITY_SCREEN_OFF_WAKE_LOCK—-pSensor导致的灭屏情况下系统不会进入休眠，正常情况下不影响系统休眠

DOZE_WAKE_LOCK—-使屏幕进入low power状态，允许cpu挂起。只有在电源管理进入doze模式时生效

DRAW_WAKE_LOCK—-保持设备awake状态已完成绘制事件，只在doze模式下生效。

2）电源锁从获取方式来看，可以分为超时锁和非超时锁，申请时调用的方法略有不同，非超时锁是直接调用wakelock的acquire()方法，这类锁只有在调用release之后才会被释放；超时锁是调用wakelock的acquire(long timeout)来获取，超时锁申请之后会postDelay一个release的runnable，在超时之后自动释放该电源锁。

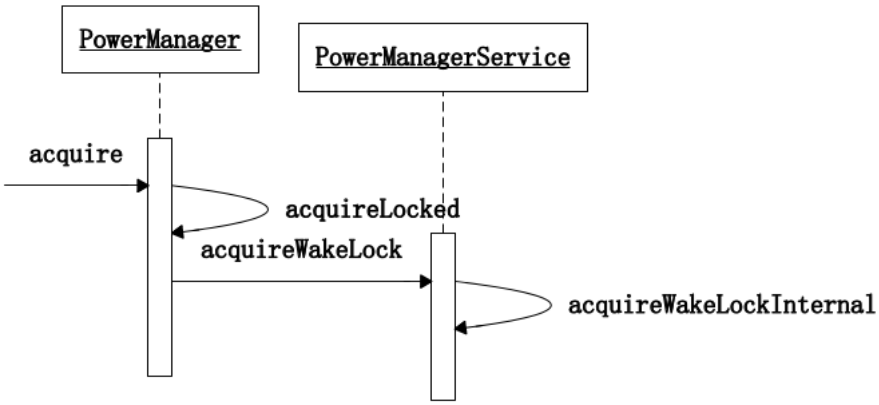
3）注意在退出应用或者关闭功能时一定要释放电源锁，否则可能会导致系统无法进入休眠而导致待机电流过高的现象出现。

3. Frameworks流程

上层申请了电源锁之后，系统是如何对其进行处理的呢？接下来看Frameworks中是如何管理wakelock的。

3.1 acquire流程

电源锁是由Frameworks中的电源管理模块来管理的。在App中是通过PowerManager接口来调用acquire方法获取电源锁，这个方法之后的流程为：



App调用PowerManager接口的acquire方法，PowerManager调用自身的acquireLocked，然后通过binder调用PowerManagerService的acquireWakeLock方法，在PMS中再继续调用acquireWakelockInternal，后面具体的操作就在这个方法中执行了。

```
    WakeLock wakeLock;
    int index = findWakeLockIndexLocked(lock); // 根据传入的lock判断该wakelock是否已存在
    boolean notifyAcquire;
    if (index >= 0) {// 如果已存在该wakelock
        wakeLock = mWakeLocks.get(index);
        if (!wakeLock.hasSameProperties(flags, tag, ws, uid, pid)) {
            // Update existing wake lock.  This shouldn't happen but is harmless.
            notifyWakeLockChangingLocked(wakeLock, flags, tag, packageName,
                    uid, pid, ws, historyTag);
            //更新电源锁相关信息
            wakeLock.updateProperties(flags, tag, packageName, ws, historyTag, uid, pid);
        }
        notifyAcquire = false;
    } else {// 新建电源锁
        wakeLock = new WakeLock(lock, flags, tag, packageName, ws, historyTag, uid, pid);
        try {
            lock.linkToDeath(wakeLock, 0);// 如果binder挂了则回调此方法
        } catch (RemoteException ex) {
            throw new IllegalArgumentException("Wake lock is already dead.");
        }
        mWakeLocks.add(wakeLock);// 将新建的wakelock添加到list中去
        setWakeLockDisabledStateLocked(wakeLock);// 判断是否该disable该wakelock，该状态影响后续使用
        notifyAcquire = true;
        wakeLock.mActiveSince = SystemClock.uptimeMillis();
    }
        // 如果申请的wakelock为ACQUIRE_CAUSE_WAKEUP的类型，需要在这一步直接wake系统
        applyWakeLockFlagsOnAcquireLocked(wakeLock, uid);
    mDirty |= DIRTY_WAKE_LOCKS;// 将wakelock标志添加到mDirty中，在更新电源状态时表示当前持有电源锁
    updatePowerStateLocked();// 调用该方法来更新电源状态
    if (notifyAcquire) {
        // This needs to be done last so we are sure we have acquired the
        // kernel wake lock.  Otherwise we have a race where the system may
        // go to sleep between the time we start the accounting in battery
        // stats and when we actually get around to telling the kernel to
        // stay awake.
        notifyWakeLockAcquiredLocked(wakeLock);// 发送电源锁获取的通知
    }
```

上述代码注释为acquireWakelockInternal这个方法的主要工作介绍，最后进入的一个关键方法就是updatePowerStateLocked，此方法是PMS中的一个重要入口，来触发电源状态的更新，而决定电源状态是否更新，如何更新的关键就是mDirty这个标志。此标志通过抑或运算（|=），将各种状态的标志都包含进去了，后面会讲到一部分。

3.2 updatePowerStateLocked

updatePowerStateLocked方法中分成了6段来做一些事情。

Phase 0：更新一些基础状态

```
    // Phase 0: Basic state updates.
    updateIsPoweredLocked(mDirty);// 更新充电状态
    updateStayOnLocked(mDirty);// 更新唤醒状态
    updateScreenBrightnessBoostLocked(mDirty);// 先保留答案
```

Phase 1：更新系统wakefulness状态

```
    // Phase 1: Update wakefulness.
    // Loop because the wake lock and user activity computations are influenced
    // by changes in wakefulness.
    final long now = SystemClock.uptimeMillis();
    int dirtyPhase2 = 0;
    for (;;) {
        int dirtyPhase1 = mDirty;
        dirtyPhase2 |= dirtyPhase1;
        mDirty = 0;

        updateWakeLockSummaryLocked(dirtyPhase1);// 更新电源锁标志
        updateUserActivitySummaryLocked(now, dirtyPhase1);// 更新用户行为
        if (!updateWakefulnessLocked(dirtyPhase1)) {// 更新系统wakefulness
            break;
        }
    }
```

这里首先要理解下系统wakefulness指的是什么，从我个人目前的理解wakefulness指的就是系统醒着和休眠的状态，更新就是指处于这两种状态和状态切换时的一些标志和变量的变化。那为什么需要在一个死循环中执行呢？因为在某些状态下调用到updateWakefulnessLocked方法之后，会对wakelockSummary和userActivitySummary的状态产生影响，所以要再次更新一下。由于此处和电源锁关系较密切，我们来通过log看下具体过程。

```
    private boolean updateWakefulnessLocked(int dirty) {
        boolean changed = false;
        android.util.Log.v("chendongqi-power", "updateWakefulnessLocked: dirty="+dirty
                +", mWakefulness="+mWakefulness
                +", isItBedTimeYetLocked="+isItBedTimeYetLocked()
                +", shouldNapAtBedTimeLocked="+shouldNapAtBedTimeLocked());
        if ((dirty & (DIRTY_WAKE_LOCKS | DIRTY_USER_ACTIVITY | DIRTY_BOOT_COMPLETED
                | DIRTY_WAKEFULNESS | DIRTY_STAY_ON | DIRTY_PROXIMITY_POSITIVE
                | DIRTY_DOCK_STATE)) != 0) {
            // 当前处于wake状态且达到了休眠的条件（系统启动完成，不需要保持wake，不是ipo关机）
            if (mWakefulness == WAKEFULNESS_AWAKE && isItBedTimeYetLocked()) {
                if (DEBUG_SPEW) {
                    Slog.d(TAG, "updateWakefulnessLocked: Bed time...");
                }
                final long time = SystemClock.uptimeMillis();
                if (shouldNapAtBedTimeLocked()) {// 一般情况下都是false
                    changed = napNoUpdateLocked(time, Process.SYSTEM_UID);
                } else {// 进入系统休眠
                    android.util.Log.v("chendongqi-power", "changed before="+changed);
                    changed = goToSleepNoUpdateLocked(time,
                            PowerManager.GO_TO_SLEEP_REASON_TIMEOUT, 0, Process.SYSTEM_UID);
                    android.util.Log.v("chendongqi-power", "changed after="+changed);
                }
            }
        }
        return changed;
    }
```

从代码逻辑来看，只有当return changed为true的时候才会一直执行循环，那么就需要关注何时为true了。从代码中看只有进入了napNoUpdateLocked或者gotToSleepNoUpdateLocked时才有可能返回true，一般情况下走不进napNoUpdateLocked（dream状态时可能进入），那么如果在系统wake的状态下，且满足进入休眠的条件，那么在系统nap或者sleep之后就会返回true，而这些方法里同时也更新了wakefulness的状态。

```
    chendongqi-power: updateWakefulnessLocked: dirty=4, mWakefulness=1, isItBedTimeYetLocked=true
```

从log中可以看到当状态改变时会出现以上信息，表示系统进入休眠，当前为wake状态，且达到休眠条件。在处理完之后mDirty的值发生了变化，所以要重新更新下wakelock和userActivity的状态。

Phase 2：更新显示状态

```
    // Phase 2: Update display power state.
    // 更新显示状态，处理背光设置相关逻辑，设置亮度，自动调节，pSensor灭屏等
    boolean displayBecameReady = updateDisplayPowerStateLocked(dirtyPhase2);
```

需要调查和背光相关流程时可以关注这个方法。

Phase 3：更新dream相关状态（暂未关注）

```
    // Phase 3: Update dream state (depends on display ready signal).
    updateDreamLocked(dirtyPhase2, displayBecameReady);
```

Phase 4：当wakefulness状态更新完时发送通知

```
// Phase 4: Send notifications, if needed.
finishWakefulnessChangeIfNeededLocked();
```

Phase 5：更新阻止系统挂起的状态

```
// Phase 5: Update suspend blocker.
// Because we might release the last suspend blocker here, we need to make sure
// we finished everything else first!
updateSuspendBlockerLocked();
```

在这个方法中和电源锁密切相关的两个方法就是Phase 1中的updateWakeLockSummaryLocked和Phase 5中的updateSuspendBlockerLocked，一个个来看。

3.3 updateWakeLockSummaryLocked

```
    private void updateWakeLockSummaryLocked(int dirty) {
        // mDirty中表示需要更新wakelock或者wakefulness时才进入
        if ((dirty & (DIRTY_WAKE_LOCKS | DIRTY_WAKEFULNESS)) != 0) {
            mWakeLockSummary = 0;// 初始化mWakeLockSummary

            final int numWakeLocks = mWakeLocks.size();
            for (int i = 0; i < numWakeLocks; i++) {// 遍历mWakeLocks列表中的所有wakelock
                final WakeLock wakeLock = mWakeLocks.get(i);
                switch (wakeLock.mFlags & PowerManager.WAKE_LOCK_LEVEL_MASK) {// 取得wakelock类型
                    case PowerManager.PARTIAL_WAKE_LOCK:
                        if (!wakeLock.mDisabled) {// 这里判断就用到了前面设置的disable
                            // We only respect this if the wake lock is not disabled.
                            mWakeLockSummary |= WAKE_LOCK_CPU;// 非disable的PARTIAL_WAKE_LOCK被加入锁cpu的标志
                        }
                        break;
                    case PowerManager.FULL_WAKE_LOCK:// 加入锁屏幕和按键背光的标志
                        mWakeLockSummary |= WAKE_LOCK_SCREEN_BRIGHT | WAKE_LOCK_BUTTON_BRIGHT;
                        break;
                    case PowerManager.SCREEN_BRIGHT_WAKE_LOCK:// 加入锁屏幕背光的标志
                        mWakeLockSummary |= WAKE_LOCK_SCREEN_BRIGHT;
                        break;
                    case PowerManager.SCREEN_DIM_WAKE_LOCK:// 加入锁屏幕背光dim的标志
                        mWakeLockSummary |= WAKE_LOCK_SCREEN_DIM;
                        break;
                    case PowerManager.PROXIMITY_SCREEN_OFF_WAKE_LOCK:// 加入pSensor灭屏锁标志
                        mWakeLockSummary |= WAKE_LOCK_PROXIMITY_SCREEN_OFF;
                        break;
                    case PowerManager.DOZE_WAKE_LOCK:// 加入系统doze唤醒标记
                        mWakeLockSummary |= WAKE_LOCK_DOZE;
                        break;
                    case PowerManager.DRAW_WAKE_LOCK:// 加入Dream下绘制界面的标记
                        mWakeLockSummary |= WAKE_LOCK_DRAW;
                        break;
                }
            }

            // Cancel wake locks that make no sense based on the current state.
            if (mWakefulness != WAKEFULNESS_DOZING) {// 如果系统不是处于idle状态，则去掉没用的锁
                mWakeLockSummary &= ~(WAKE_LOCK_DOZE | WAKE_LOCK_DRAW);
            }
            // 如果系统处于休眠，或者mWakeLockSummary中还有doze的标记（经过上一步处理可以判断此时处于doze）
            // 则去除所有锁背光的标记
            if (mWakefulness == WAKEFULNESS_ASLEEP
                    || (mWakeLockSummary & WAKE_LOCK_DOZE) != 0) {
                mWakeLockSummary &= ~(WAKE_LOCK_SCREEN_BRIGHT | WAKE_LOCK_SCREEN_DIM
                        | WAKE_LOCK_BUTTON_BRIGHT);
                if (mWakefulness == WAKEFULNESS_ASLEEP) {
                    /* Power Key Force Wakeup to Keep Proximity Wakelock */
                    // mWakeLockSummary &= ~WAKE_LOCK_PROXIMITY_SCREEN_OFF;
                    //<2015/9/11-Lianxi,[D5][PORTING][COMMON][CALL/SS_USSD][CQ][NJSCM00055202]Fix the power key can work when enable
                    //<2015/04/22-georgelin, [V10][Bug][Common][Call/SS_USSD][CQ][NJSCM00055202]Fix the power key can work when enabl
                    mWakeLockSummary &= ~WAKE_LOCK_PROXIMITY_SCREEN_OFF;
                    //>2015/04/22-georgelin
                    //>2015/9/11-Lianxi
                }
            }

            // Infer implied wake locks where necessary based on the current state.
            // 当前状态下有锁屏幕背光的标记
            if ((mWakeLockSummary & (WAKE_LOCK_SCREEN_BRIGHT | WAKE_LOCK_SCREEN_DIM)) != 0) {
                if (mWakefulness == WAKEFULNESS_AWAKE) {
                    // 如果当前系统处于awake，则加入锁cpu和保持awake的标记
                    mWakeLockSummary |= WAKE_LOCK_CPU | WAKE_LOCK_STAY_AWAKE;
                } else if (mWakefulness == WAKEFULNESS_DREAMING) {
                    // 如果处于dream，则加入锁cpu的标记
                    mWakeLockSummary |= WAKE_LOCK_CPU;
                }
            }
            // 需要保持系统awake状态
            if ((mWakeLockSummary & WAKE_LOCK_DRAW) != 0) {
                mWakeLockSummary |= WAKE_LOCK_CPU;
            }

            if (DEBUG_SPEW) {
                Slog.d(TAG, "updateWakeLockSummaryLocked: mWakefulness="
                        + PowerManagerInternal.wakefulnessToString(mWakefulness)
                        + ", mWakeLockSummary=0x" + Integer.toHexString(mWakeLockSummary));
            }
        }
    }
```

在这个方法中，各个部分的代码解析参照上图，其中起到重要作用的就是一个全局变量mWakeLockSummary，根据当前系统状态和申请的电源锁的状态，来更新这个变量的标志，在后续使用。那么mWakeLockSummary这个变量是如何被使用的呢？

1. 决定pSensor亮灭屏的功能

```
private boolean shouldUseProximitySensorLocked() {
    return (mWakeLockSummary & WAKE_LOCK_PROXIMITY_SCREEN_OFF) != 0;
}
```

在通话时会申请WAKE_LOCK_PROXIMITY_SCREEN_OFF电源锁，具体可以参照我博客中的另一篇文章 Android电源管理之通话中的pSensor工作原理分析 (https://chendongqi.github.io/blog/2017/02/23/pm_pSensor_in_call/)，在PMS的shouldUseProximitySensorLocked方法中就是判断mWakeLockSummary变量中是否有WAKE_LOCK_PROXIMITY_SCREEN_OFF来决定pSensor是否处于可用状态。

2. 决定按键背光

```
if ( ( (mWakeLockSummary & WAKE_LOCK_BUTTON_BRIGHT) != 0 ) ||
        ( (mUserActivitySummary & USER_ACTIVITY_BUTTON_BRIGHT) != 0) ) {
    mButtonLight.setBrightness(screenBrightness);
    Slog.i(TAG, "setBrightness mButtonLight, screenBrightness=" + screenBrightness);
} else {
    mButtonLight.turnOff();
    Slog.i(TAG, "setBrightness mButtonLight 0.");
}
```

在updateDisplayPowerStateLocked方法中，会mWakeLockSummary中是否包含WAKE_LOCK_BUTTON_BRIGHT标志会成为决定点亮按键背光的一个条件。

3. 决定屏幕背光状态

```
private int getDesiredScreenPolicyLocked() {
    if ((mWakefulness == WAKEFULNESS_ASLEEP) || mShutdownFlag) {
        return DisplayPowerRequest.POLICY_OFF;
    }

    if (mWakefulness == WAKEFULNESS_DOZING) {
        if ((mWakeLockSummary & WAKE_LOCK_DOZE) != 0) {
            return DisplayPowerRequest.POLICY_DOZE;
        }
        if (mDozeAfterScreenOffConfig) {
            return DisplayPowerRequest.POLICY_OFF;
        }
        // Fall through and preserve the current screen policy if not configured to
        // doze after screen off.  This causes the screen off transition to be skipped.
    }

    if ((mWakeLockSummary & WAKE_LOCK_SCREEN_BRIGHT) != 0
            || (mUserActivitySummary & USER_ACTIVITY_SCREEN_BRIGHT) != 0
            || !mBootCompleted
            || mScreenBrightnessBoostInProgress
            || bypassUserActivityTimeout()) {
        return DisplayPowerRequest.POLICY_BRIGHT;
    }

    return DisplayPowerRequest.POLICY_DIM;
}
```

调用getDesiredScreenPolicyLocked方法来决定屏幕背光的设置策略，在这里就根据mWakeLockSummary中的不同位，来判断是要设置中OFF、DOZE、BRIGHT或者是DIM。这个方法在updateDisplayPowerStateLocked方法开始时被调用来设置mDisplayPowerRequest.policy的值。

4. 定系统是否能进入suspend状态

```
private void updateSuspendBlockerLocked() {
    final boolean needWakeLockSuspendBlocker = ((mWakeLockSummary & WAKE_LOCK_CPU) != 0);
    final boolean needDisplaySuspendBlocker = needDisplaySuspendBlockerLocked();
    final boolean autoSuspend = !needDisplaySuspendBlocker;
    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) {
        setHalAutoSuspendModeLocked(false);
    }

    // First acquire suspend blockers if needed.
    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);
    }
}
```

这里就要涉及到前面提到的updatePowerStateLocked方法的phase 5和电源锁密切相关的的重要方法updateSuspendBlockerLocked。这个方法可以作为说是承上启下的，它将流程从framework带入到了native，具体流程是这样走的。在这个方法中核心的内容就是两个SuspendBlocker，mWakeLockSuspendBlocker和mDisplaySuspendBlocker，SuspendBlocker是一个接口，其实现为PowerManagerService的内部类SuspendBlockerImpl，主要为acquire方法和release方法。SuspendBlocker从字面上看可以理解为使系统不能进入suspend状态的阻断者。那么为什么要定义两个不同名字的Blocker呢？从mWakeLockSuspendBlocker和mDisplaySuspendBlocker创建到发挥作用的过程来看，在创建时，传入的名字不同，前者是PowerManagerService.WakeLocks，后者是PowerManagerService.Display，结合前面所介绍过的内容可以理解，前者表示因为持有相关的wakelock而导致CPU无法休眠，后者是因为屏幕没有休眠的原因而需要阻止系统休眠。从结果上来看两者调用acquire之后效果都是一样的，就是阻止系统进入suspend。那么唯一的作用就是用来区别block的来源了。

接下来看一下是如何阻止系统Suspend的，从SuspendBlockerImpl的acquire开始。

```
public void acquire() {
    synchronized (this) {
        mReferenceCount += 1;// 标示blocker的数量
        if (mReferenceCount == 1) {// 之前没有blocker的情况下acquire，否则没有必要
            if (DEBUG_SPEW) {
                Slog.d(TAG, "Acquiring suspend blocker \"" + mName + "\".");
            }
            Trace.asyncTraceBegin(Trace.TRACE_TAG_POWER, mTraceName, 0);
            nativeAcquireSuspendBlocker(mName);
        }
    }
}
```

这里需要理解mReferenceCount的作用，先做自增操作，再判断是否为1，这个逻辑就用来检测如果之前为0（没有任何blocker），那么就需要调用该blocker的acquire来使得阻止系统suspend；如果之前已经有blocker了，那么再acquire一次没有意义，因为其实所有blocker的功能其实对于kernel来说都是一样的，区别就在于传入的name不同而已。

然后就开始进入native层了。

4. native流程

续上面流程，调用nativeAcquireSuspendBlocker使得流程进入了native，nativeAcquireSuspendBlocker方法是在com_android_server_power_PowerManagerService.cpp中实现的。

```
static void nativeAcquireSuspendBlocker(JNIEnv *env, jclass /* clazz */, jstring nameStr) {
    ScopedUtfChars name(env, nameStr);
    acquire_wake_lock(PARTIAL_WAKE_LOCK, name.c_str());
}
```

这个方法通过调用acquire_wake_lock使流程进入HAL层，相关的代码为hardware/libhardware_legacy/power/power.c。从这里能看到acquire_wake_lock传入的两个参数，第一个为PARTIAL_WAKE_LOCK，这个变量不是PowerManager中定义的那个PARTIAL_WAKE_LOCK，而是power.h中定义的。第二个则为blocker的名字。

```
enum {
    PARTIAL_WAKE_LOCK = 1,  // the cpu stays on, but the screen is off
    FULL_WAKE_LOCK = 2      // the screen is also on
};
```

可以和PowerManager中定义的其实是相等的，这里看到这个枚举类型中定义了两种wakelock的值，但是遍寻代码中的acquire_wake_lock方法，传入的参数都是PARTIAL_WAKE_LOCK。所以上面也说不管是由于上层wakelock的原因还是display的原因，到acquire_wake_lock这里之后都归一了，到HAL层之后表现的效果都是一样的。

5. HAL流程

HAL层的代码集中在hardware/libhardware_legacy/power/power.c中。接native层的代码，会调用的power.c的acquire_wake_lock。

```
int
acquire_wake_lock(int lock, const char* id)
{
    initialize_fds();// 初始化设备节点

    //    ALOGI("acquire_wake_lock lock=%d id='%s'\n", lock, id);

    if (g_error) return -g_error;

    int fd;

    if (lock == PARTIAL_WAKE_LOCK) {
        // 获取打开的设备节点 /sys/power/wake_lock
        fd = g_fds[ACQUIRE_PARTIAL_WAKE_LOCK];
    }
    else {
        return -EINVAL;
    }
    // 往设备节点写入wakelock名称
    return write(fd, id, strlen(id));
}
```

来看下这个函数中所作的事情：1）初始化设备节点

```
static inline void
initialize_fds(void)
{
    // XXX: should be this:
    //pthread_once(&g_initialized, open_file_descriptors);
    // XXX: not this:
    if (g_initialized == 0) {
        if(open_file_descriptors(NEW_PATHS) < 0)
            open_file_descriptors(OLD_PATHS);
        g_initialized = 1;
    }
}
```

通过open_file_descriptors函数来打开设备节点，设备节点定义在两个数组中，NEW_PATHS和OLD_PATHS。Android6.0代码中用到了NEW_PATHS。

```
const char * const OLD_PATHS[] = {
    "/sys/android_power/acquire_partial_wake_lock",
    "/sys/android_power/release_wake_lock",
};

const char * const NEW_PATHS[] = {
    "/sys/power/wake_lock",
    "/sys/power/wake_unlock",
};
```

```
static int
open_file_descriptors(const char * const paths[])
{
    int i;
    for (i=0; i<OUR_FD_COUNT; i++) {
        int fd = open(paths[i], O_RDWR | O_CLOEXEC);
        if (fd < 0) {
            fprintf(stderr, "fatal error opening \"%s\"\n", paths[i]);
            g_error = errno;
            return -1;
        }
        g_fds[i] = fd;
    }

    g_error = 0;
    return 0;
}
```

打开之后把设备描述符fd放入在g_fds数组中，这个数组的大小定义为g_fds[OUR_FD_COUNT]，而OUR_FD_COUNT的数值定义为2，也就是两个设备节点wake_lock和wake_unlock。

```
enum {
    ACQUIRE_PARTIAL_WAKE_LOCK = 0,
    RELEASE_WAKE_LOCK,
    OUR_FD_COUNT
};
```

初始化结束之后然后判断如果lock为PARTIAL_WAKE_LOCK（其实6.0的代码在acquire时也就这一种参数），则从g_fds数组中取得wake_lock的设备描述符，然后往该设备节点中写入SuspendBlocker的名称。

以上就是acquire电源锁时，HAL层对SuspendBlocker的处理。我们再来实际读取下这个设备节点的信息验证一下。

当系统正常suspend时，查看wake_lock节点的值发现是空的。

```
root@TECNO_W4:/sys/power # cat wake_lock

root@TECNO_W4:/sys/power # cat wake_lock

root@TECNO_W4:/sys/power # cat wake_lock
```

当屏幕亮着，没有操作任何应用时，可以看到SuspendBlocker是display

```
root@TECNO_W4:/sys/power # cat wake_lock
PowerManagerService.Display
root@TECNO_W4:/sys/power # cat wake_lock
PowerManagerService.Display
root@TECNO_W4:/sys/power # cat wake_lock
PowerManagerService.Display
```

当屏幕亮着，进行一些操作时，可以看到SuspendBlocker有display和WakeLocks

```
root@TECNO_W4:/sys/power # cat wake_lock
PowerManagerService.Display PowerManagerService.WakeLocks
```

在灭屏或者亮屏的过程中，还会出现一个名为BroadCast的SuspendBlocker，这种SuspendBlocker为PMS发送通知中（例如亮灭屏通知）所需要的blocker，必须保证通知结束之后才能够进入休眠，另外在PMS中还定义了一种WirelessChargerDetector，应该是和无线充电相关。

```
root@TECNO_W4:/sys/power # cat wake_lock
PowerManagerService.Broadcasts
```

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

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

除此之外，自己写了个测试apk，申请了一个PARTIAL_WAKE_LOCK的wakelock来阻止CPU休眠，运行apk之后灭屏，可以发现系统确实无法进入suspend了，原因就是WakeLocks。

```
root@TECNO_W4:/sys/power # cat wake_lock
PowerManagerService.WakeLocks
root@TECNO_W4:/sys/power # cat wake_lock
PowerManagerService.WakeLocks
```

6. kernel流程

续上文HAL层往wake_lock设备节点中写入SuspendBlocker的名称，然后会调用到kernel的代码。至于写设备节点和调用kernel代码的关系，这里简单介绍一点。

设备节点会在kernel中注册：

```
    #ifdef CONFIG_PM_WAKELOCKS
    static ssize_t wake_lock_show(struct kobject *kobj,
                              struct kobj_attribute *attr,
                              char *buf)
    {
        return pm_show_wakelocks(buf, true);
    }

    static ssize_t wake_lock_store(struct kobject *kobj,
                               struct kobj_attribute *attr,
                               const char *buf, size_t n)
    {
        int error = pm_wake_lock(buf);
        return error ? error : n;
    }

    power_attr(wake_lock);
```

通常一个设备节点都会注册名字，一个show的方法和store的方法用来显示和存储数据。例如这里涉及到的/sys/power/wake_lock设备节点的注册就在kernel-3.18/kernel/power/main.c文件中（/sys/power目录下的节点注册都在此处）。可以看下power_attr的注册函数，代码位于kernel-3.18/kernel/power/power.h，这是一个宏定义，一般不同目录下的设备节点注册都会用宏定义，所以这个注册的函数名称一般都不相同。

```
    #define power_attr(_name) \
    static struct kobj_attribute _name##_attr = {       \
        .attr   = {                             \
                .name = __stringify(_name),     \
                .mode = 0644,                   \
        },                                      \
        .show   = _name##_show,                 \
        .store  = _name##_store,                \
    }
```

册的时候就是注册了设备节点的name，读写权限，它的show方法和store方法。这样也就是把wake_lock这个设备节点和wake_unlock_store和wake_unlock_show方法关联起来了，在HAL层调用write往设备节点中写数据的时候就会调用wake_unlock_store函数来处理。上面解释了这个关联的产生，至于从write之后是如何调用到wake_unlock_store函数的，这个机制等后续再学习。

那么再来看wake_unlock_store函数是如何继续的？接下来调用了kernel-3.18/kernel/power/wakelock.c的pm_wake_lock函数，将申请的锁名称传递下去。

```
    static ssize_t wake_lock_store(struct kobject *kobj,
                               struct kobj_attribute *attr,
                               const char *buf, size_t n)
    {
        int error = pm_wake_lock(buf);
        return error ? error : n;
    }
```

来看下pm_wake_lock方法中的较核心的一段

```
    wl = wakelock_lookup_add(buf, len, true);
        if (IS_ERR(wl)) {
                ret = PTR_ERR(wl);
                goto out;
        }
        if (timeout_ns) {
                u64 timeout_ms = timeout_ns + NSEC_PER_MSEC - 1;

                do_div(timeout_ms, NSEC_PER_MSEC);
                __pm_wakeup_event(&wl->ws, timeout_ms);
        } else {
                __pm_stay_awake(&wl->ws);
        }

        wakelocks_lru_most_recent(wl);
```

首先是检查这个wakelock是否存在红黑二叉树上，如果没有的话就把此wakelock添加到树上，把名字短的放在左叉，名字长的放在右叉。然后把wakelock数量自增。然后保持系统处于awake状态。Kernel部分详细逻辑未理解，待后续学习。

7. 后续学习方向

获取wakelock的kernel部分代码深入学习
释放wakelock代码逻辑学习
Wakelock是如何和系统suspend产生关联的

PREVIOUS
ANDROID电源管理之通话中的PSENSOR工作原理分析
(/2017/02/23/PM_PSENSOR_IN_CALL/)

NEXT
吉他谱——李志_和你在一起 **(/2017/02/23/GUITAR_LIZHI_HENIZAIYIQI/)**

FEATURED TAGS (/tags/)

前端 (/tags/#前端) Android (/tags/#Android) frameworks (/tags/#frameworks) AlarmManager (/tags/#AlarmManager) Performance (/tags/#Performance) systrace (/tags/#systrace) PowerManager (/tags/#PowerManager) Wakelock (/tags/#Wakelock) Guitar (/tags/#Guitar) 民谣 (/tags/#民谣) 赵雷 (/tags/#赵雷) Doze (/tags/#Doze) Android Performance Patterns (/tags/#Android Performance Patterns)

FRIENDS

待遇见志同道合的你 (https://github.com) 小明 (http://www.betterming.cn)

(https://twitter.com/chendongqi) (https://www.zhihu.com/people/chendongqi)
(http://weibo.com/chendongqi) (https://www.facebook.com/chendongqi)
(https://github.com/chendongqi) (https://www.linkedin.com/in/firstname-lastname-idxxxx)