```
 private void goToSleepInternal(long eventTime, int reason) {
         synchronized (mLock) {
         if (goToSleepNoUpdateLocked(eventTime, reason)) {
             updatePowerStateLocked();
         }
         }
         }
  //只是更新狀態，沒有實際的執行sleep的動作
         private boolean goToSleepNoUpdateLocked(long eventTime, int reason) {
         //首先判斷sleep的條件，以下情況返回false，sleep的時間小於上次sleep的時間、本來
就處於sleep狀態、boot沒有完成、系統沒有準備好。
         if (eventTime < mLastWakeTime || mWakefulness == WAKEFULNESS_ASLEEP
                 || !mBootCompleted || !mSystemReady) {
         return false;
         }

         switch (reason) {
         case PowerManager.GO_TO_SLEEP_REASON_DEVICE_ADMIN:
                 Slog.i(TAG, "Going to sleep due to device administration policy...");
                 break;
         case PowerManager.GO_TO_SLEEP_REASON_TIMEOUT: //wakeup的時間用完了
                 Slog.i(TAG, "Going to sleep due to screen timeout...");
                 break;
         default:
                 Slog.i(TAG, "Going to sleep by user request...");
                 reason = PowerManager.GO_TO_SLEEP_REASON_USER;//用戶請求sleep
                 break;
         }

         sendPendingNotificationsLocked();  //發送之前的廣播，同時將標誌位置為false
         mNotifier.onGoToSleepStarted(reason);//將sleep的原因保存起來
         mSendGoToSleepFinishedNotificationWhenReady = true;

         mLastSleepTime = eventTime; //更新最近sleep的時間
         mDirty |= DIRTY_WAKEFULNESS; //保存mWakefulness標誌位的變化，只能表示是否
變化了，如果需要知道具體值，需要查看mWakefulness
```

```
        mWakefulness = WAKEFULNESS_ASLEEP; //表示device處於的狀態，是醒著的還是
睡眠中，或者處於兩者之間的一種狀態

        // 計算需要清除的鎖數量，沒有包括PARTIAL_WAKE_LOCK類型的鎖
        int numWakeLocksCleared = 0;
        final int numWakeLocks = mWakeLocks.size();/
        for (int i = 0; i < numWakeLocks; i++) {
        final WakeLock wakeLock = mWakeLocks.get(i);
        switch (wakeLock.mFlags & PowerManager.WAKE_LOCK_LEVEL_MASK) {
                case PowerManager.FULL_WAKE_LOCK:
                case PowerManager.SCREEN_BRIGHT_WAKE_LOCK:
                case PowerManager.SCREEN_DIM_WAKE_LOCK:
                numWakeLocksCleared += 1;
                break;
        }
        }
        EventLog.writeEvent(EventLogTags.POWER_SLEEP_REQUESTED,
numWakeLocksCleared);
        return true;
        }


        private void updatePowerStateLocked() {

        if (!mSystemReady || mDirty == 0) {

        return;

        }


        // Phase 0: Basic state updates.

     updateIsPoweredLocked(mDirty);//更新充電相關的狀態

     updateStayOnLocked(mDirty);//更新常亮的設置標誌位


        // 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; //重置變量，這點從側面說明了這次updatePowerState之後，會把前面所有
發生的power state執行，不會讓其影響到下一次的變化。同時也在為下一次的
                        // power state從頭開始更新做好準備。

        updateWakeLockSummaryLocked(dirtyPhase1);
        updateUserActivitySummaryLocked(now, dirtyPhase1);
        if (!updateWakefulnessLocked(dirtyPhase1)) {
                break;
        }
        }

        // Phase 2: Update dreams and display power state.
        updateDreamLocked(dirtyPhase2);
        updateDisplayPowerStateLocked(dirtyPhase2);

        // Phase 3: Send notifications, if needed.
        if (mDisplayReady) {
        sendPendingNotificationsLocked();
        }

        // Phase 4: Update suspend blocker.
        // Because we might release the last suspend blocker here, we need to make sure
        // we finished everything else first!
        updateSuspendBlockerLocked();
        }
        /**
        * 更新mIsPowered的值，即設置DIRTY_IS_POWERED位
        */
        private void updateIsPoweredLocked(int dirty) {
        if ((dirty & DIRTY_BATTERY_STATE) != 0) { //判斷電池狀態是否發生變化
        final boolean wasPowered = mIsPowered;
        final int oldPlugType = mPlugType;
        mIsPowered =
mBatteryService.isPowered(BatteryManager.BATTERY_PLUGGED_ANY);
        mPlugType = mBatteryService.getPlugType();
        mBatteryLevel = mBatteryService.getBatteryLevel();

        //如果充電狀態或連接狀態變化，則置mDirty的DIRTY_IS_POWERED位為1
        if (wasPowered != mIsPowered || oldPlugType != mPlugType) {
                mDirty |= DIRTY_IS_POWERED;

                // 更新無線充電的狀態，判斷是否在進行無線充電.
                final boolean dockedOnWirelessCharger = mWirelessChargerDetector.update(
                mIsPowered, mPlugType, mBatteryLevel);

                //判斷插入拔出連接是否喚醒
                final long now = SystemClock.uptimeMillis();
                if (shouldWakeUpWhenPluggedOrUnpluggedLocked(wasPowered, oldPlugType,
                dockedOnWirelessCharger)) {
            wakeUpNoUpdateLocked(now);//否則更新喚醒狀態
```

```
                }
        //插拔充電連接也算是用戶事件，更新用戶事件狀態
                userActivityNoUpdateLocked(
                now, PowerManager.USER_ACTIVITY_EVENT_OTHER, 0,
Process.SYSTEM_UID);
        //如果正在進行無線充電，發送相關消息
                if (dockedOnWirelessCharger) {
                mNotifier.onWirelessChargingStarted();
                }
        }
        }
        }
  //判斷插入或者拔出充電連接時是否喚醒
        private boolean shouldWakeUpWhenPluggedOrUnpluggedLocked(
        boolean wasPowered, int oldPlugType, boolean dockedOnWirelessCharger) {
        // 除非配置喚醒，否則不喚醒
        if (!mWakeUpWhenPluggedOrUnpluggedConfig) {
        return false;
        }

        // Don't wake when undocked from wireless charger.
        // 當移除無線充電時，不喚醒
        if (wasPowered && !mIsPowered
                && oldPlugType == BatteryManager.BATTERY_PLUGGED_WIRELESS) {
        return false;
        }
```

```
// Don't wake when docked on wireless charger unless we are certain of it.
// 當接入無線充電時，不喚醒
if (!wasPowered && mIsPowered
        && mPlugType == BatteryManager.BATTERY_PLUGGED_WIRELESS
        && !dockedOnWirelessCharger) {
return false;
}

//處於屏保狀態時不喚醒
if (mIsPowered && (mWakefulness == WAKEFULNESS_NAPPING
        || mWakefulness == WAKEFULNESS_DREAMING)) {
return false;
}

// Otherwise wake up!
return true;
}

/**
* Updates the value of mStayOn.
*更新mStayOn的值，如果改變了就設置mDirty的DIRTY_STAY_ON位
*/
private void updateStayOnLocked(int dirty) {
//判斷電池狀態位和設置位有沒有變動
if ((dirty & (DIRTY_BATTERY_STATE | DIRTY_SETTINGS)) != 0) {
final boolean wasStayOn = mStayOn;
//device的屬性Settings.Global.STAY_ON_WHILE_PLUGGED_IN為true，並且沒有達到
```

```
電池充電時持續開屏時間的最大值（也就是說，在插入電源後的一段時間內保
    //持開屏狀態），那麼mStayOn為真
    if (mStayOnWhilePluggedInSetting != 0
        && !isMaximumScreenOffTimeoutFromDeviceAdminEnforcedLocked())
        mStayOn = mBatteryService.isPowered(mStayOnWhilePluggedInSetting);
    } else {
        mStayOn = false;
    }

    if (mStayOn != wasStayOn) {//保存是否變更了DIRTY_STAY_ON位，具體是true還是
false必須查看mStayOn變量
        mDirty |= DIRTY_STAY_ON;
    }
    }
    }

    //統計所有喚醒鎖的狀態，將其保存到變量mWakeLockSummary中，當系統處於sleep
狀態時，會忽略喚醒鎖，除了PARTIAL_WAKE_LOCK類型的鎖
    private void updateWakeLockSummaryLocked(int dirty) {
    //喚醒鎖或者係統狀態發生變化
    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);
        switch (wakeLock.mFlags & PowerManager.WAKE_LOCK_LEVEL_MASK) {
```

```
        case PowerManager.PARTIAL_WAKE_LOCK:
        mWakeLockSummary |= WAKE_LOCK_CPU;
        break;
        case PowerManager.FULL_WAKE_LOCK:
        if (mWakefulness != WAKEFULNESS_ASLEEP) {
            mWakeLockSummary |= WAKE_LOCK_CPU
            | WAKE_LOCK_SCREEN_BRIGHT |
WAKE_LOCK_BUTTON_BRIGHT;
            if (mWakefulness == WAKEFULNESS_AWAKE) {
            mWakeLockSummary |= WAKE_LOCK_STAY_AWAKE;
            }
        }
        break;
        case PowerManager.SCREEN_BRIGHT_WAKE_LOCK:
        if (mWakefulness != WAKEFULNESS_ASLEEP) {
            mWakeLockSummary |= WAKE_LOCK_CPU |
WAKE_LOCK_SCREEN_BRIGHT;
            if (mWakefulness == WAKEFULNESS_AWAKE) {
            mWakeLockSummary |= WAKE_LOCK_STAY_AWAKE;
            }
        }
        break;
        case PowerManager.SCREEN_DIM_WAKE_LOCK:
        if (mWakefulness != WAKEFULNESS_ASLEEP) {
            mWakeLockSummary |= WAKE_LOCK_CPU |
WAKE_LOCK_SCREEN_DIM;
            if (mWakefulness == WAKEFULNESS_AWAKE) {
```

```
            mWakeLockSummary |= WAKE_LOCK_STAY_AWAKE;
            }
        }
        break;
        case PowerManager.PROXIMITY_SCREEN_OFF_WAKE_LOCK:
        if (mWakefulness != WAKEFULNESS_ASLEEP) {
            mWakeLockSummary |= WAKE_LOCK_PROXIMITY_SCREEN_OFF;
        }
        break;
        }
    }
    }
    }
    }

    /**
    *
    * 統計用戶事件，並發送一個延時消息觸發下一狀態。要注意的是在
updateUserActivitySummaryLocked在中鎖屏時間和變暗時間的的比較。假如說在系統中設置
的睡眠時間
    *是30s，而在PowerManagerService中默認的SCREEN_DIM_DURATION是7s，這就
意味著：如果沒有用戶活動的話，在第23s，設備的屏幕開始變換，持續7s時間，然後
    *屏幕開始關閉。      */
    private void updateUserActivitySummaryLocked(long now, int dirty) {
    // Update the status of the user activity timeout timer.
    if ((dirty & (DIRTY_USER_ACTIVITY | DIRTY_WAKEFULNESS | DIRTY_SETTINGS))
!= 0) {
    mHandler.removeMessages(MSG_USER_ACTIVITY_TIMEOUT);

    long nextTimeout = 0;
    if (mWakefulness != WAKEFULNESS_ASLEEP) {//當處於sleep狀態時忽略用戶事件
        final int screenOffTimeout = getScreenOffTimeoutLocked();
        final int screenDimDuration = getScreenDimDurationLocked(screenOffTimeout);

        mUserActivitySummary = 0;
        if (mLastUserActivityTime >= mLastWakeTime) { //只有發生用戶喚醒事件才更
新超時時間和狀態
        nextTimeout = mLastUserActivityTime
            + screenOffTimeout - screenDimDuration;
        if (now < nextTimeout) {
        mUserActivitySummary |= USER_ACTIVITY_SCREEN_BRIGHT;
        } else {
        nextTimeout = mLastUserActivityTime + screenOffTimeout;
        if (now < nextTimeout) {
            mUserActivitySummary |= USER_ACTIVITY_SCREEN_DIM;
        }
        }
        }
        //如果當前不是處於亮屏或者暗屏狀態，那麼
        if (mUserActivitySummary == 0
        && mLastUserActivityTimeNoChangeLights >= mLastWakeTime) {
        nextTimeout = mLastUserActivityTimeNoChangeLights + screenOffTimeout;
        if (now < nextTimeout
            && mDisplayPowerRequest.screenState
```

```
                    != DisplayPowerRequest.SCREEN_STATE_OFF) {
                mUserActivitySummary = mDisplayPowerRequest.screenState
                    == DisplayPowerRequest.SCREEN_STATE_BRIGHT ?
                    USER_ACTIVITY_SCREEN_BRIGHT :
        USER_ACTIVITY_SCREEN_DIM;
                }
                }
                if (mUserActivitySummary != 0) {
                Message msg =
        mHandler.obtainMessage(MSG_USER_ACTIVITY_TIMEOUT);
                msg.setAsynchronous(true);
                mHandler.sendMessageAtTime(msg, nextTimeout);
                }
            } else {
                mUserActivitySummary = 0;
            }
            }
            }
            }
```