android wake_lock 简述

Last edited by **caoquanli** 1 month ago

Wakelocks

WakeLock是android系统中一种锁的机制，只要有进程持有这个锁，系统就无法进入休眠状态。应用程序要申请WakeLock时，需要在清单文件中配置android.Manifest.permission.WAKE_LOCK权限。

根据**作用时间**wakelock可以分为

1. 超时锁 -> 到达给定时间后自动释放,实现原理为Handler
2. 永久锁 -> 只要获取了Wakelock锁,则必须显示的进行释放,否则系统一直持有该锁

根据**释放原则**wakelock可以分为

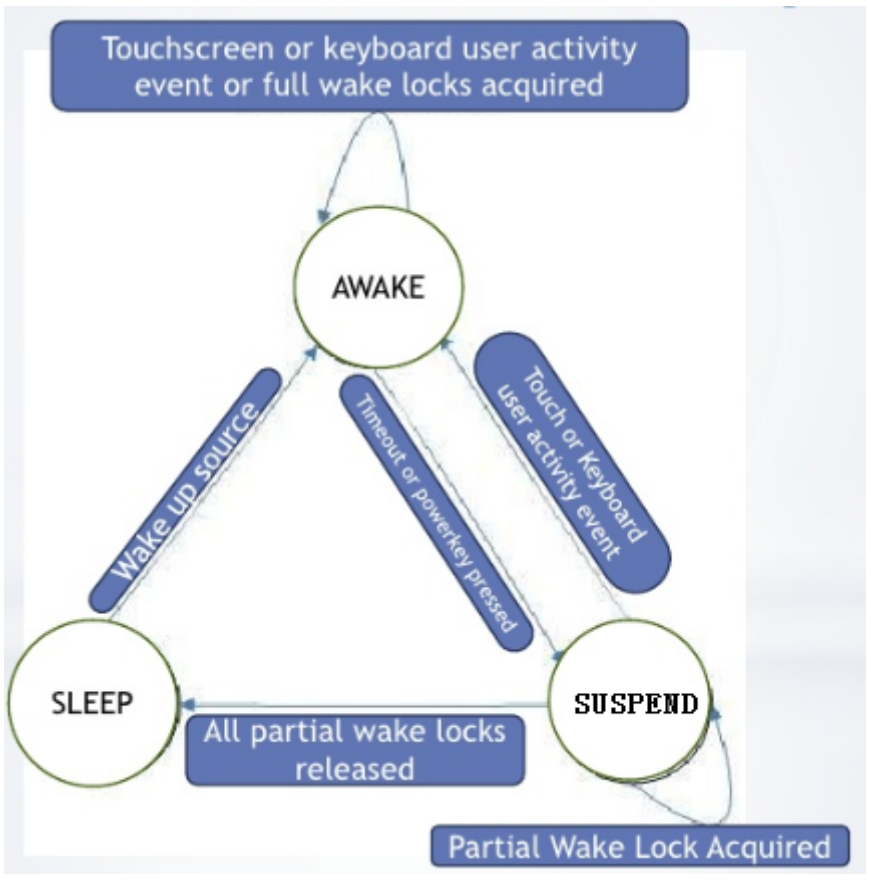
1. 计数锁 -> 每申请一次都需要对应释放
2. 非计数锁 -> 不管申请多少次,只要释放一次该锁

默认情况下, Android尝试让系统尽可能的进入睡眠或是挂起状态

- App 运行在Dalvik虚拟机中将会组织系统进入睡眠或挂起状态
- 通过使用wakelock也可以解决上述问题
- 如果没有(active)wakelock -> Cpu将会关闭
- 如果有partial wakelock -> 屏幕和键盘将会被关闭

wake_lock_type	
PARTIAL_WAKE_LOCK	保证CPU运行，屏幕和键盘灯允许关闭
SCREEN_DIM_WAKE_LOCK	保证屏幕亮（可能是dim状态），键盘灯允许关闭
SCREEN_BRIGHT_WAKE_LOCK	保证屏幕亮（at full brightness），键盘灯允许关闭
FULL_WAKE_LOCK	保证屏幕和键盘灯亮（at full brightness
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

系统状态的变化



一个简单的例子

申请wakelock和释放的代码片段

```
PowerManager pm = (PowerManager) getSystemService(Context.POWER_SERVICE);
PowerManager.WakeLock wl = pm.newWakeLock(PowerManager.FULL_WAKE_LOCK, "My Tag");
wl.acquire();
Wl.acquire(int timeout);//超时锁

wl.release();
```

大致流程

acquire时大致时序

在整个WakeLock机制中：

- PowerManger.WakeLock：PowerManagerService和其他应用、服务交互的接口
- PowerManagerService.WakeLock：PowerManager.WakeLock在PMS中的表现形式
- SuspendBlocker：PowerManagerService.WakeLock在向底层节点（/sys/power/wake_lock&wake_unlock）操作时的表现形式

PowerManager中的WakeLock

new && acquire & release 详细时序

其中，由于updateSuspendBlockerLocked涉及和JNI层的操作，会在之后的流程再说

要获取、申请Wakelock时，直接通过PowerManager的WakeLock进行。它作为系统服务的接口来供应用调用。

1. 获取WakeLock对象
   获取WakeLock实例在PowerManager中进行。在应用中获取WakeLock对象，方式如下：

```
PowerManager.WakeLock mWakeLock =
              mPowerManager.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK, TAG);
```

应用中获取wakelock对象，获取的是位于PowerManager中的内部类——WakeLock的实例

```
public WakeLock newWakeLock(int levelAndFlags, String tag) {
      validateWakeLockParameters(levelAndFlags, tag);
      return new WakeLock(levelAndFlags, tag, mContext.getOpPackageName());
}
```

```
  WakeLock(int flags, String tag, String packageName) {
            mFlags = flags;
            mTag = tag;
            mPackageName = packageName;
            mToken = new Binder();
            mTraceName = "WakeLock (" + mTag + ")";
        }
```

2. 申请WakeLock 当获取到WakeLock实例后，就可以申请WakeLock了

```
mWakeLock.acquire();
```

- 永久锁源码

```
public void acquire() {
    synchronized (mToken) {
        acquireLocked();
    }
}
```

- 超时锁源码

```
public void acquire(long timeout) {
    synchronized (mToken) {
        acquireLocked();
        //申请锁之后，内部会维护一个Handler去完成自动释放锁
        mHandler.postDelayed(mReleaser, timeout);
    }
}
```

(选读)接上->调用到PWMS

```
  private void acquireLocked() {
     //应用每次申请wakelock，内部计数和外部计数加1
     mInternalCount++;
     mExternalCount++;
     //如果是非计数锁或者内部计数值为1,即第一次申请该锁，才会真正去申请
     if (!mRefCounted || mInternalCount == 1) {
         mHandler.removeCallbacks(mReleaser);
         Trace.asyncTraceBegin(Trace.TRACE_TAG_POWER, mTraceName, 0);
         try {
             //向PowerManagerService申请锁
             mService.acquireWakeLock(mToken, mFlags, mTag, mPackageName, mWorkSource,
                     mHistoryTag);
         } catch (RemoteException e) {
             throw e.rethrowFromSystemServer();
         }
         //表示此时持有该锁
         mHeld = true;
     }
}
```

PowerManagerService中的wakelock

申请wakelock以及释放wakelock在pwms的实现-即 和jni对接部分 **主要依靠updateSuspendBlockerLocked()方法**

首先看一下pwms的构造方法

```
public PowerManagerService(Context context) {
    ...
    synchronized (mLock) {
        mWakeLockSuspendBlocker = createSuspendBlockerLocked("PowerManagerService.WakeLocks
        mDisplaySuspendBlocker = createSuspendBlockerLocked("PowerManagerService.Display");
        mDisplaySuspendBlocker.acquire();
        mHoldingDisplaySuspendBlocker = true;
        mHalAutoSuspendModeEnabled = false;
        mHalInteractiveModeEnabled = true;
        ...
    }
}
```

在PWMS的构造方法中创建了两个SuspendBlocker对象：
mWakeLockSuspendBlocker：获取一个PARTIAL_WAKELOCK类型的WakeLock使CPU保持活动状态
mDisplaySuspendBlocker：当屏幕亮屏、用户活动时使CPU保持活动状态。
因此实际上，上层PowerManager申请和释放锁，最终在PMS中都交给了SuspendBlocker去申请和释放锁。也可以说SuspendBlocker类的两个对象是WakeLock锁反映到底层的对象。 **只要持有二者任意锁，都会使得CPU处于活动状态。**

通过对WakeLock锁的申请和释放流程分析，知道实际上通过操作/sys/power/wake_lock和 /sys/power/wake_unlock节点来控制设备的唤醒和休眠。
当应用需要**唤醒设备**时，申请一个WakeLock锁，最终会在/sys/power/wake_lock 中写入SuspendBlocker锁名，从而保持了设备的唤醒。
当应用执行完操作后，则**释放WakeLock锁**，最终会在/sys/power/wake_unlock 中写入SuspendBlocker锁名。

现在PWMS中的updateSuspendBlockerLocked()方法就很好理解了

```
private void updateSuspendBlockerLocked() {
        //是否需要保持CPU活动状态的SuspendBlocker锁，具体表现为持有Partical WakeLock
final boolean needWakeLockSuspendBlocker =
           ((mWakeLockSummary & WAKE_LOCK_CPU) != 0);
    //是否需要保持CPU活动状态的SuspendBlocker锁，具体表现保持屏幕亮
    final boolean needDisplaySuspendBlocker = needDisplaySuspendBlockerLocked();
    //是否自动挂起，如果不需要屏幕保持唤醒，则说明可以自动挂起CPU
    final boolean autoSuspend = !needDisplaySuspendBlocker;
    //是否处于交互模式，屏幕处于Bright或者Dim状态时为true
    final boolean interactive = mDisplayPowerRequest.isBrightOrDim();
    //mDecoupleHalAutoSuspendModeFromDisplayConfig:自动挂起模式和显示状态解偶
    if (!autoSuspend && mDecoupleHalAutoSuspendModeFromDisplayConfig) {
        //禁止CPU自动挂起模式
        setHalAutoSuspendModeLocked(false);
    }
    //如果存在PARTIAL_WAKE_LOCK类型的WakeLock,申请mWakeLockSuspendBlocker锁
    if (needWakeLockSuspendBlocker && !mHoldingWakeLockSuspendBlocker) {
        mWakeLockSuspendBlocker.acquire();
        mHoldingWakeLockSuspendBlocker = true;
    }
    //如果当前屏幕需要保持亮屏，申请mDisplaySuspendBlocker锁
    if (needDisplaySuspendBlocker && !mHoldingDisplaySuspendBlocker) {
        mDisplaySuspendBlocker.acquire();
        mHoldingDisplaySuspendBlocker = true;
    }

    if (mDecoupleHalInteractiveModeFromDisplayConfig) {
        //设置Hal层交互模式?
        if (interactive || mDisplayReady) {
            setHalInteractiveModeLocked(interactive);
        }
    }
    //如果不再持有PARTIAL_WAKELOCK类型的WakeLock锁，释放mWakeLockSuspendBlocker锁
    if (!needWakeLockSuspendBlocker && mHoldingWakeLockSuspendBlocker) {
        mWakeLockSuspendBlocker.release();
        mHoldingWakeLockSuspendBlocker = false;
```

```
    }
    //如果不再需要屏幕保持亮屏，释放mDisplaySuspendBlocker锁
    if (!needDisplaySuspendBlocker && mHoldingDisplaySuspendBlocker) {
        mDisplaySuspendBlocker.release();
        mHoldingDisplaySuspendBlocker = false;
    }
        //启动自动挂起模式
    if (autoSuspend && mDecoupleHalAutoSuspendModeFromDisplayConfig) {
        setHalAutoSuspendModeLocked(true);
    }
}
```

PowerManagerService.Broadcasts锁

这个类型的SuspendBlocker并没有在PMS中进行实例化，它以构造方法的形式传入了Notifier中，Notifier类相当于是PMS的”中介“，PMS中和其他服务的部分交互通过Notifier进行，还有比如亮屏广播、灭屏广播等，都是由PMS交给Notifier来发送。

如果CPU在广播发送过程中进入休眠，则广播无法发送完成，因此，需要一个锁来保证Notifier中广播的成功发送，这就是PowerManagerService.Broadcasts 锁的作用，当广播发送完毕后，该锁立即就释放了。