

4316

调用SystemServiceManager的startService方法创建PowerManagerService对象并注册到ServiceManager中。PowerManagerService的构造方法代码如下：

```
[java]
1. public PowerManagerService(Context context) {
2.     super(context);
3.     mContext = context;
4.     //创建处理消息的线程和Handler对象
5.     mHandlerThread = new ServiceThread(TAG,
6.         Process.THREAD_PRIORITY_DISPLAY, false/"allowIo"/);
7.     mHandlerThread.start();
8.     mHandler = new PowerManagerHandler(mHandlerThread.getLooper());
9.
10.    synchronized (mLock) {
11.        mWakeLockSuspendBlocker = createSuspendBlockerLocked("PowerManagerService.WakeLocks");
12.        mDisplaySuspendBlocker = createSuspendBlockerLocked("PowerManagerService.Display");
13.        mDisplaySuspendBlocker.acquire();
14.        mHoldingDisplaySuspendBlocker = true;
15.        mHalAutoSuspendModeEnabled = false;
16.        mHalInteractiveModeEnabled = true;
17.
18.        mWakefulness = WAKEFULNESS_AWAKE; //设置PowerManagerService的状态
19.
20.        nativeInit();
21.        nativeSetAutoSuspend(false);
22.        nativeSetInteractive(true);
23.    }
24. }
```

PowerManagerService的构造方法中首先创建了处理消息的线程和发送消息的PowerManagerHandler对象，接着创建了mWakeLockSuspendBlocker对象、mDisplaySuspendBlocker对象。

变量mWakefulness的值被设置成WAKEFULNESS_AWAKE，它用来标示PowerManagerService的状态，一共有四种定义：

! WAKEFULNESS_ASLEEP：表示系统当前处于休眠状态，只能被wakeUp()调用唤醒。

! WAKEFULNESS_AWAKE：表示系统目前处于正常运行状态。

! WAKEFULNESS_DREAMING：表示系统当前正处于播放屏保的状态。

! WAKEFULNESS_DOZING：表示系统正处于“doze”状态。这种状态下只有低耗电的“屏保”可以运行，其他应用进程都被挂起。

最后，构造方法调用了nativeInit()方法，主要工作就是装载“Power”模块，之后调用模块的初始化函数init()。

系统准备工作—SystemReady方法

SystemService创建PowerManagerService后，还会调用它的SystemReady()方法，相当于在系统准备就绪后对PowerManagerService再进行一些初始化工作。SystemReady()方法代码如下：

```
[java]
1. public void systemReady(IAppOpsService appOps) {
2.     synchronized (mLock) {
3.         mSystemReady = true;
4.         mAppOps = appOps;
5.         mDreamManager = getLocalService(DreamManagerInternal.class); //获取DreamManagerService对象
6.
7.         mDisplayManagerInternal = getLocalService(DisplayManagerInternal.class);
8.         //DisplayManagerService
9.         mPolicy = getLocalService(WindowManagerPolicy.class); //WindowManagerPolicy
10.        mBatteryManagerInternal = getLocalService(BatteryManagerInternal.class);
11.        //BatteryService
12.
13.        //获取最小、最大、默认屏幕亮度
14.        . . . . .
15.        //创建SensorManager对象，用于和SensorService交互
16.        SensorManager sensorManager = new SystemSensorManager(mContext, mHandler.getLooper());
17.
18.        mBatteryStats = BatteryStatsService.getService(); //获得BatteryStatsService的引用对象
19.        mNotifier = new Notifier(Looper.getMainLooper(), mContext, mBatteryStats,
20.            mAppOps, createSuspendBlockerLocked("PowerManagerService.Broadcasts"),
21.            mPolicy); //创建Notifier对象
22.
23.        mWirelessChargerDetector = new WirelessChargerDetector(sensorManager,
24.            createSuspendBlockerLocked("PowerManagerService.WirelessChargerDetector"),
25.            mHandler); //创建检测无线充电的对象WirelessChargerDetector
26.        mSettingsObserver = new SettingsObserver(mHandler); //创建监听系统设置项变化的对象
27.
28.        mLightsManager = getLocalService(LightsManager.class); //LightsManager对象
29.        mAttentionLight = mLightsManager.getLight(LightsManager.LIGHT_ID_ATTENTION);
30.
31.        //初始化Power的管理模块
32.        mDisplayManagerInternal.initPowerManagement(
33.            mDisplayPowerCallbacks, mHandler, sensorManager);
34.
35.        . . . . . //注册广播接收器
36.        . . . . . //注册监听更多的settns项的变化
37.        // Go.
38.        readConfigurationLocked();
39.        updateSettingsLocked();
40.        mDirty |= DIRTY_BATTERY_STATE;
41.        updatePowerStateLocked();
42.    }
43. }
```

```
49. | }
```

systemReady()方法中通过调用getLocalService()方法得到一些在SystemServer中运行的内部服务的对象。在systemservice中也创建了一些内部使用的服务，这些服务没有通过ServiceManager发布，而是通过内部的LocalService类来管理。这些内部服务的共同特征是从SystemService类派生，通过getLocalService()方法可以获得参数关联的内部服务对象。

systemReady()方法完成的主要工作如下：

- l 获取最小、最大、默认3种屏幕亮度。
- l 创建SystemSensorManager对象，用于和SensorService交互。
- l 创建Notifier对象。用于广播系统中和Power相关的变化。
- l 创建WirelessChargerDetector对象，用于无线充电检测的传感器。
- l 调用DisplayManagerService的initPowerManagement()方法来初始化Power管理模块。
- l 注册Observer监听系统设置的变化。
- l 监听其他模块广播的Intent。PowerManagerService需要关注系统的变化，这里注册了很多系统广播的接收器。包括系统启动完成、“屏保”启动和关闭、用户切换、Dock插拔等。

报告用户活动—userActivity接口

PowerManager是PowerManagerService的代理类，它提供了一些接口让用户进程可以和PowerManagerService交互，下面分析一些接口来进一步了解PowerManagerService的工作。userActivity()接口用于用户进程向PowerManagerService报告用户影响系统休眠的活动。例如，用户点击屏幕时，系统会调用该方法来告诉PowerManagerService用户点击的时间，这样PowerManagerService将更新内部保存的时间值，从而推迟系统休眠的时间。userActivity()方法主要通过调用内部的userActivityInternal()方法来完成工作，userActivity流程图如图2-2所示：

图2-2 userActivity流程图

具体代码如下：



```
[java]
1. private void userActivityInternal(long eventTime, int event, int flags, int uid) {
2.     synchronized (mLock) {
3.         if (userActivityNoUpdateLocked(eventTime, event, flags, uid)) {
4.             updatePowerStateLocked();
5.         }
6.     }
7. }

userActivityInternal()先调用了userActivityNoUpdateLocked()方法，然后再调用updatePowerStateLocked()方法。userActivityNoUpdateLocked()方法只是把参数保存到内部变量中，并不会采取任何动作，而PowerManagerService中核心的方法是updatePowerStateLocked()。我们先看下userActivityNoUpdateLocked()方法：

[java]
1. private boolean userActivityNoUpdateLocked(longeventTime,intevent,intflags,intuid) {
2.     if (eventTime< mLastSleepTime || eventTime < mLastWakeTime
3.         || !mBootCompleted ||!mSystemReady) {
4.         return false;
5.     }
6.
7.     Trace.traceBegin(Trace.TRACE_TAG_POWER, "userActivity");
8.     try {
9.         if (eventTime> mLastInteractivePowerHintTime) {
10.            powerHintInternal(POWER_HINT_INTERACTION, 0);//powerHintInternal是通过JNI调用底层函数，将cpu频率提高等等
11.            mLastInteractivePowerHintTime= eventTime;//记录时间
12.        }
13.
14.        mNotifier.onUserActivity(event,uid);//发出通知
15.
16.        if (mWakefulness== WAKEFULNESS_ASLEEP
17.            || mWakefulness ==WAKEFULNESS_DOZING
18.            || (flags &PowerManager.USER_ACTIVITY_FLAG_INDIRECT) != 0) {
19.            return false;
20.        }//如果系统处于休眠或doze模式，返回
21.
22.        if ((flags& PowerManager.USER_ACTIVITY_FLAG_NO_CHANGE_LIGHTS) !=0){
23.            if (eventTime> mLastUserActivityTimeNoChangeLights
24.                && eventTime> mLastUserActivityTime) {
25.                mLastUserActivityTimeNoChangeLights = eventTime;//记录时间
26.                mDirty |=DIRTY_USER_ACTIVITY;
27.                return true;
28.            }
29.        } else {
30.            if (eventTime> mLastUserActivityTime) {
31.                mLastUserActivityTime =eventTime;//记录时间
32.                mDirty |=DIRTY_USER_ACTIVITY;
33.                return true;
34.            }
35.        }
36.    } finally {
37.        Trace.traceEnd(Trace.TRACE_TAG_POWER);
38.    }
39.    return false;
40. }
```

userActivityNoUpdateLocked()方法主要的工作是更新几个内部变量。其中mLastUserActivityTime变量和mLastUserActivityTimeNoChangeLights变量用来记录调用userActivity()方法的时间，mDirty用来记录用户的操作类型，这些变量的值在updatePowerStateLocked()方法中将会作为是否要执行睡眠或唤醒操作的依据。

强制系统进入休眠模式—gotoSleep接口

gotoSleep()接口用来强制系统进入休眠模式。通常当系统一段时间无人操作后，系统将调用gotoSleep()接口来进入休眠模式。大体流程如下：

图2-3 gotoSleep流程

PowerManagerService的gotoSleep()接口主要是调用内部方法goToSleepInternal()来完成其功能。如下：

```
private void goToSleepInternal(longeventTime,intreason,int flags,intuid) {
    synchronized (mLock) {
        if (goToSleepNoUpdateLocked(eventTime,reason, flags, uid)) {
            updatePowerStateLocked();
        }
    }
}

goToSleepInternal()代码的结构和前面的userActivity类似，都是先调用一个内部方法，然后再调用updatePowerStateLocked()方法，我们先看下goToSleepNoUpdateLocked()方法，如下：
private boolean goToSleepNoUpdateLocked(longeventTime,intreason,intflags,intuid) {
    try {
        switch (reason){
            case PowerManager.GO_TO_SLEEP_REASON_DEVICE_ADMIN:
                Slog.i(TAG, "Going to sleep due to deviceadministration policy "
                    + "(uid "+ uid +")...");
                break;
            case PowerManager.GO_TO_SLEEP_REASON_TIMEOUT:
                Slog.i(TAG, "Going to sleep due to screen timeout(uid "+ uid +")...");
```



```
        break;
    case PowerManager.GO_TO_SLEEP_REASON_LID_SWITCH:
        Slog.i(TAG, "Going to sleep due to lid switch (uid"+ uid +")...");
        break;
    case PowerManager.GO_TO_SLEEP_REASON_POWER_BUTTON:
        Slog.i(TAG, "Going to sleep due to power button (uid"+ uid +")...");
        break;
    case PowerManager.GO_TO_SLEEP_REASON_HDMI:
        Slog.i(TAG, "Going to sleep due to HDMI standby (uid"+ uid +")...");
        break;
    default:
        Slog.i(TAG, "Going to sleep by application request(uid "+ uid +")...");
        reason =PowerManager.GO_TO_SLEEP_REASON_APPLICATION;
        break;
    }
    //修改成员变量的值
    mLastSleepTime = eventTime;
    mSandmanSummoned = true;
    setWakefulnessLocked(WAKEFULNESS_DOZING, reason);//设置mWakefulness的值

    // Report the number of wake locks that will be clearedby going to sleep.
    int numWakeLocksCleared=0;
    final int numWakeLocks= mWakeLocks.size();
    for (inti =0; i < numWakeLocks; i++) {
        final WakeLockwakeLock = 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);//打印
    EventLog

    // Skip dozing if requested.
    if ((flags& PowerManager.GO_TO_SLEEP_FLAG_NO_DOZE) !=0) {
        reallyGoToSleepNoUpdateLocked(eventTime, uid);
    }
} finally {
    Trace.traceEnd(Trace.TRACE_TAG_POWER);
}
return true;
}
goToSleepNoUpdateLocked()只是发送了将要休眠的通知，然后修改了成员变量mDirty、
mLastSleepTime、mWakefulness的值。更多的实际工作还是在updatePowerStateLocked()方法中完成。
```

三、控制系统的休眠机制

Android设备的休眠和唤醒主要基于WakeLock机制。WakeLock是一种上锁机制，只要有进程获得了WakeLock锁系统就不会进入休眠。例如，在下载文件或播放音乐时，即使休眠时间到了，系统也不能进行休眠。WakeLock可以设置超时，超时后会自动解锁。应用使用WakeLock功能前，需要先使用new WakeLock()接口创建一个WakeLock类对象，然后调用它的acquire()方法禁止系统休眠，应用完成工作后调用release()方法来恢复休眠机制，否则系统将无法休眠，直到耗光所有电量。WakeLock类中实现acquire()和release()方法实际上是调用了PowerManagerService的acquireWakeLock()和releaseWakeLock()方法。

1、PMS中WakeLock相关接口

acquireWakeLock()方法检查完权限后，调用了内部方法acquireWakeLockInternal()方法，如下：

```
private void acquireWakeLockInternal(IBinder lock,intflags, String tag, String packageName,
    WorkSource ws, String historyTag,int uid, int pid) {
    synchronized (mLock) {
        if(mBlockedUids.contains(newInteger(uid)) && uid != Process.myUid()) {
            WakeLock wakeLock;
            int index= findWakeLockIndexLocked(lock);//检查这个lock是否已经存在
            boolean notifyAcquire;
            if (index>= 0){//lock已经存在
                wakeLock = mWakeLocks.get(index);
                if (!wakeLock.hasSameProperties(flags.tag, ws, uid, pid)) {
                    notifyWakeLockChangingLocked(wakeLock, flags, tag,packageName,
```



```

        uid, pid, ws, historyTag);
        wakeLock.updateProperties(flags, tag, packageName, ws, historyTag, uid, pid);
    }
    notifyAcquire = false;
} else {
    wakeLock = new WakeLock(lock, flags, tag, packageName, ws, historyTag, uid, pid);
    try {
        lock.linkToDeath(wakeLock, 0);
    } catch (RemoteException ex) {
        throw new IllegalArgumentException("Wake lock is already dead.");
    }
    mWakeLocks.add(wakeLock); // 将新建的WakeLock对象加入到mWakeLocks中
    notifyAcquire = true;
}

applyWakeLockFlagsOnAcquireLocked(wakeLock, uid);
mDirty |= DIRTY_WAKE_LOCKS;
updatePowerStateLocked();
if (notifyAcquire) {
    notifyWakeLockAcquiredLocked(wakeLock);
}
}
}

acquireWakeLockInternal()方法的主要工作是创建WakeLock对象并加入到mWakeLocks列表中，这个列表中包含了所有WakeLock对象。但是如果mWakeLocks列表中已经存在具有相同token的WakeLock对象，则只更新其属性值，不会再创建对象，这个token是用户进程调用gotoSleep时传递的参数：用户进程中WakeLock对象。创建或更新WakeLock对象后，接下来调用applyWakeLockFlagsOnAcquireLocked()方法，这个方法只是调用了wakeUpNoUpdateLocked方法，如下：
private boolean wakeUpNoUpdateLocked(long eventTime, int uid) {
    if (YulongFeature.FEATURE_POWERKEY_FORCE_SCREENON) {
        if (eventTime < mLastSleepTime
            || (mWakefulness == WAKEFULNESS_AWAKE && mProximityPositive != true)
            || !mBootCompleted || !mSystemReady) {
            return false;
        }
    } else {
        if (eventTime < mLastSleepTime || mWakefulness == WAKEFULNESS_AWAKE
            || !mBootCompleted || !mSystemReady) {
            return false;
        }
    }
}
if (YulongFeature.FEATURE_POWERKEY_FORCE_SCREENON) {
    if (mProximityPositive == true) {
        mDisplayManagerInternal.setPowerKeyState(1);
    }
}

Trace.traceBegin(Trace.TRACE_TAG_POWER, "wakeUp");
try {
    switch (mWakefulness) {
        case WAKEFULNESS_ASLEEP:
            Slog.i(TAG, "Waking up from sleep (uid "+ uid + ")...");
            break;
        case WAKEFULNESS_DREAMING:
            Slog.i(TAG, "Waking up from dream (uid "+ uid + ")...");
            break;
        case WAKEFULNESS_DOZING:
            Slog.i(TAG, "Waking up from dozing (uid "+ uid + ")...");
            break;
    }

    mLastWakeTime = eventTime;
    setWakefulnessLocked(WAKEFULNESS_AWAKE, 0);

    userActivityNoUpdateLocked(
        eventTime, PowerManager.USER_ACTIVITY_EVENT_OTHER, 0, uid);
} finally {
    Trace.traceEnd(Trace.TRACE_TAG_POWER);
}
return true;
}

```

在这个方法中主要是设置mLastWakeTime、mWakefulness，最后调用userActivityNoUpdateLocked方法设置mLastUserActivityTime的值。我们将acquireWakeLock()方法的流程图简单概括如下：

图3-1 acquireWakeLock流程图

我们再看下PMS的releaseWakeLock()接口，这个接口也是调用PMS的releaseWakeLockInternal()方法，如下：



```
private void releaseWakeLockInternal(IBinder lock,intflags) {
    synchronized (mLock) {
        int index= findWakeLockIndexLocked(lock);
        if (index< 0){
            return;
        }

        if ((flags& PowerManager.RELEASE_FLAG_WAIT_FOR_NO_PROXIMITY) !=0){
            mRequestWaitForNegativeProximity = true;
        }

        wakeLock.mLock.unlinkToDeath(wakeLock, 0);
        removeWakeLockLocked(wakeLock,index);
    }
}
releaseWakeLockInternal()方法首先查找lock在mWakeLocks中的index，然后从mWakeLocks中得到WakeLock对象，最后调用removeWakeLockLocked方法，如下：
private void removeWakeLockLocked(WakeLock wakeLock,intindex) {
    mWakeLocks.remove(index);
    notifyWakeLockReleasedLocked(wakeLock);

    applyWakeLockFlagsOnReleaseLocked(wakeLock);
    mDirty |= DIRTY_WAKE_LOCKS;
    updatePowerStateLocked();
}
removeWakeLockLocked方法首先从mWakeLocks中移除WakeLock对象并发出通知，接着调用applyWakeLockFlagsOnReleaseLocked()，这个方法中只是调用userActivityNoUpdateLocked()方法来把mLastUserActivityTime更新为当前时间，这样当休眠时间到时，系统就会休眠。
```

2、WakeLock的native层实现

我们先回到PowerManagerService的构造方法中，看看是如何创建两个变量mWakeLockSuspendBlocker和mDisplaySuspendBlocker的。

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

从代码中可以看出这两个变量都是调用createSuspendBlocker()方法创建的，只是参数不同，一个是PowerManagerService.WakeLocks，一个是PowerManagerService.Display；方法代码如下：

```
private SuspendBlocker createSuspendBlockerLocked(String name) {
    SuspendBlocker suspendBlocker = new SuspendBlockerImpl(name);
    mSuspendBlockers.add(suspendBlocker);
    return suspendBlocker;
}
```

createSuspendBlockerLocked()方法创建了一个SuspendBlockerImpl对象并返回，因此mWakeLockSuspendBlocker和mDisplaySuspendBlocker变量的类型应该是SuspendBlockerImpl。我们看下它的acquire()和release()方法，流程图如下所示。

图3-2 acuire()方法流程图

详细代码如下：

```
public void acquire() {
    synchronized (this) {
        mReferenceCount+= 1;
        if (mReferenceCount == 1) {
            Trace.asyncTraceBegin(Trace.TRACE_TAG_POWER, mTraceName, 0);
            nativeAcquireSuspendBlocker(mName);
        }
    }
}

@Override
public void release() {
    synchronized (this) {
        mReferenceCount-= 1;
        if (mReferenceCount == 0) {
            nativeReleaseSuspendBlocker(mName);
            Trace.asyncTraceEnd(Trace.TRACE_TAG_POWER, mTraceName, 0);
        } else if (mReferenceCount <0) {
            mReferenceCount = 0;
        }
    }
}
```

SuspendBlockerImpl类中维护了一个计数器，调用acquire()方法时计数器加1，当计数器的值为1时，调用nativeAcquireSuspendBlocker()方法。调用release()方法时计数器减1，当计数器的值为0时，调用nativeReleaseSuspendBlocked()方法。

调用的native层的函数中又分别调用了acquire_wake_lock()函数和release_wake_lock()函数，其实现如下：

从上面两个函数的实现可以看到，都是通过向不同的驱动文件中写数据来实现其功能。这里写的数据就是

前面构造方法中创建变量时传递的参数“PowerManagerService.WakeLocks”和“PowerManagerService.Display”。那么acquire()和release()中使用的文件设备句柄是如何创建的呢？看下initialize_fds()函数，如下：

Initialize_fds()函数先打开NEW_PATHS数组中的文件，不成功再打开OLD_PATHS数组中的设备文件。

因此，Android实现防止系统休眠的功能是通过向设备文件“sys/power/wake_lock”中写数据来完成的，如果写的是“PowerManagerService.WakeLocks”，系统将不能进入休眠状态，但是屏幕会关闭；如果写的是“PowerManagerService.Display”，则屏幕不会关闭。如果系统要恢复休眠，再向设备文件“sys/power/wake_unlock”中写入同样的字符串就OK了。

3、理解updatePowerStateLocked方法

```
private void updatePowerStateLocked() {
    if (!mSystemReady || mDirty == 0) {
        return;
    }
    Trace.traceBegin(Trace.TRACE_TAG_POWER, "updatePowerState");
    try {
        // Phase 0: 更新基本状态
        updateIsPoweredLocked(mDirty); //更新mIsPowered、mPlugType、mBatteryLevel
        updateStayOnLocked(mDirty); //更新mStayOn
        updateScreenBrightnessBoostLocked(mDirty);

        // Phase 1: 更新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mWakeLockSummary
            updateUserActivitySummaryLocked(now, dirtyPhase1); //更新mUserActivitySummary的值
            if (!updateWakefulnessLocked(dirtyPhase1)) { //更新mWakefulness的值
                break;
            }
        }

        // Phase 2: Update display power state. 更新显示设备状态；确定屏幕状态和亮度，并设置到
        // DisplayPowerController对象中。
        boolean displayBecameReady = updateDisplayPowerStateLocked(dirtyPhase2);

        // Phase 3: Update dream state (depends on display ready signal).
        updateDreamLocked(dirtyPhase2, displayBecameReady); //更新屏保状态，是否启动屏保

        // Phase 4: Send notifications, if needed. 发送通知
        if (mDisplayReady) {
            finishWakefulnessChangeLocked();
        }

        // Phase 5: Update suspend blocker.
        updateSuspendBlockerLocked();
    } finally {
        Trace.traceEnd(Trace.TRACE_TAG_POWER);
    }
}
```

(1) 首先调用updateIsPoweredLocked()方法，这个方法主要是通过调用BatteryService的接口来更新几个成员变量的值，如下：

```
mIsPowered = mBatteryManagerInternal.isPowered(BatteryManager.BATTERY_PLUGGED_ANY);
mPlugType = mBatteryManagerInternal.getPlugType();
mBatteryLevel = mBatteryManagerInternal.getBatteryLevel();
mBatteryLevelLow = mBatteryManagerInternal.getBatteryLevelLow();
```

mIsPowered表示是否在充电，mPlugType表示充电的类型，mBatteryLevel表示当前电池电量的等级。

(2) 调用updateStayOnLocked()方法来更新变量mStayOn的值，mStayOn如果为true，屏幕将保持长亮状态。在Setting中可以设置充电时屏幕长亮，如果Setting中设置了该选项，updateStayOnLocked()函数中如果检测到正在充电，会将mStayOn的值设为true。

(3) 接着调用updateScreenBrightnessBoostLocked()方法，这是Android5.1新增加的方法。

DIRTY_SCREEN_BRIGHTNESS_BOOST是5.1新增加的，表示屏幕亮度提高的状态；

(4) 接下来是一个无限for循环，其实这个for循环，最多两次就结束了，后面分析。我们先来看看在循环中调用的updateWakeLockSummaryLocked()方法，这个方法的主要作用是根据PowerManagerService中所有的WakeLock对象的类型，计算一个最终的类型集合，并保存在变量mWakeLockSummary中。不管系统中一共创建了多少个WakeLock对象，一个就足以阻止系统休眠，因此，这里把所有WakeLock对象的状态总结后放到一个变量中。

应用在创建WakeLock对象时，会指定对象的类型，这个类型将作为参数传递到PowerManagerService中。

WakeLock类型有:

PARTIAL_WAKE_LOCK

FULL_WAKE_LOCK

SCREEN_BRIGHT_WAKE_LOCK

SCREEN_DIM_WAKE_LOCK

PROXIMITY_SCREEN_OFF_WAKE_LOCK: Android5.0新增锁, 这个类型并不是用来阻止系统进入休眠, 而是用来打开距离传感器控制屏幕开关的功能。如果应用持有这种类型的WakeLock, 当距离传感器被遮挡时, 屏幕将会关闭。

DOZE_WAKE_LOCK: Android5.0新增锁, 这个类型用来让屏保管理器实现doze模式。

(5) updateUserActivitySummaryLocked()方法, 这个方法根据最后一次调用userActivity()方法的时间, 计算现在是否可以将表示屏幕状态的变量mUserActivitySummary的值设为SCREEN_STATE_DIM或SCREEN_STATE_OFF。如果时间还没到, 则发送一个定时消息MSG_USER_ACTIVITY_TIMEOUT。当处理消息的时间到了以后, 会在消息的处理方法handleUserActivityTimeout()中重新调用updatePowerStateLocked()方法, 再次调用updatePowerStateLocked方法时, 会根据当前状态重新计算mUserActivitySummary的值。

(6) updateWakefulnessLocked()方法, 这个方法是结束循环的关键。

如果它的返回值是true, 表示PowerManagerService的状态发生了变化, 将继续循环, 然后重新调用前面的两个方法updateWakelockSummaryLocked()和updateUserActivitySummaryLocked()方法来更新状态。而第二次调用updateWakefulnessLocked方法时通常会返回false, 跳出循环。

```
private boolean updateWakefulnessLocked(int dirty) {
    boolean changed = false;
    if ((dirty & (DIRTY_WAKE_LOCKS | DIRTY_USER_ACTIVITY | DIRTY_BOOT_COMPLETED
        | DIRTY_WAKEFULNESS | DIRTY_STAY_ON | DIRTY_PROXIMITY_POSITIVE
        | DIRTY_DOCK_STATE)) != 0) {
        if (mWakefulness == WAKEFULNESS_AWAKE && !isBedTimeYetLocked()) {
            if (DEBUG_SPEW) {
                Slog.d(TAG, "updateWakefulnessLocked: Bedtime...");
            }
            final long time = SystemClock uptimeMillis();
            if (shouldNapAtBedTimeLocked()) {
                changed = napNoUpdateLocked(time, Process.SYSTEM_UID);
            } else {
                changed = goToSleepNoUpdateLocked(time,
                    PowerManager.GO_TO_SLEEP_REASON_TIMEOUT, 0, Process.SYSTEM_UID);
            }
        }
    }
    return changed;
}

updateWakefulnessLocked方法中首先判断dirty的值, 如果是第一次调用, 这个条件很容易满足。注意第二个if语句的判断条件, mWakefulness为WAKEFULNESS_AWAKE并且isBedTimeYetLocked()方法返回true时才会执行, 否则方法结束并返回false, 返回false时就会跳出循环。
我们先假定调用的时候mWakefulness为WAKEFULNESS_AEAKE, 下面看看isBedTimeYetLocked方法什么情况下返回true。

private boolean isBedTimeYetLocked() {
    return mBootCompleted && !isBeingKeptAwakeLocked();
}

private boolean isBeingKeptAwakeLocked() {
    return mStayOn
        || mProximityPositive
        || (mWakeLockSummary & WAKE_LOCK_STAY_AWAKE) != 0
        || (mUserActivitySummary & (USER_ACTIVITY_SCREEN_BRIGHT
            | USER_ACTIVITY_SCREEN_DIM)) != 0
        || mScreenBrightnessBoostInProgress;
}

我们看下isBeingKeptAwakeLocked()方法, 如果系统目前不能睡眠, 这个方法返回true, 这几个变量正是前面方法中设置的判断系统是否能够睡眠的变量。
因此, isBedTimeYetLocked方法只有在系统能够进入睡眠的情况下才会返回true。
我们回到updateWakefulnessLocked方法中, 假如系统能够睡眠, 接下来将调用方法shouldNapAtBedTimeLocked(), 这个方法将检查系统有没有设置睡眠时间到启动屏保或者插在Dock上启动屏保。如果设置了将调用napNoUpdateLocked方法, 如果没有设置则调用goToSleepnoUpdateLocked方法。我们看下napNoUpdateLocked方法:

private boolean napNoUpdateLocked(long eventTime, int uid) {
    if (eventTime < mLastWakeTime || mWakefulness != WAKEFULNESS_AWAKE
        || !mBootCompleted || !mSystemReady) {
        return false;
    }

    Trace.traceBegin(Trace.TRACE_TAG_POWER, "nap");
    try {
        mSandmanSummoned = true;
        setWakefulnessLocked(WAKEFULNESS_DREAMING, 0);
    } finally {
        Trace.traceEnd(Trace.TRACE_TAG_POWER);
    }
    return true;
}
```

```
}
private void setWakefulnessLocked(int wakefulness, int reason) {
    if (mWakefulness != wakefulness) {
        finishWakefulnessChangeLocked();

        mWakefulness = wakefulness;
        mWakefulnessChanging = true;
        mDirty |= DIRTY_WAKEFULNESS;
        mNotifier.onWakefulnessChangeStarted(wakefulness, reason);
    }
}
```

从上面代码中可以看到，如果if语句中4项表达式有一项为true，则返回false。但是如果是循环中第一次调用该方法，则4项正常情况下都为false，不会执行到这里。这样就会继续向下执行，就会改变mDirty和mWakefulness的值。mWakefulness的值既然改变了，当循环中第二次调用到该方法时，就会返回false，这样就结束了updatePowerStateLocked方法中的循环。

(7) 结束循环后，接着调用updateDisplayPowerStateLocked()方法。这个方法的主要作用就是根据更新后的mUserActivitySummary的值来确定屏幕的状态和亮度，并设置到DisplayPowerController对象中。

(8) updateDreamLocked()方法，用来启动屏保。

(9) updateSuspendBlockerLocked()方法。来决定系统是休眠还是唤醒。

4、管理显示设备

在更新电源状态的updatePowerStateLocked()方法中，我们看到调用了updateDisplayPowerStateLocked()方法，该方法的主要作用就是更新PowerManagerService中表示显示屏状态的变量mDisplayPowerRequest，以及重新确定屏幕的亮度。注意在这个方法的结尾处调用了mDisplayPowerController的requestPowerState()方法，这个方法的返回值会赋值给mDisplayReady，如果mDisplayReady为false，屏幕是不会关闭的。

这里我们先看下requestPowerState()方法是怎样被调用的，流程图如下：

图3-4 显示控制流程

具体代码如下：

```
public boolean requestPowerState(DisplayPowerRequest request,
    boolean waitForNegativeProximity) {
    synchronized (mLock) {
        boolean changed = false;
        //如果和近距离传感器相关的变量发生变化，则将change设为true
        if (waitForNegativeProximity
            && !mPendingWaitForNegativeProximityLocked) {
            mPendingWaitForNegativeProximityLocked = true;
            changed = true;
        }
        //使用参数更新mPendingRequestLocked对象，如果两者不相同，则将change设为true
        if (mPendingRequestLocked == null) {
            mPendingRequestLocked = new DisplayPowerRequest(request);
            changed = true;
        } else {
            if (!mPendingRequestLocked.equals(request)) {
                mPendingRequestLocked.copyFrom(request);
                changed = true;
            }
        }
        if (changed) { //如果change为true，将mDisplayReadyLocked设为false
            mDisplayReadyLocked = false;
        }

        if (changed && !mPendingRequestChangedLocked) {
            mPendingRequestChangedLocked = true;
            sendUpdatePowerStateLocked(); //发送消息
        }

        return mDisplayReadyLocked; //返回mDisplayReadyLocked
    }
}
```

从上面代码中可以看到，requestPowerState()方法会将参数request和waitForNegativeProximity与DisplayPowerController对象中已有的值比较，如果不同，则更新DisplayPowerController对象中的值，然后返回mDisplayReadyLocked的值，此时为false。这就意味着，如果屏幕状态发生了变化，即使这种变化是要求关闭屏幕，也不会在updateSuspendBlockerLocked()方法中立即关闭屏幕，那么这个关闭屏幕的操作在什么地方呢？

在requestPowerState()方法中最后会调用sendUpdatePowerStateLocked()方法，发送MSG_UPDATE_POWER_STATE消息，消息的处理方法是updatePowerStateLocked()，这个方法的主要作用是开始播放各种打开或关闭屏幕的动画，或者屏幕变亮或变暗的动画，之后才会将mDisplayReadyLocked设为true，这样屏幕就能关闭了。

```
private void updatePowerState() {
    // . . . . . 播放动画
    // Notify the power manager when ready.
    if (ready && mustNotify) {
```



```
// Send state change.
synchronized (mLock) {
    if (!mPendingRequestChangedLocked) {
        mDisplayReadyLocked = true;//将mDisplayReadyLocked设为true
    }
}
sendOnStateChangedWithWakelock();//发送通知
}
}
updatePowerState()方法最后会调用sendOnStateChangedWithWakelock()方法来发送消息，如下：
private void sendOnStateChangedWithWakelock() {
    mCallbacks.acquireSuspendBlocker();
    mHandler.post(mOnStateChangedRunnable);
}

private final Runnable mOnStateChangedRunnable = new Runnable() {
    @Override
    public void run() {
        mCallbacks.onStateChanged();
        mCallbacks.releaseSuspendBlocker();
    }
};
mCallbacks在PowerManagerService中定义了Callback方法，如下：
private final DisplayManagerInternal.DisplayPowerCallbacks mDisplayPowerCallbacks =
    new DisplayManagerInternal.DisplayPowerCallbacks(){
    @Override
    public void onStateChanged() {
        synchronized (mLock) {
            mDirty |=DIRTY_ACTUAL_DISPLAY_POWER_STATE_UPDATED;
            updatePowerStateLocked();
        }
    }
}
```

在onStateChanged()方法中又调用了updatePowerStateLocked()方法，重新开始处理Power系统的状态更新。

四、电池管理服务

1、概述

Android的电池管理功能用于管理电池的充、放电功能。整个电池管理的部分包括Linux电池驱动、Android 电池服务、电池属性和参数、电池曲线优化四个部分。
Linux电池驱动用户和PMIC交互、负责监听电池产生的相关事件，例如低电报警、电量发生变化、高温报警、USB插拔等。
Android电池服务，用来监听内核上报的电池事件，并将最新的电池数据上报给系统，系统收到新数据后会去更新电池显示状态、剩余电量等信息。如果收到过温报警和低电报警，系统会自动触发关机流程，保护电池和机器不受到危害。
整个电池系统的工作流程，即从底层向Framework层上报数据的流程如下：

2、电池服务

1）、电池服务的启动和运行流程：



```
    初始化本地电池数据结构，将power_supply路径下属性节点路径填充进去，
    └─ BatteryMonitor.h
    └─ BatteryPropertiesRegistrar.cpp
        创建电池属性监听器，并将其注册到Android的系统服务中
    └─ BatteryPropertiesRegistrar.h
```

3、BatteryService类的作用

在PowerManagerService中调用了BatteryService类的一些接口来获得电池的状态，下面来看看BatteryService是如何获得电池状态数据的。

```
public BatteryService(Context context) {
    super(context);
    mBatteryStats =BatteryStatsService.getService();//获得BatteryStatsService对象
    //读取系统设定的各种低电量报警值
    // 监听设备文件: 无效的充电设备
    if(newFile("/sys/devices/virtual/switch/invalid_charger/state").exists()) {
        mInvalidChargerObserver.startObserving(
            "DEVPATH=/devices/virtual/switch/invalid_charger");
    }
}

public void onStart() {
    IBinder b =ServiceManager.getService("batteryproperties");
    final IBatteryPropertiesRegistrar batteryPropertiesRegistrar =
        IBatteryPropertiesRegistrar.Stub.asInterface(b);

    try {
        batteryPropertiesRegistrar.registerListener(new BatteryListener());
    } catch (RemoteException e) {
        // Should never happen.
    }
}
```

1、BatteryService的构造方法首先获得BatteryStatsService的对象，BatteryStatsService主要的功能是收集系统中各个模块和进程的耗电情况。通过BatteryStatsService记录的数据，可以找到耗电量大的模块然后进行分析加以改进。

2、接下来读取了系统设定的各种低电量报警值，如下：

! mCriticalBatteryLevel: 表示电量严重不足时的值，低于这个值系统将关闭；

! mLowBatteryWarningLevel: 表示电量不足时的值，低于这个值系统将发出警告；

! mLowBatteryCloseWarningLevel: 表示停止电量不足警告的值。电量高于这个值后系统将停止电量不足的警告；

! mShutdownBatteryTemperature: 表示电池温度太高的值，高于这个温度系统将关机。

3、再接下来就是创建mInvalidChargerObserver对象，这个对象是一个用于监听UEvent事件的对象，这里主要用于监听设备插入了无效充电器的事件，事件发生时调用该对象的onUEvent设备文件方法，如下：

```
private final UEventObserver mInvalidChargerObserver = new UEventObserver() {
    @Override
    public void onUEvent(UEventObserver.UEvent event) {
        final int invalidCharger = "1".equals(event.get("SWITCH_STATE")) ? 1 : 0;
        synchronized (mLock) {
            if (mInvalidCharger != invalidCharger) {
                mInvalidCharger = invalidCharger;
            }
        }
    }
};
```

onUEvent()方法中将设置BatteryService的成员变量mInvalidCharger的值。这样外界通过BatteryService就能查询充电器是否匹配。通过BatteryManagerInternal类的getInvalidCharger()方法可获取。

```
public int getInvalidCharger() {
    synchronized (mLock) {
        return mInvalidCharger;
    }
}
```

4、在onStart方法中，创建了一个BatteryListener对象，并加入到batteryproperties服务的回调接口中。我们先看下BatteryListener类的定义：

```
private final class BatteryListener extends IBatteryPropertiesListener.Stub {
    @Override
    public void batteryPropertiesChanged(BatteryProperties props) {
        final long identity = Binder.clearCallingIdentity();
        try {
            BatteryService.this.update(props);
        } finally {
            Binder.restoreCallingIdentity(identity);
        }
    }
}
```

BatteryListener是一个Binder服务类，因此，batteryproperties服务可以通过它传递数据。这里传递回来的数据类型是BatteryProperties，如下：

```
public class BatteryProperties implements Parcelable {
    public boolean chargerAcOnline; //正在用AC充电器充电
```



```
public boolean chargerUsbOnline;//正在用USB充电器充电
public boolean chargerWirelessOnline;//正在用无线充电器充电
public int batteryStatus;//电池状态值
public int batteryHealth;//电池的健康值
public boolean batteryPresent;//设备是否在使用电池供电
public int batteryLevel;//电量级别
public int batteryVoltage;//电压值
public int batteryTemperature;//电池温度
public String batteryTechnology;//电池的制造商信息
```

4、Healthd守护进程

BatteryService中使用的batteryproperties服务位于healthd守护进程中，healthd在init.rc中的定义如下：
healthd服务位于core分組，系统初始化的时候就会启动。代码位于目录/system/core/healthd下，看下入口函数：

```
int main(int argc, char **argv) {
    . . . . . //解析参数

    ret = healthd_init();
    . . . . . //错误处理
    healthd_mainloop();//进入主循环
    KLOG_ERROR("Main loop terminated, exiting\n");
    return 3;
}

我们看下healthd_init()函数都进行了哪些初始化操作，如下：
static int healthd_init() {
    . . . . .

    wakealarm_init();//创建一个定时器
    uevent_init();//初始化uevent环境
    gBatteryMonitor = new BatteryMonitor();//创建BatteryMonitor对象
    gBatteryMonitor->init(&healthd_config);
    return 0;
}
```

在healthd_init函数中调用wakealarm_init()函数的目的是创建一个定时器的文件句柄，如下：

```
static void wakealarm_init(void) {
    wakealarm_fd = timerfd_create(CLOCK_BOOTTIME_ALARM, TFD_NONBLOCK);
    . . . . .

    //设置时间
    wakealarm_set_interval(healthd_config.periodic_chores_interval_fast);
}
```

Wakealarm_init()函数调用timerfd_create()函数创建了一个定时器的文件句柄，并保存在全局变量wakealarm_fd中，最后通过wakealarm_set_interval()来设置定时器的超时时间。

```
static void healthd_mainloop(void) {
    while (1) {
        . . . . .

        nevents = epoll_wait(epollfd, events, eventct, timeout);
        . . . . .

        for (int n = 0; n < nevents; ++n) {
            if (events[n].data.ptr)
                (*(void (*)(int))events[n].data.ptr)(events[n].events);//调用事件的处理函数
        }
        if (!nevents)//nevents为0表示没有无Event，调用periodic_chores()
            periodic_chores();
        healthd_mode_ops->heartbeat();
    }
    return;
```

最终都会调用到healthd_battery_update()函数，该函数中调用gBatteryMonitor的update()函数来读取电池的状态，然后根据返回值来决定是否更新定时器的时间周期。这样在定时器的作用下，系统中电池的状态会持续更新。

5、读取电池的各种参数----BatteryMonitor类

BatteryMonitor的作用是从设备文件中读取电池的各种参数并返回给上层应用。在healthd的main()函数中会创建BatteryMonitor对象，并调用它的init()函数。

Init()函数首先打开/sys/class/power_supply，该目录下包含一些子目录，其中battery目录下存放的是电池信息，这个目录下的每个文件对应电池的一种属性，文件的内容就是各个属性的当前值。Usb目录表示USB充电器的信息，目录下的online文件的内容为1表示正在用usb充电。


Init()函数的主要功能就是生成所有这些文件的文件名并保存到成会变量中，方便以后读取。

BatteryMonitor类的update()函数用来读取电池的信息，在healthd的healthd_mainloop()函数中会周期性低调用这个函数。




android 之 PowerManager 与 电源管理 (http://blog.csdn.net/xieqibao/article/details/6562...

PowerManager这个类主要是用来控制电源状态的. 通过使用该类提供的api可以控制电池的待机时间, 一般情况下不要使用。如果确实需要使用, 那么尽可能的使用最低级别的WakeLocks锁。并且确保...

 xieqibao (http://blog.csdn.net/xieqibao) 2011年06月22日 22:48 23522


Android电源管理机制剖析 (http://blog.csdn.net/galensphang/article/details/9943805)

Android 的电源管理也是很重要的一部分。比如在待机的时候关掉不用的设备, timeout之后的屏幕和键盘背光的关闭, 用户操作的时候该打开多少设备等等, 这些都直接关系到产品的待机时间, 以及用户体验。...

 galensphang (http://blog.csdn.net/galensphang) 2013年08月13日 18:46 5829


Android电源管理机制的实现 (http://blog.csdn.net/incanus/article/details/8863579)

从电源模块的初始化函数(pm_init)开始分析: /kernel/power/main.c static int __init pm_init(void) { int error = pm...

 incanus (http://blog.csdn.net/incanus) 2013年04月28日 15:29 1996


Android电源管理－Healthd (http://blog.csdn.net/u012296694/article/details/45420495)

OS: Android 4.4.2 Android电源管理底层用的是Linux power supply框架。驱动部分不叙述。只看JAVA、JNI和CPP应用层。从Android 4.4开...

 u012296694 (http://blog.csdn.net/u012296694) 2015年05月01日 17:29 2353


Android 电源管理 -- wakelock机制 (http://blog.csdn.net/wh_19910525/article/details/828...

Wake Lock是一种锁的机制, 只要有人拿着这个锁,系统就无法进入休眠, 可以被用户态程序和内核获得. 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...

 wh_19910525 (http://blog.csdn.net/wh_19910525) 2012年12月12日 19:10 23299


Android电源管理框架 (/qq_695538007/article/details/41208135)

一、wakelock机制 Android电源管理使用的wakelock机制: 系统中有激活(未释放)的wakelock存在, 系统就不能进入休眠状态; 如果系统没有激活的wakelock存在, 则系统立即进...

 hongwazi_2010 (http://blog.csdn.net/hongwazi_2010) 2014-11-17 14:03 1598


Android Battery(四) 电池管理 (/wds1181977/article/details/51142590)

欢迎转载, 转载请注明出处: http://blog.csdn.net/zhgxhuuaa 说明本篇将介绍省电管理篇, 主要介绍一下Android的耗电情况和目前市面上《...

 wds1181977 (http://blog.csdn.net/wds1181977) 2016-04-13 13:44 2633


Android 5.1长按电源键添加重启功能 (/zhoumushui/article/details/51722646)

现在长按Power键只有一个关关键, 需要添加一个重启, 以下是我的添加步骤: 1.在frameworks/base/core/res/res/values/config.xml里添加重启: ...

 zhoumushui (http://blog.csdn.net/zhoumushui) 2016-06-20 18:44 1480


android5.1 按下 power键 系统不休眠 (/xishuang_gongzi/article/details/52857669)

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

 xishuang_gongzi (http://blog.csdn.net/xishuang_gongzi) 2016-10-19 11:11 876


android5.1添加android 长按 power键重启功能 (/kc58236582/article/details/45866617)

当用户长按power键的时候, 系统会在PhoneWindowManager中调用 mGlobalActions.showDialog, 来显示关机、飞行、重启等界面选项。而我们需要在Global...

 kc58236582 (http://blog.csdn.net/kc58236582) 2015-05-20 09:26 3156

Android7.0 PowerManagerService(1) 启动过程 (/gaugamela/article/details/52785041)



PowerManagerService的启动过程

 Gaugamela (http://blog.csdn.net/Gaugamela) 2016-10-12 14:19 3609





android6.0 PowerManagerService状态分析 (/kc58236582/article/details/51595390)

这篇博客我们主要分析下PowerManagerService的各个状态，主要从goToSleep，wakeUp，userActivity，nap函数作为入口分析。一、PowerManagerServ...

 kc58236582 (http://blog.csdn.net/kc58236582) 2016-06-06 20:04  6766



Android提供的系统服务之--PowerManager(电源服务) (/u014450015/article/details/5053...

Android提供的系统服务之--PowerManager(电源服务) &...

 u014450015 (http://blog.csdn.net/u014450015) 2016-01-18 23:57  360



android的 PowerManager和 PowerManager.WakeLock (/peidonghui/article/details/8175146)

前言 学习android一段时间了，为了进一步了解android的应用是如何设计开发的，决定详细研究几个开源的android应用。从一些开源应用中吸收点东西，一边进行量的积累，一边探索andro...

 peidonghui (http://blog.csdn.net/peidonghui) 2012-11-12 16:58  6741

Android 电池管理系统 (/bhj1119/article/details/52947344)

一、Android 电池服务 Android电池服务，用来监听内核上报的电池事件，并将最新的电池数据上报给系统，系统收到新数据后会去更新电池显示状态、剩余电量等信息。如果收到过温报警和低压报警，系统会...

 BHJ1119 (http://blog.csdn.net/BHJ1119) 2016-10-27 16:48  1375