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

Android电源管理之释放电源锁

介绍Android系统中电源锁的释放流程

Posted by Cheson on February 25, 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 Android电源管理之申请电源锁 (https://chendongqi.github.io/blog/2017/02/23/pm_wl_acquire_flow/)中已经介绍了在app中电源锁的申请方法，也提到了两种电源锁的申请方法，非定时的和定时的。非定时的电源锁没有手动release的操作是不会被释放的，可能就会导致系统无法进入休眠，待机电流过高这类问题。释放电源锁的操作非常简单，还是以在 Android电源管理之申请电源锁 (https://chendongqi.github.io/blog/2017/02/23/pm_wl_acquire_flow/)中用到的测试代码为例，在onCreate时进行申请电源锁的操作

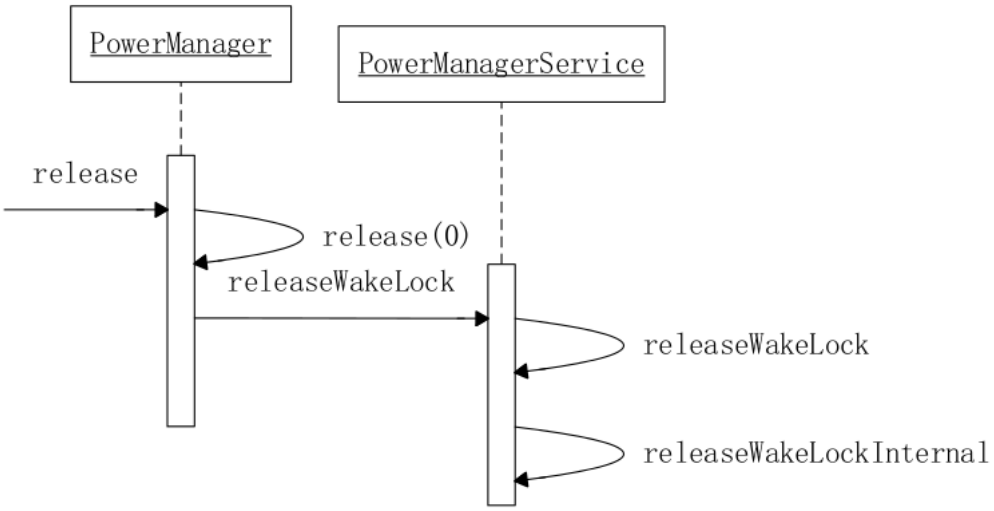
```
PowerManager pm = (PowerManager) getSystemService(Context.POWER_SERVICE);
wakeLock = pm.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK, "chendongqi_wakelock");
wakeLock.acquire();
```

那么在onDestroy中加入释放的操作

```
@Override
protected void onDestroy() {
    wakeLock.release();
    super.onDestroy();
}
```

3.Frameworks流程

调用PowerManager.WakeLock的release方法之后流程就进入了frameworks，方法调用流程参照下图。



来看一下PowerManager.java中的release方法

```
public void release(int flags) {
    synchronized (mToken) {
        // 如果是非计数锁，或者计数锁的数量降到了0，进行释放操作
        if (!mRefCounted || --mCount == 0) {
            mHandler.removeCallbacks(mReleaser);
            if (mHeld) {
                Trace.asyncTraceEnd(Trace.TRACE_TAG_POWER, mTraceName, 0);
                try {
                    mService.releaseWakeLock(mToken, flags);
                } catch (RemoteException e) {
                }
                mHeld = false;
            }
        }
        if (mCount < 0) {
            throw new RuntimeException("WakeLock under-locked " + mTag);
        }
    }
}
```

这里补充一点 Android电源管理之申请电源锁 (https://chendongqi.github.io/blog/2017/02/23/pm_wl_acquire_flow/)中遗落的知识。电源锁分类还可以分成计数锁和非计数锁，可以通过 setReferenceCounted(boolean value) 来指定，一般默认为计数机制。这两种机制的区别在于，前者无论 acquire() 了多少次，只要通过一次 release()即可解锁。而后者正真解锁是在（ –count == 0 ）的时候，同样当 (count++ == 0) 的时候才会去申请加锁。所以 PowerManager.WakeLock 的计数机制并不是正真意义上的对每次请求进行申请 / 释放每一把锁，它只是对同一把锁被申请 / 释放的次数进行了统计，然后再去操作。

然后代码流程走了PMS中的releaseWakeLock，这里没有什么和wakelock相关的操作，又进一步调用了PMS中的releaseWakeLockInternal方法，来看下代码：

```
    private void releaseWakeLockInternal(IBinder lock, int flags) {
        synchronized (mLock) {
            int index = findWakeLockIndexLocked(lock);// 查找该wakelock的索引
            if (index < 0) {
                if (DEBUG_SPEW) {
                    Slog.d(TAG, "releaseWakeLockInternal: lock=" + Objects.hashCode(lock)
                            + " [not found], flags=0x" + Integer.toHexString(flags));
                }
                return;
            }

            WakeLock wakeLock = mWakeLocks.get(index);// 取得该wakelock对象
            wakeLock.mTotalTime = SystemClock.uptimeMillis() - wakeLock.mActiveSince;// 计算其持有的时间

            if (DEBUG_SPEW) {// 非常重要的一条log，在syslog中打出，通常用来排查电源锁持有是否异常
                Slog.d(TAG, "releaseWakeLockInternal: lock=" + Objects.hashCode(lock)
                        + " [" + wakeLock.mTag + "], flags=0x" + Integer.toHexString(flags) + ", total_time=" + wakeLock.mTotalTime +
            }

            // 检查是否是pSensor持有的wakelock类型
            if ((flags & PowerManager.RELEASE_FLAG_WAIT_FOR_NO_PROXIMITY) != 0) {
                mRequestWaitForNegativeProximity = true;
            }

            wakeLock.mLock.unlinkToDeath(wakeLock, 0);// 注销binder进入death的回调方法
            removeWakeLockLocked(wakeLock, index);// 移除掉该wakelock
        }
    }
```

随后就开始移除wakelock的动作和一些善后处理了：

```
    private void removeWakeLockLocked(WakeLock wakeLock, int index) {
        mWakeLocks.remove(index);// 从mWakeLocks这个队列中移除到这个wakelock
        notifyWakeLockReleasedLocked(wakeLock);// 通过Notifier将释放wakelock的消息发送出去

        // 特殊处理：如果wakelock的flag中有ON_AFTER_RELEASE，则会在这个wakelock释放掉之后继续保持亮屏for a little time
        applyWakeLockFlagsOnReleaseLocked(wakeLock);
        mDirty |= DIRTY_WAKE_LOCKS;// 在mDirty标志中加入wakelock变化的标志
        updatePowerStateLocked();// mDirty标志改变，更新电源状态
    }
```

4. Native流程

PMS中的release wakelock流程代码中绝大多数都是在对上层的电源状态做一些检查、处理和通知善后的工作。而且frameworks层对上层应用申请的wakelock的管理也就是在mWakeLocks这个队列中来完成的，在真正意义上并不会对系统休眠产生影响。而真正往底层走下去的代码入口和申请电源锁的流程一样，都是在updatePowerStateLocked这个方法中通过各种标志来更新电源管理的状态，而wakelock相关的就是在phase 5中的updateSuspendBlockerLocked方法里。

```
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);
        }
    }

    // 以下为release相关

    // Then release suspend blockers if needed.
    if (!needWakeLockSuspendBlocker && mHoldingWakeLockSuspendBlocker) {
        mWakeLockSuspendBlocker.release();// 释放阻塞cpu休眠的blocker
        mHoldingWakeLockSuspendBlocker = false;
    }
    if (!needDisplaySuspendBlocker && mHoldingDisplaySuspendBlocker) {
        mDisplaySuspendBlocker.release();// 释放阻塞display休眠的blocker
        mHoldingDisplaySuspendBlocker = false;
    }

    // Enable auto-suspend if needed.
    if (autoSuspend && mDecoupleHalAutoSuspendModeFromDisplayConfig) {
        setHalAutoSuspendModeLocked(true);
    }
}
```

这个方法真正的含义应该是更新阻塞源的一个处理函数，从系统休眠的角度来看，上层申请的电源锁到底层是转换成了阻塞cpu或者display进入休眠的阻塞源(blocker)。前半段是申请电源锁相关，当有一个至少一个阻塞源的时候这块代码就起效了。而释放电源锁的流程中，需要当所有电源锁都释放完，阻塞的标志位才会清空，这部分代码也才会起效。

于是需要进一步看SuspendBlocker中release方法的实现，SuspendBlocker是PMS中定义的一个接口，由PMS中的内部类SuspendBlockerImpl实现其接口方法，来看SuspendBlockerImpl的release方法：

```
@Override
public void release() {
    synchronized (this) {
        mReferenceCount -= 1;// 计数引用减1
        if (mReferenceCount == 0) {// 如果正好是0
            if (DEBUG_SPEW) {
                Slog.d(TAG, "Releasing suspend blocker \"" + mName + "\".");
            }
            nativeReleaseSuspendBlocker(mName);// 释放该电源锁
            Trace.asyncTraceEnd(Trace.TRACE_TAG_POWER, mTraceName, 0);
        } else if (mReferenceCount < 0) {// 异常情况
            Slog.wtf(TAG, "Suspend blocker \"" + mName
                    + "\" was released without being acquired!", new Throwable());
            mReferenceCount = 0;
        }
    }
}
```

这里释放掉的电源名称已经不是上层申请时取名的TAG名称了，而是特定的几个阻塞源的名字。是在创建SuspendBlocker时传入的，例如：

```
mWakeLockSuspendBlocker = createSuspendBlockerLocked("PowerManagerService.WakeLocks");
mDisplaySuspendBlocker = createSuspendBlockerLocked("PowerManagerService.Display");
```

所以这部分log也可以从syslog中作为调查上层阻塞源和系统休眠关系的线索。

Native层的代码位于frameworks/base/services/core/jni/com_android_server_power_PowerManagerService.cpp，从nativeReleaseSuspendBlocker方法开始：

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

5. HAL层

通过release_wake_lock方法把代码带到了HAL层，位于hardware/libhardware_legacy/power/power.c

```
release_wake_lock(const char* id)
{
    initialize_fds();// 初始化设备节点

//    ALOGI("release_wake_lock id='%s'\n", id);

    if (g_error) return -g_error;

    ssize_t len = write(g_fds[RELEASE_WAKE_LOCK], id, strlen(id));// 写入设备节点
    return len >= 0;
}
```

这里做了两件事，首先是初始化设备节点的动作，这个在 Android电源管理之申请电源锁一文中的HAL章节(https://chendongqi.github.io/blog/2017/02/23/pm_wl_acquire_flow/#5-hal流程)中有同样的介绍，本篇不再赘述；第二个动作就是写入设备节点。在释放电源锁时写入节点的名称为g_fds[RELEASE_WAKE_LOCK]，RELEASE_WAKE_LOCK又是在枚举类型中定义的变量：

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

之前的操作打开了两个设备节点，路径存放于g_fds数组中，所以g_fds[RELEASE_WAKE_LOCK]刚好对应了：

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

中的"/sys/power/wake_unlock"，而写入到节点中的信息也就是传入的char指针类型的变量，也就是释放的电源锁的类型名称，和申请的类型中的名称也是对应的。可以来看下这个节点中信息的例子：

```
root@mlv1:/ # cat sys/power/wake_unlock
KeyEvents PowerManagerService.Broadcasts PowerManagerService.Display PowerManagerService.WakeLocks SensorService_wakelock
```

6. Kernel

和申请电源锁流程类似，在释放时kernel中的处理方法同样还是对通过对节点信息的监听，在kernel-3.18/kernel/power/main.c中注册了"/sys/power/wake_unlock"节点的store和show方法。

```
static ssize_t wake_unlock_show(struct kobject *kobj,
                            struct kobj_attribute *attr,
                            char *buf)
{
        return pm_show_wakelocks(buf, false);
}

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

power_attr(wake_unlock);
```

重点来看store方法对写节点的处理，调用了kernel-3.18/kernel/power/wakelock.c的pm_wake_unlock函数：

```
int pm_wake_unlock(const char *buf)
{
        struct wakelock *wl;
        size_t len;
        int ret = 0;

        if (!capable(CAP_BLOCK_SUSPEND))
                return -EPERM;

        len = strlen(buf);
        if (!len)
                return -EINVAL;

    if (buf[len-1] == '\n')
            len--;

    if (!len)
            return -EINVAL;

    mutex_lock(&wakelocks_lock);

        wl = wakelock_lookup_add(buf, len, false);// 1.查找该wakelock
    if (IS_ERR(wl)) {
            ret = PTR_ERR(wl);
                goto out;
    }
        __pm_relax(&wl->ws);// 2.释放唤醒源

    wakelocks_lru_most_recent(wl);
        wakelocks_gc();// 3.回收wakelock资源

 out:
        mutex_unlock(&wakelocks_lock);
    return ret;
}
```

在释放wakelock时主要有三个关键动作：1）查找wakelock；2）释放wakelock中的唤醒源(wake source)，这里的代码流程应该走到了kernel3.18/drivers/base/power/wakeup.c中的__pm_relax方法，这篇中不再深入，留待后续介绍系统休眠和唤醒时再研究；3）回收wakelock的资源，代码内容也不展开了。

PREVIOUS
ANDROID电源管理之POWER键锁屏流程
(/2017/02/24/PM_LOCK_SCREEN/)

NEXT
ANDROID电源管理之系统休眠 **(/2017/02/27/PM_SUSPEND/)**

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)

Copyright © Cheson Blog 2017

Theme by Cheson (https://github.com/chendongqi/blog) |