android——PowerManagerService源码分析

作者：King1425 (/authorarticle.html?author=King1425) 2017-04-18 ☆ 收录到我的专题 (/select.html?articleId=370783)

23

August

电源管理架构

Android电源管理主要是通过wakelock机制来管理系统的状态，整个android电源管理可以分为四层：

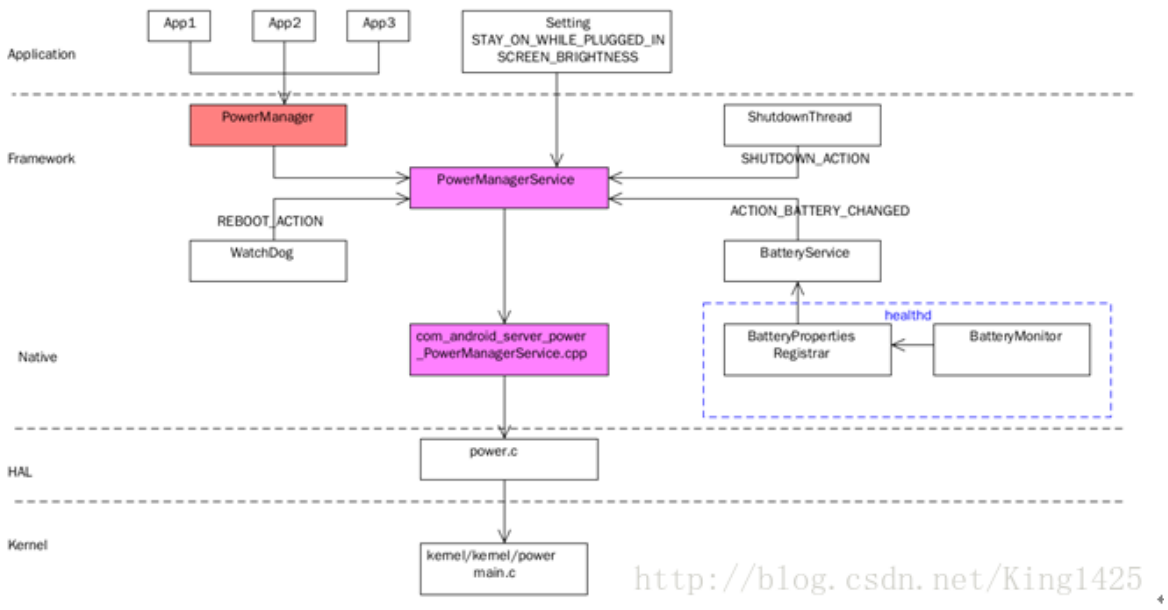
应用接口层:PowerManager中开放给应用一系列接口，应用可以调用PM的接口申请wakelock，唤醒系统，使系统进入睡眠等操作；

Framework层：应用调用PowerManager开放的接口，来对系统进行一些列的操作是在PowerManagerService中完成的，PowerManagerService计算系统中和Power相关的计算，是整个电源管理的决策系统。同时协调Power如何与系统其它模块的交互，比如亮屏，暗屏，系统睡眠，唤醒等等。

HAL层：该层只有一个power.c文件，该文件通过上层传下来的参数，向/sys/power/wake_lock或者/sys/power/wake_unlock文件节点写数据来与kernel进行通信，主要功能是申请/释放锁，维持屏幕亮灭。

Kernel层：内核层实现电源管理的方案主要包含三个部分：
1、Kernel/power/：实现了系统电源管理框架机制。
2、Arch/arm(ormips or powerpc)/mach-XXX/pm.c：实现对特定板的处理器电源管理。
3、drivers/power：是设备电源管理的基础框架，为驱动提供了电源管理接口。

Android电源管理框架如下图：



电源管理服务——PowerManagerService

PowerManagerServcie是android系统电源管理的核心服务，它在Framework层建立起一个策略控制方案，向下决策HAL层以及kernel层来控制设备待机状态，控制显示屏，背光灯，距离传感器，光线传感器等硬件设备的状态。向上提供给应用程序相应的操作接口，比如听音乐时持续保持系统唤醒，应用通知来临唤醒手机屏幕等场景

启动过程

SystemServer在系统启动的时候会启动三类服务：引导关键服务，核心服务，其他服务；PowerManagerService是在SystemServer中创建的，并将其作为一个系统服务加入到ServiceManager中：

```
mPowerManagerService = mSystemServiceManager.startService(PowerMan
```

agerService.class);

在启动引导关键服务调用startBootstrapServices()，其中各种服务都是通过SystemServiceManager中的startService()函数来启动：

```
1 public <T extends SystemService> T startService(Class<T> serviceClass) {
2 final String name = serviceClass.getName();
3 final T service;
4 Constructor<T> constructor = serviceClass.getConstructor(Context.class);
5 service = constructor.newInstance(mContext);
6 mServices.add(service);//注册服务到服务列表中去
7 service.onStart();//启动服务
```

在启动PowerManagerService时，传入的参数类是PowerManagerService，在startService()中首先调用PowerManagerService的构造函数，然后调用其onStart()函数
PowerManagerServcie的构造函数：

```
1 mHandler = new PowerManagerHandler(mHandlerThread.getLooper());
2
3 synchronized (mLock) {
4 mWakeLockSuspendBlocker = createSuspendBlockerLocked("PowerManagerService.Wa
5 keLocks");
6 mDisplaySuspendBlocker = createSuspendBlockerLocked("PowerManagerService.Displ
7 ay");
8 mDisplaySuspendBlocker.acquire();
  mWakefulness = WAKEFULNESS_AWAKE;
```

nativeInit();
创建一个处理消息和发送消息的线程mHandler，并且两种标志flag的wakelock锁：PowerManagerService.WakeLocks和PowerManagerService.Display，前者是传入到底层是控制cpu唤醒状态，后者则是控制屏幕亮灭。在构造函数最后调用nativeInit();在native层初始化相关资源。将mWakefulness 置成WAKEFULNESS_AWAKE状态，mWakefulness 标识系统当前状态共有四种定义：
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。

在SystemServer中startService中调用到PowerManagerService构造函数做完初始化操作之后便会调用PowerManagerService的onStart()函数：

```
1 publishBinderService(Context.POWER_SERVICE, new BinderService());
2 publishLocalService(PowerManagerInternal.class, new LocalService());
3 Watchdog.getInstance().addMonitor(this);
4 Watchdog.getInstance().addThread(mHandler);
```

Onstart完成的工作就是将POWER_SERVICE作为Binder的服务端，注册到SystemService中去；将PowerManagerInternal注册到本地服务中，将自己加到watchdog的监控队列中去;将之前在构造函数中创建的mHandler对象加入到watchdog的中，用于监视mHandler的looper是否空闲；

系统准备工作

SystemServer在调用PowerManagerService之后还会调用其SystemReady，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：

```
1 mAppOps = appOps;
2 mDreamManager = getLocalService(DreamManagerInternal.class);
3 mDisplayManagerInternal = getLocalService(DisplayManagerInternal.class);
4 mPolicy = getLocalService(WindowManagerPolicy.class);
5 mBatteryManagerInternal = getLocalService(BatteryManagerInternal.class);
6 PowerManager pm = (PowerManager) mContext.getSystemService(Context.POWER_SER
7 VICE);
8 mScreenBrightnessSettingMinimum = pm.getMinimumScreenBrightnessSetting();
9 mScreenBrightnessSettingMaximum = pm.getMaximumScreenBrightnessSetting();
10 mScreenBrightnessSettingDefault = pm.getDefaultScreenBrightnessSetting();
11
```

获取与PowerManager相关的本地服务，比如屏保（mDreamManager），屏幕显示（mDisplayManagerInternal），窗口策略（mPolicy），电池电量（mBatteryManagerInternal）等服务

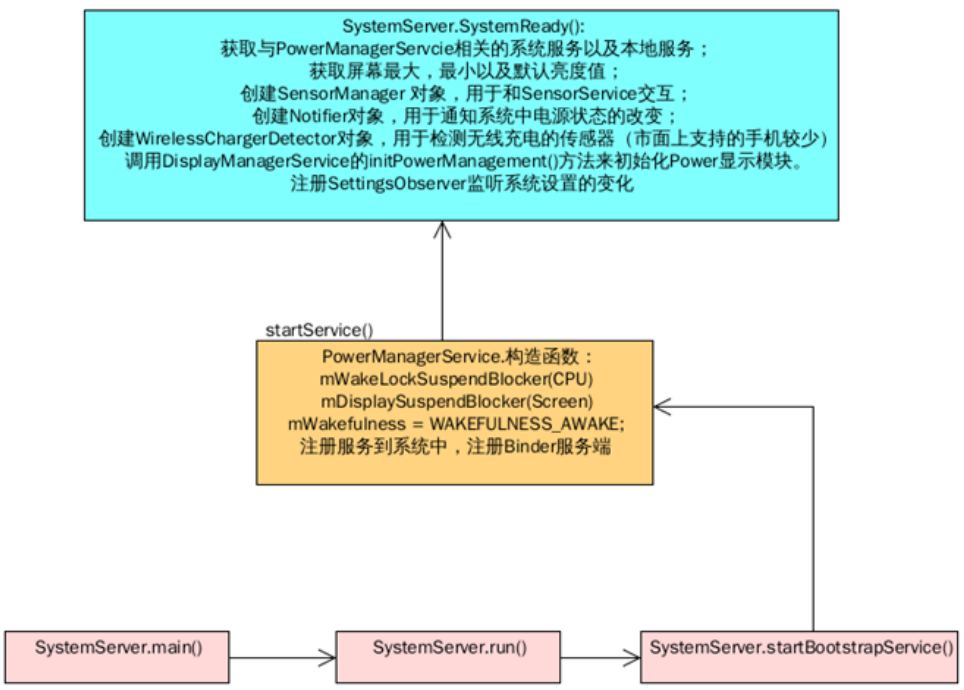
，然后初始化屏幕最大亮度，最小亮度，和默认亮度；

```
SensorManager sensorManager = new SystemSensorManager(mContext, mHandler.get
Looper());
mBatteryStats = BatteryStatsService.getService();
mNotifier = new Notifier(Looper.getMainLooper(), mContext, mBatteryStats,
mAppOps, createSuspendBlockerLocked("PowerManagerService.Broadcasts"),
mPolicy);
mWirelessChargerDetector = new WirelessChargerDetector(sensorManager,
createSuspendBlockerLocked("PowerManagerService.WirelessChargerDetector"), mHa
ndler);
mSettingsObserver = new SettingsObserver(mHandler);
mLightsManager = getLocalService(LightsManager.class);
mAttentionLight = mLightsManager.getLight(LightsManager.LIGHT_ID_ATTENTION);
```

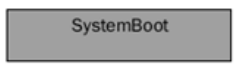
创建sensorManager 对象，用于与sensor交互，比如距离传感器，光线传感器，加速度传感器（doze上使用）。获取电池状态服务，和背光服务；
创建mNotifier 对象，在通过mNotifier 发送通知时候，会传入底层申请PowerManagerService.Broadcasts的wakelock锁。
创建mSettingsObserver 监听系统设置变化，比如亮屏时间，自动背光，屏幕亮度，屏保，低电模式等等

总而言之在SystemReady方法中完成的主要工作如下：
获取与PowerManagerServcie相关的系统服务以及本地服务；
获取屏幕最大，最小以及默认亮度值；
创建SensorManager 对象，用于和SensorService交互；
创建Notifier对象，用于通知系统中电源状态的改变；
创建WirelessChargerDetector对象，用于检测无线充电的传感器（市面上支持的手机较少）
调用DisplayManagerService的initPowerManagement()方法来初始化Power显示模块。
注册SettingsObserver监听系统设置的变化

PowerManagerServcie的启动初始化过程如下：



==========

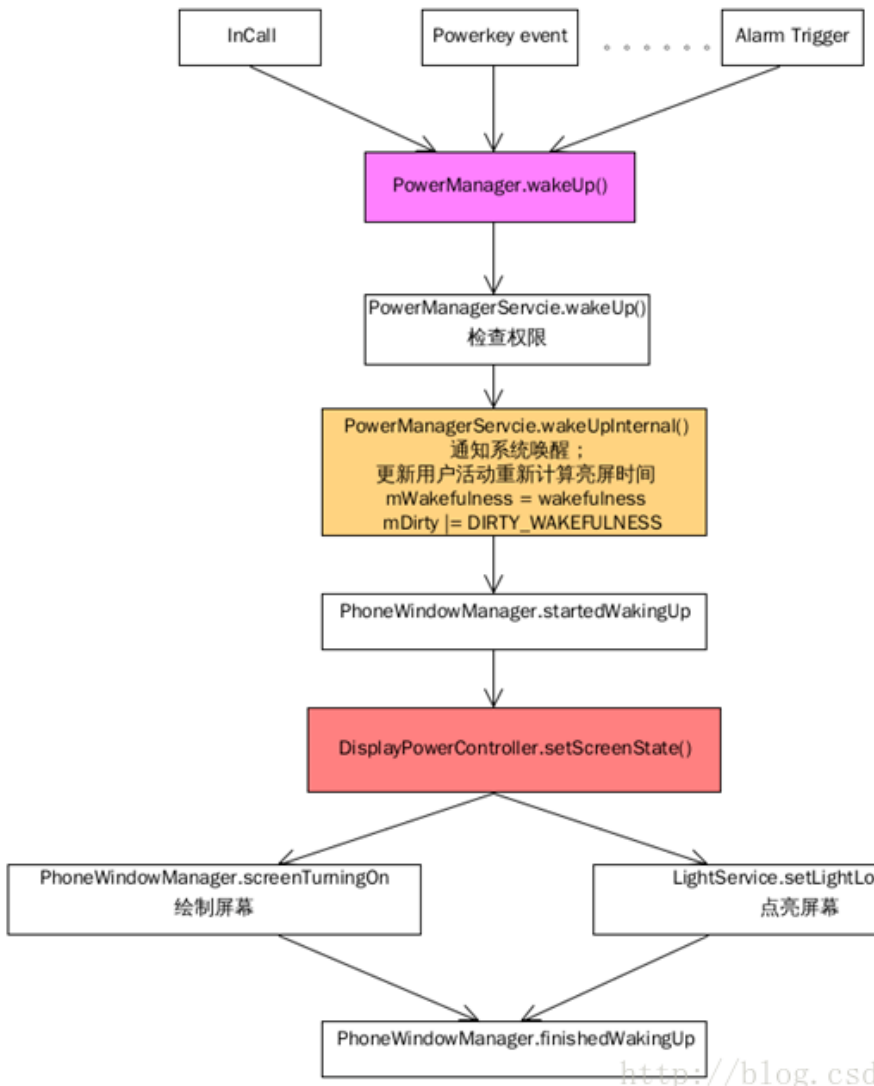


public void onBootPhase(int phase) {}阶段

相关接口
PowerManager向应用提供了相应的接口，以供应用程序调用，来改变系统待机状态，屏幕状态，屏幕亮度等，PowerManager是PowerManagerService的代理类，PowerManager向上层应用提供交互的接口，具体的处理工作在PowerManagerService中完成。下面介绍PowerManager中提供的相应接口作用：
Wakeup()：强制系统从睡眠状态唤醒，此接口对应用是不开放的，应用想唤醒系统必须通过设置亮屏标志（后面即将讲到）；
gotoSleep()：强制系统进入到睡眠状态，此接口也是应用不开放。
userActivity()：向PowerManagerService报告影响系统休眠的用户活动，重计算灭屏时间，背光亮度等，例如触屏，划屏，power键等用户活动；
Wakelock：wakelock是PowerManager的一个内部类，提供了相关的接口来操作wakelock锁，比如newWakeLock()方法来创建wakelock锁，acquire()和release()方法来申请和释放锁。
isDeviceIdleMode()：返回设备当前的状态，如果处于Idle状态，则返回true，Idle状态是在手机长时间没有被使用以及没有运动的情况下，手机进入到一种Doze低功耗的模式下，这种状态下手机可能会关掉网络数据访问，可以通过监视DEVICE_IDLE_MODE_CHANGED这个广播信息，来监控手机状态的改变

唤醒——wakeup

PowerManager的wakeup接口属性是@hide的，所以对于上层应用是不可见的，上层应用要唤醒系统大都依靠两种方式：1.在应用启动Activity时候设置相应的window的flags，通过WMS来唤醒系统；2.在应用申请wakelock锁时附带ACQUIRE_CAUSES_WAKEUP标志；
Wakeup流程如下图所示



PowerManager的wakeup接口，可供应用程序调用，来强制唤醒系统，如果该设备处于睡眠状态，调用该接口会立即唤醒系统，比如按Power键，来电，闹钟等场景都会调用该接口。唤醒系统需要android.Manifest.permission#DEVICE_POWER的权限；
我们来看看PowerManagerServcie中wakeup接口的代码：

```
1  mContext.enforceCallingOrSelfPermission(android.Manifest.permission.DEVICE_POWE
2  R, null);
3  final int uid = Binder.getCallingUid();
4  final long ident = Binder.clearCallingIdentity();
5
6  wakeUpInternal(eventTime, reason, uid, opPackageName, uid);
```

Wakeup接口中仅仅是对调用者的权限进行检查；然后放到wakeUpInterna1()中处理，wakeUpInterna1()中没有做操作，只是调用wakeUpNoUpdateLocked()函数，然后更新调用updatePowerStateLocked()更新电源状态
wakeUpNoUpdateLocked()关键代码

```
1  mLastWakeTime = eventTime;
2  setWakefulnessLocked(WAKEFULNESS_AWAKE, 0);
3
4  mNotifier.onWakeUp(reason, reasonUid, opPackageName, opUid);
5  userActivityNoUpdateLocked(
6  eventTime, PowerManager.USER_ACTIVITY_EVENT_OTHER, 0, reasonUid);
7
8  return true;
```

setWakefu1nessLocked()函数将mWakefu1ness赋值为wakefu1ness和mDirty |= DIRTY_WAKEFULNESS;这两个标志在后面更新电源状态时有重要作用，同时调用到mNotifier中的onWakefu1nessChangeStarted调用到hand1eEar1yInteractiveChange调用到PhoneWindowManager的startedWakingUp函数，来通知到PhoneWindowManager屏幕开始启动；
调用mNotifier向系统中通知系统被唤醒；
更新用户活动，将mDirty |= DIRTY_USER_ACTIVITY置位;来重新计算亮屏时间。

在updatePowerStateLocked()中更新电源状态，updatePowerStateLocked为PowerManagerService的核心函数，后面会详细介绍，这里简单介绍在wakeup中的流程。在updatePowerStateLocked()中的updateDisp1ayPowerStateLocked()函数中将mDisp1ayPowerRequest.po1i

cy设置成POLICY_BRIGHT；然后调用：

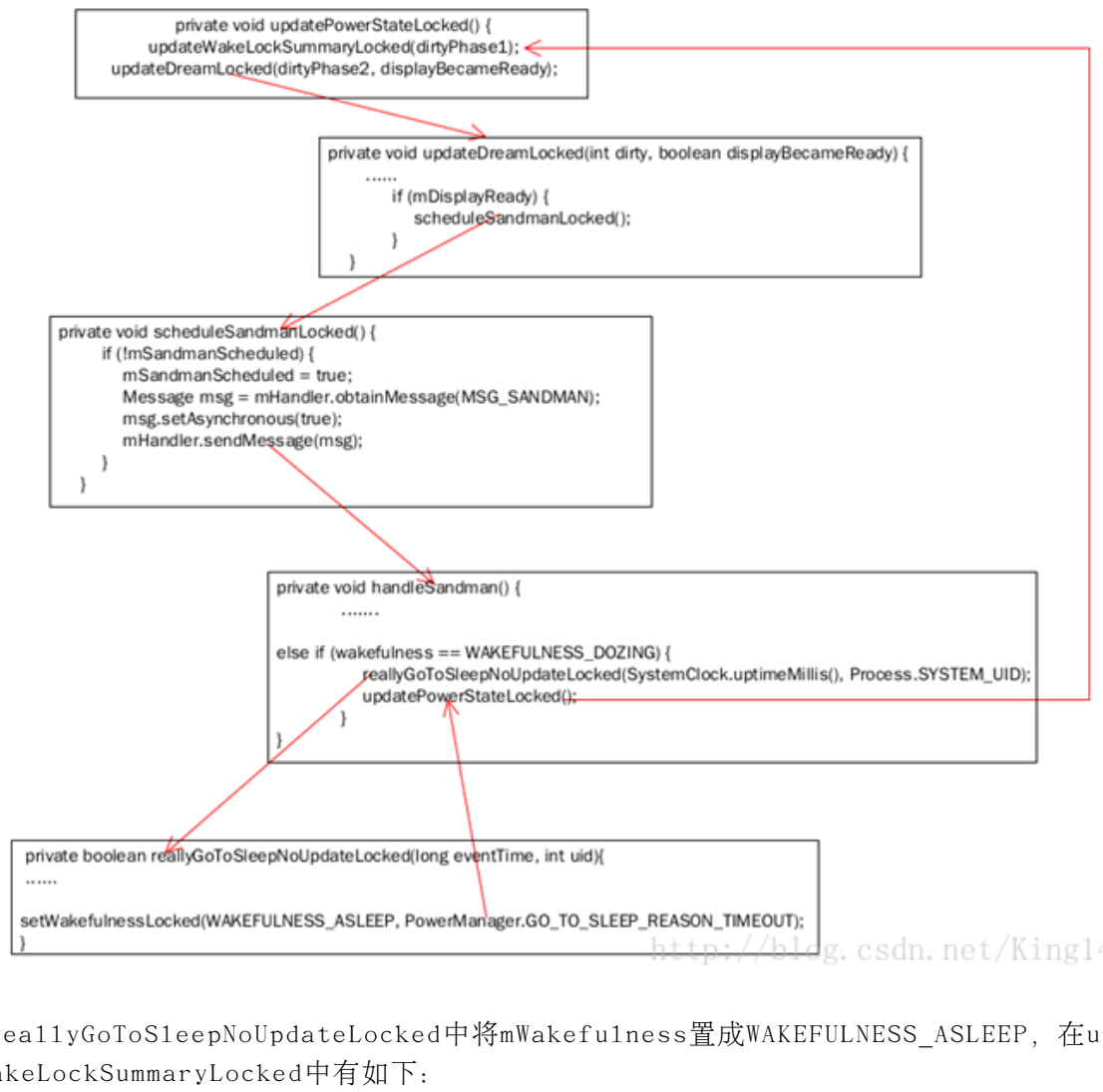
```
1 mDisplayReady = mDisplayManagerInternal.requestPowerState(mDisplayPowerReques
  t,mRequestWaitForNegativeProximity);
```

通过调用到DisplayManagerService中对屏幕状态作出相应的改变，通过去与WindowManager Service执行亮屏之前的屏幕绘制过程，与LightServcie交互来点亮屏幕背光灯。其调用过程和具体逻辑在亮屏流程文档中有详细分析。

睡眠——goToSleep

PowerManager的gotoSleep()接口是@hide属性，因此对于上层应用是不开放的，设备强制进入睡眠状态，在处理一些灭屏按键事件时，会通过WMS来调用PowerManager的gotoSleep接口，一般在系统一段时间没有被操作时，系统将会自动调用gotoSleep函数，让其进入到睡眠模式；
与wakeup唤醒一样，PowerManager的gotoSleep()在PowerManagerService中处理，PMS中的gotoSleep()首先检查调用者是否拥有android.Manifest.permission.DEVICE_POWER权限。然后调用到goToSleepInternal()中处理：
if (goToSleepNoUpdateLocked(eventTime, reason, flags, uid)) {
updatePowerStateLocked();

在goToSleepNoUpdateLocked()中完成发送了将要休眠的通知，然后修改了Wakefulness，将其置成WAKEFULNESS_DOZING，将mDirty |= DIRTY_WAKEFULNESS置位，更多的实际工作在updatePowerStateLocked()中完成。在updateDreamLocked中完成真正进入睡眠的过程；
其调用过程大致如下：



在reallyGoToSleepNoUpdateLocked中将mWakefulness置成WAKEFULNESS_ASLEEP，在updateWakeLockSummaryLocked中有如下：

```
1 if (mWakefulness == WAKEFULNESS_ASLEEP
2 || (mWakeLockSummary & WAKE_LOCK_DOZE) != 0) {
3 mWakeLockSummary &= ~(WAKE_LOCK_SCREEN_BRIGHT | WAKE_LOCK_SCREEN_DIM
4 | WAKE_LOCK_BUTTON_BRIGHT);
```

将mWakeLockSummary列表中的wakelock锁所形成的集合变量mWakeLockSummary 中将WAKE_LOCK_SCREEN_BRIGHT，WAKE_LOCK_SCREEN_DIM，WAKE_LOCK_BUTTON_BRIGHT三种wakelock锁置为无效，再次调用updatePowerStateLocked更新电源状态时候，会在updateDisplayPowerStateLocked中做灭屏操作，其流程与wakeup唤醒系统亮屏操作流程大致一样。

用户活动——userActivity
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。
在userActivityInternal()中并没有做任何操作，仅仅是将mLastUserActivityTime 更新为当前event的时间eventTime，mDirty |= DIRTY_USER_ACTIVITY;置位操作。具体操作仍然是在PowerManagerService中的核心函数updatePowerStateLocked()中完成；

在updateUserActivitySummaryLocked()中

```
final int sleepTimeout = getSleepTimeoutLocked();
final int screenOffTimeout = getScreenOffTimeoutLocked(sleepTimeout);
final int screenDimDuration = getScreenDimDurationLocked(screenOffTimeout);

mUserActivitySummary = 0;
if (mLastUserActivityTime >= mLastWakeTime) {
nextTimeout = mLastUserActivityTime
+ screenOffTimeout - screenDimDuration;
if (now < nextTimeout) {
mUserActivitySummary = USER_ACTIVITY_SCREEN_BRIGHT;
} else {
nextTimeout = mLastUserActivityTime + screenOffTimeout;
if (now < nextTimeout) {
mUserActivitySummary = USER_ACTIVITY_SCREEN_DIM;
}
}
```

重新计算睡眠超时时间，灭屏超时时间，暗屏超时时间，将mUserActivitySummary 置为0，通过计算上一次的用户事件时间与超时时间作对比，来判断将屏幕置为亮屏 (USER_ACTIVITY_SCREEN_BRIGHT)还是暗屏(USER_ACTIVITY_SCREEN_DIM)，前提是手机处于非睡眠状态。

```
if (mUserActivitySummary != 0 && nextTimeout >= 0) {
Message msg = mHandler.obtainMessage(MSG_USER_ACTIVITY_TIMEOUT);
msg.setAsynchronous(true);
mHandler.sendMessageAtTime(msg, nextTimeout);
}
```

如果时间还没到，则返回发送一个MSG_USER_ACTIVITY_TIMEOUT的定时消息，当处理时间到了，会在消息的处理方法handleUserActivityTimeout中重新调用updatePowerStateLocked()电源状态。再次调用时会根据当前的状态重新计算mUserActivitySummary 的值。

控制系统休眠
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。
updatePowerStateLocked为PowerManagerService的核心函数；在执行完申请锁，释放锁，用户事件，强制唤醒/睡眠等操作都需要调用updatePowerStateLocked()来更新电源状态，

wakelock
Wakelock是android系统上特有的电源管理机制，只要有应用拿着这个锁，系统就不能进入睡眠状态，在上层不同的应用程序可以持有多个不同的wakelock锁，但是反映到底层就只有三种：控制系统休眠PowerManagerService.WakeLock，控制屏幕显示的PowerManagerService.Display和控制电源状态改变通知的PowerManagerService.Broadcasts。
PowerManagerService有加锁和解锁两种状态，加锁有两种方式：
第一种是永久的锁住，这样的锁除非显式的放开，否则是不会解锁的，所以这种锁用起来要非常的小心（默认）。
第二种锁是超时锁，这种锁会在锁住后一段时间解锁。

相关接口
newWakeLock()：创建wakelock锁，当外界创建wakelock之前需要创建PowerManager的服务对象，然后其创建wakelock锁。
setReferenceCounted()设置计数锁和非计数锁；wakelock分为计数锁和非计数锁两种：计数锁是应用调用一次acquire申请必定会对应一个release来释放；非计数锁应用调用多次acquire，调用一次release就可释放前面acquire的锁。在申请wakelock时默认申请的是计数锁。
isHeld()判断一个wakelock锁是否acquire申请了，但是没有release释放；
acquire()和release()方法来申请和获取锁，acquire申请锁有两种：
acquire()：申请wakelock永久锁（默认），需要手动release
acquire(long timeout) ：申请wakelock超时锁，timeout为设置的超时时间，超时自动release掉该wakelock。

Andoid的控制系统休眠是用wakelock机制，应用程序在使用wakelock前，必须在其manifest.xml文件中注册android.permission.WAKE_LOCK权限，应用要使用wakelock，需先调用newWakeLock()创建wakelock，然后acquire()申请该锁，从而阻止系统休眠，在处理完事物之后要及时调用release()来释放wakelock，否则系统始终无法进入睡眠状态，直到电量耗光。

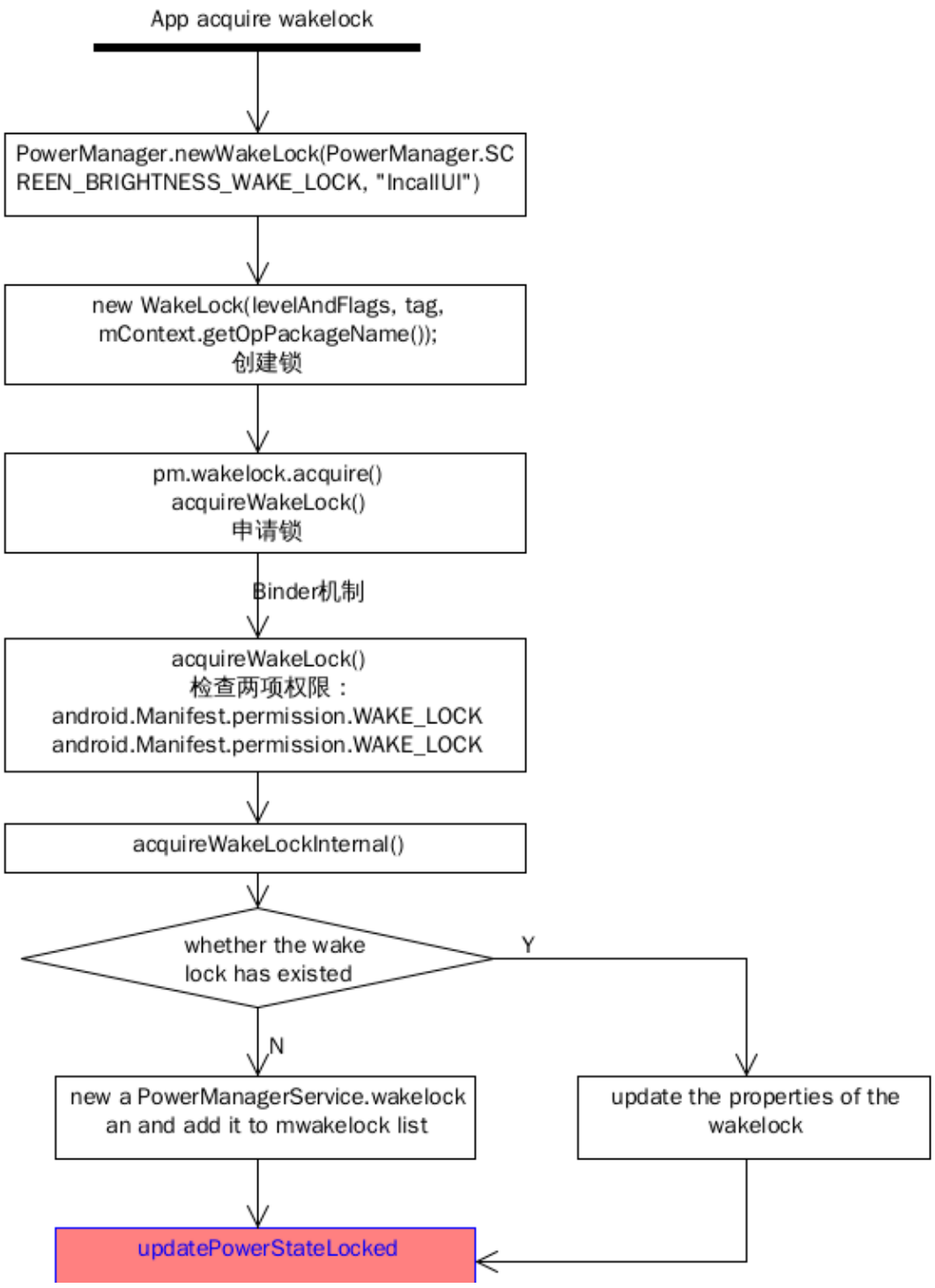
Wakelock类型

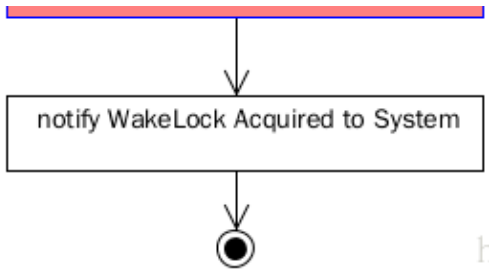
锁类型	cpu	screen	Keyboard	电源键影响	应用情景	备注
PARTIAL_WAKE_LOCK = 0x00000001	On	Off	Off	不受	听音乐，后台下载等	
~~SCREEN_DIM_WAKE_LOCK~~ = 0x00000006	On	Dim	Off	受	即将进入灭屏休眠状态时	这三种类型的锁，在 6.0 以后版本会逐渐被抛弃使用，改用 WindowManager.LayoutParams 的一个参数 FLAG_KEEP_SCREEN_ON 来替换上述三种类型的锁,因为它将由平台被准确地管理用户应用程序之间的动作,并且不需要特殊的权限。
~~SCREEN_BRIGHT_WAKE_LOCK~~ = 0x0000000a	On	Bright	Off	受	看电子书，看视频，操作屏幕没有操作到键盘等	
~~FULL_WAKE_LOCK~~ = 0x0000001a	On	Bright	On	受	来电话，闹钟触发等	
PROXIMITY_SCREEN_OFF_WAKE_LOCK = 0x00000020	Off	Bright/Off	Off	受	打电话靠近或远离手机时	需要设备支持距离传感器，不能和 ACQUIRE_CAUSES_WAKEUP 一起用
DOZE_WAKE_LOCK = 0x00000040	Off	Off	Off	受	低电状态，doze 模式下，允许 cpu 进入 suspend 状态	系统支持 doze
DRAW_WAKE_LOCK = 0x00000080	On	Off	Off	不受	保持设备唤醒，能正常进行绘图	windowManager 允许应用在 dozing 状态绘制屏幕
ACQUIRE_CAUSES_WAKEUP = 0x10000000	说明：正常情况下，获取 wakelock 是不会唤醒设备的，加上该标志之后，acquire wakelock 也会唤醒设备，该标志常用于闹钟触发，蓝牙链接提醒等场景。					不能和 PARTIAL_WAKE_LOCK 一起用
ON_AFTER_RELEASE = 0x20000000	说明：和用户体验有关，当 wakelock 释放后如果没有该标志，屏幕会立即黑屏，如果有该标志，屏幕会亮一小会然后在黑屏					不能和 PARTIAL_WAKE_LOCK 一起用

申请锁——acquire

应用创建锁之后，必须通过acquire申请锁才能持有wakelock锁，才能保证系统处于唤醒状态来使用电力资源。PowerManager和PowerManagerService中都有wakelock内部类，在PowerManagerService启动之时便将其注册到Binder服务端，其客户端代理调用由PowerManager来完成，故PowerManager中acquire申请锁具体操作实现在服务端PowerManagerService中的acquireWakeLock()。

申请锁流程图如下：





在acquireWakeLock中检查完权限后，调用到acquireWakeLockInternal()中

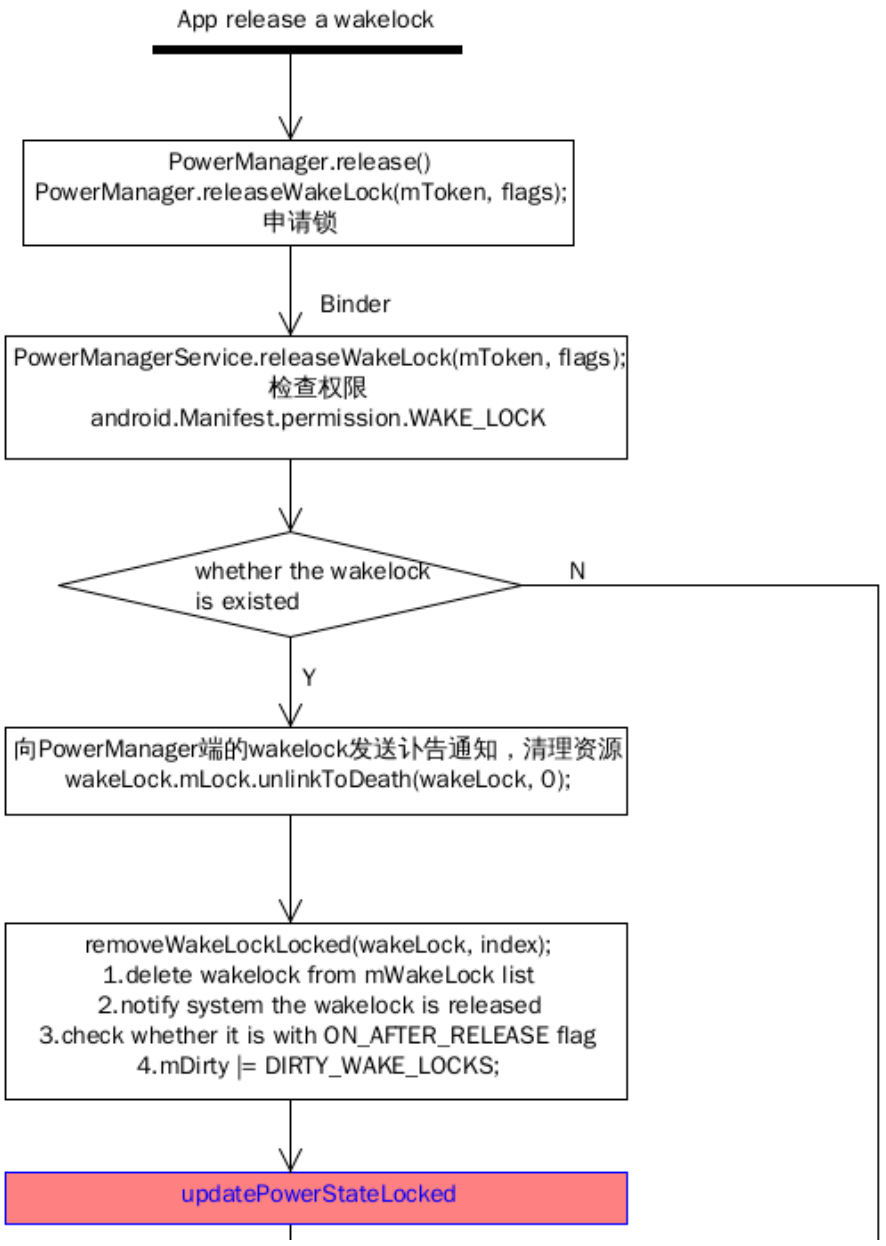
```
int index = findWakeLockIndexLocked(lock);
boolean notifyAcquire;
if (index >= 0) {
wakeLock = mWakeLocks.get(index);
if (!wakeLock.hasSameProperties(flags, tag, ws, uid, pid)) {
notifyWakeLockChangingLocked(wakeLock, flags, tag, packageName, uid, pid, ws, histor
yTag);
wakeLock.updateProperties(flags, tag, packageName, ws, historyTag, uid, pid);
}
notifyAcquire = false;
} else {
wakeLock = new WakeLock(lock, flags, tag, packageName, ws, historyTag, uid, pid);
```

acquireWakeLockInternal()方法首先检查mWakeLock列表中是否已经存在有相同的wakelock，如果存在那么调用wakeLock.updateProperties()更新该wakelock的属性值，如果不存在，那么创建wakelock锁，然后更新电源状态：

```
applyWakeLockFlagsOnAcquireLocked(wakeLock, uid); mDirty |= DIRTY_WAKE_LOCKS;
updatePowerStateLocked();
```

在申请wakelock锁时候applyWakeLockFlagsOnAcquireLocked()检查申请锁是否带有ACQUIRE_CAUSES_WAKEUP标志，若带有该标志，则会直接调用到wakeup调用线程上，执行唤醒操作，该标志在闹钟，蓝牙链接，来电以及短信窗口提醒等功能中有应用；mDirty记录申请锁的操作，最后调用到updatePowerStateLocked中更新电源状态。

释放锁——release
在应用持有wakelock锁执行完相应的事物之后，要及时调用release()，来执行释放wakelock操作，否则会导致设备保持唤醒，迟迟进入不了睡眠状态，严重影响手机功耗。正常情况下，每个wakelock的acquire都应该对应一个release操作，release操作和acquire流程相似。
其流程如下图所示



接口是PowerManager的release()接口，具体实现在PowerManagerService的releaseWakeLock()，releaseWakeLock()检查完权限之后，到releaseWakeLockInterna1()处理

```
int index = findWakeLockIndexLocked(lock);
WakeLock wakeLock = mWakeLocks.get(index);

wakeLock.mLock.unlinkToDeath(wakeLock, 0);
removeWakeLockLocked(wakeLock, index);
}
}
```

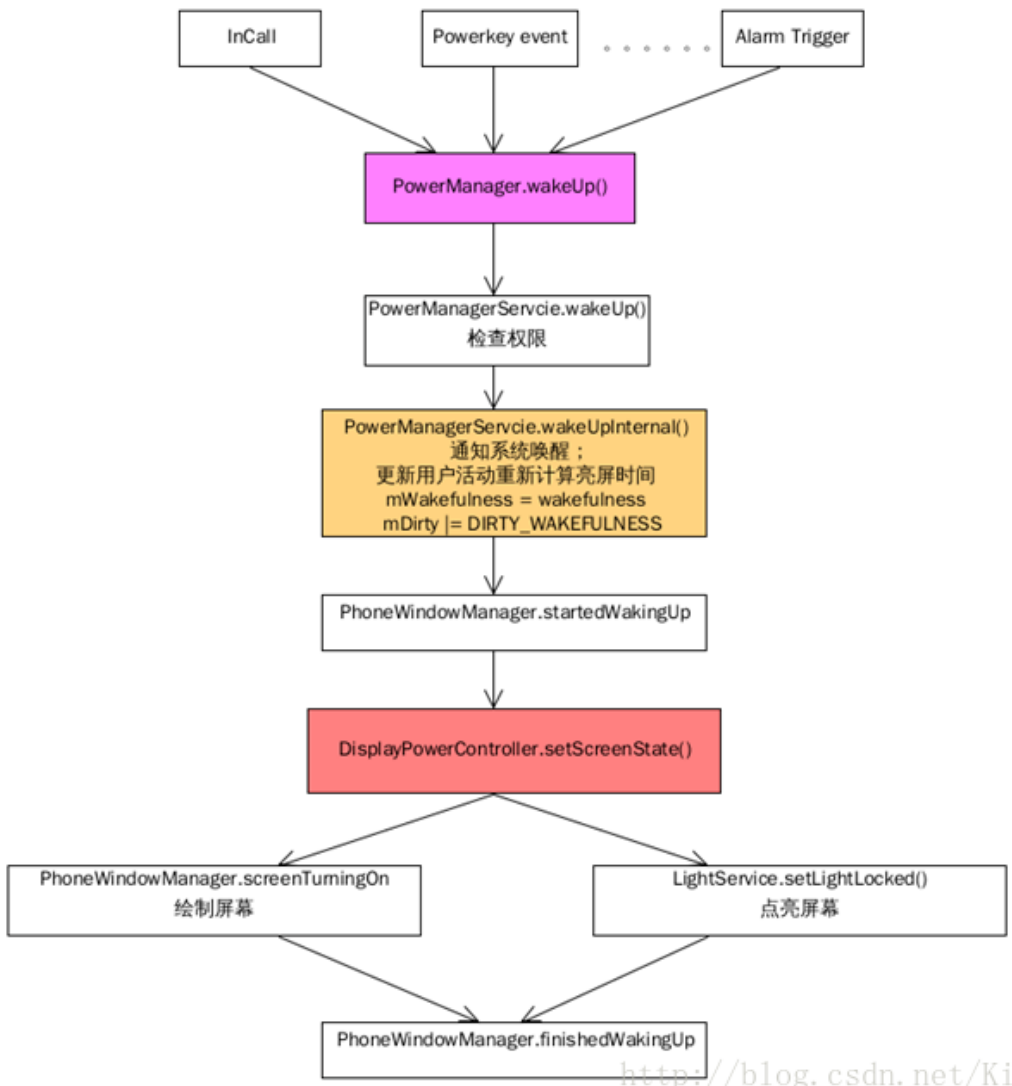
获取要释放的wakelock在mWakeLocks列表中的下标值，然后调用removeWakeLockLocked()删除该wakelock，删除该wakelock的操作如下；

```
private void removeWakeLockLocked(WakeLock wakeLock, int index) {
mWakeLocks.remove(index);
notifyWakeLockReleasedLocked(wakeLock);

applyWakeLockFlagsOnReleaseLocked(wakeLock);
mDirty |= DIRTY_WAKE_LOCKS;
updatePowerStateLocked();
}
```

首先在mWakeLocks列表中将该wakelock删除，然后通知系统该wakelock已经被删除，applyWakeLockFlagsOnReleaseLocked()中判断该wakelock是否有ON_AFTER_RELEASE标志，如果带有这个标志释放wakelock锁后系统不会立即进入黑屏状态，而是屏幕继续亮一小会再灭屏，在蓝牙连接，小区广播，来电等功能场景下有应用，由于wakelock被释放了，所以mDirty置为DIRTY_WAKE_LOCKS。再次调用到updatePowerStateLocked()更新电源状态。

Wakelock在流程反映
先看一张wakelock工作的流程图



SuspendBlocker是用来保持CPU唤醒的一种锁，前面讲到，上层应用可以申请许多不同的wakelock锁，所有的wakelock锁放到mWakelocks的一个列表中，在updateWakeLockSummaryLocked中将所有的wakelock通过置位的方法统计到一个集合变量mWakeLockSummary中，列表中有多个相同类型的wakelock，反映在相应的二进制位上是一样的，所以只需要一个wakelock就可以保证CPU唤醒，
我们回到PowerManagerService的构造函数，
mWakeLockSuspendBlocker=createSuspendBlockerLocked(“PowerManagerService.WakeLocks”);
mDisplaySuspendBlocker=createSuspendBlockerLocked(“PowerManagerService.Displ

ay”);
里面创建了两个变量mWakeLockSuspendBlocker和mDisplaySuspendBlocker,在PowerManagerService启动时调用SystemReady时有:

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

创建两个SuspendBlocker对象mNotifier 和mWirelessChargerDetector ，mNotifier 是在电池屏幕亮度发生迅速提升时（如暗屏触摸），应用点亮屏幕时或者使用无线充电时，申请PowerManagerService.Broadcasts的SuspendBlocker锁。当检测到用无线充电时，会申请PowerManagerService.WirelessChargerDetector标志的SuspendBlocker锁

SuspendBlocker是一个抽象类。其创建的对象是其实现类SuspendBlockerImpl对象，SuspendBlockerImpl类中维护了一个计数器mReferenceCount，调用acquire时候，计数器加1，当计数器值为1时，调用到底层的nativeAcquireSuspendBlocker方法，申请wakelock；调用release时，计数器减1，当计数器为0时候调用底层的nativeReleaseSuspendBlocker方法，释放wakelock；这两个方法如下

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

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

其 acquire_wake_lock(release_wake_lock)直接调用到HAL层的power.c向系统文件节点/sys/power/wake_lock(/sys/power/unwake_lock)中写数据，这里写数据就是前面构造函数和Systemready中创建变量时的参数”PowerManagerService.WakeLocks”，”PowerManagerService.Display”等参数

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行了。

电源管理核心——updatePowerStateLocked

updatePowerSateLocked()方法为PowerManagerService之核心，前面分析了接口调用都是更新成员变量值，最后都是需要调用到updatePowerSateLocked()来更新电源状态。

```
private void updatePowerStateLocked() {
if (!mSystemReady || mDirty == 0) {
return;
}
if (!Thread.holdsLock(mLock)) {
Slog.wtf(TAG, "Power manager lock was not held when calling updatePowerStateLocke
d");
}
Trace.traceBegin(Trace.TRACE_TAG_POWER, "updatePowerState");
try {
updateIsPoweredLocked(mDirty);
updateStayOnLocked(mDirty);
updateScreenBrightnessBoostLocked(mDirty);

final long now = SystemClock.uptimeMillis();
int dirtyPhase2 = 0;
for (;;) {
int dirtyPhase1 = mDirty;
dirtyPhase2 |= dirtyPhase1;
mDirty = 0;
updateWakeLockSummaryLocked(dirtyPhase1);
updateUserActivitySummaryLocked(now, dirtyPhase1);
if (!updateWakefulnessLocked(dirtyPhase1)) {
break;
}
}
boolean displayBecameReady = updateDisplayPowerStateLocked(dirtyPhase2);

updateDreamLocked(dirtyPhase2, displayBecameReady);

finishWakefulnessChangeIfNeededLocked();

updateSuspendBlockerLocked();
} finally {
Trace.traceEnd(Trace.TRACE_TAG_POWER);
}
}
```

updatePowerStateLocked()方法并不长，但是其涉及调用的方法较为复杂，还是不容以理解，下面作详细分析：
电源状态更新，最重要的标志变量为mDirty，当与电源相关的状态改变，都会通过置位的方法反映在mDirty集合变量了，比如充电状态，屏幕亮度，电源设置，唤醒状态等发生改变都会在mDirty中反映出来

1).updateIsPoweredLocked():该方法主要是通过调用BatteryService更新电池状态，包括电池充电，电量等级等状态。

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

mIsPowered表示是否在充电，mP1ugType 表示充电类型，mBatteryLeve1 表示当前电量等级，mBatteryLeve1Low 表示是否为低电水平；

2).updateStayOnLocked()来更新变量mStayOn的值，如果mStayOn如果为true，则屏幕长亮，在Setting中可以设置充电时候屏幕长亮，如果Setting中设置了该选项，updateIsPoweredLocked检查到正在充电，会将mStayOn置为true。

3).接下来就是一个无限循环，但是这个循环最多执行两次便退出了，这一点将在后面详细分析，循环中最先调用updateWakeLockSummaryLocked，来将系统中所有的wakelock锁更新到一个集合变量mWakeLockSummary中，也就是不管系统中创建了多少个wakelock，一个便足以阻止系统进入睡眠状态，因此这里将所有的wakelock总结后通过置位的方法保存到一个变量中，应用创建wakelock时会指定wakelock的类型，不同的wakelock类型置于不同的位。

4)循环调用的第二个方法是updateUserActivitySummaryLocked，在方法中根据系统最后一次调用userActivity()方法的时间计算现在是否可以将屏幕状态的变量mUserActivitySummary置成USER_ACTIVITY_SCREEN_BRIGHT(亮屏)还是USER_ACTIVITY_SCREEN_DIM(暗屏)等，这在上面已经分析过了，这里就不多赘述了。

5).循环中调用的第三个方法是updateWakefu1nessLocked()，这个方法是循环结束的关键，

如果它的返回值为true，则表示wakefulness的状态发生改变了，降继续循环重新调用前面两个方法更新userActivity和Wakelock集合变量。如果能第二次调用updateWakefulnessLocked()一定会返回false，继而跳出循环，方法实现为

```
1  private boolean updateWakefulnessLocked(int dirty) {
2  boolean changed = false;
3  if ((dirty & (DIRTY_WAKE_LOCKS | DIRTY_USER_ACTIVITY | DIRTY_BOOT_COMPLETED
4  | DIRTY_WAKEFULNESS | DIRTY_STAY_ON | DIRTY_PROXIMITY_POSITIVE
5  | DIRTY_DOCK_STATE)) != 0) {
6  if (mWakefulness == WAKEFULNESS_AWAKE && isItBedTimeYetLocked()) {
7  if (DEBUG_SPEW) {
8  }
9  final long time = SystemClock.uptimeMillis();
10 if (shouldNapAtBedTimeLocked()) {
11 changed = napNoUpdateLocked(time, Process.SYSTEM_UID);
12
13 } else {
14 changed = goToSleepNoUpdateLocked(time,
15 PowerManager.GO_TO_SLEEP_REASON_TIMEOUT, 0, Process.SYSTEM_UID);
16 }
17 }
18 }
19 return changed;
20 }
```

第一个if非常用以满足，第二个条件要求mWakefulness为WAKEFULNESS_AWAKE，且isItBedTimeYetLocked()为true，此函数官方解释为当系统马上要进入睡眠状态时会返回true，也就是当系统一直处于活跃状态，则其返回false所以，updateWakefulnessLocked()方法返回值为false，那么这个死循环只用执行一次就跳出了，这里假定系统一段时间未被操作，即将接下来就要调用进入睡眠状态，则isItBedTimeYetLocked()函数返回true，接下来就该调用shouldNapAtBedTimeLocked()方法了，该方法检查又没有设置睡眠之前启动动态屏保或者插在座充上启动屏保，如果设置了，调用napNoUpdateLocked()，没有设置则调用gotoSleepNoUpdateLocked()。
napNoUpdateLocked方法主要代码如下：

```
1  if (eventTime < mLastWakeTime || mWakefulness != WAKEFULNESS_AWAKE
2  || !mBootCompleted || !mSystemReady) {
3  return false;
4  }
5  try {
6  mSandmanSummoned = true;
7  setWakefulnessLocked(WAKEFULNESS_DREAMING, 0);
8  } finally {
9  Trace.traceEnd(Trace.TRACE_TAG_POWER);
10 }
11 return true;
```

如果if语句中有一项成立则返回false，则跳出死循环，当时如果第一次调用该方法，正常情况下当为false，如果第二次调用到此肯定会返回false，因为第二次调用时mWakefulness 为，WAKEFULNESS_DREAMING。而gotoSleepNoUpdateLocked前面已作分析，这里就不多讲了。

6).跳出循环则调用到updateDisplayPowerStateLocked()更新屏幕显示，当前面如果申请了亮屏锁和更新userActivity时，mUserActivitySummary带有USER_ACTIVITY_SCREEN_BRIGHT标志，则会将mDisplayPowerRequest.policy置为POLICY_BRIGHT，这个标志在DisplayPowerController中会将屏幕从灭屏状态下唤醒。
7).接下来调用updateDreamLocked(),更新屏保模式，具体处理在handleSandman()函数中，该函数是当系统进入/退出屏保状态或者Dozing下状态调用，下面详细分析该函数代码：

```
1  synchronized (mLock) {
2  mSandmanScheduled = false;
3  wakefulness = mWakefulness;
4  if (mSandmanSummoned && mDisplayReady) {
5  startDreaming = canDreamLocked() || canDozeLocked();
6  mSandmanSummoned = false;
7  } else {
8  startDreaming = false;
9  }
10 }
```

这里主要更新变量startDreaming，当前状态如果可以进入屏保或者Dozing状态，则置为true，否则置为false。，如果设置了屏保，则灭屏之后会进入一段时间屏保，然后灭屏；如果没有设置屏保则默认进入dozing（打盹）状态。

```
final boolean isDreaming;
if (mDreamManager != null) {
mDreamManager.stopDream(false /*immediate*/);
mDreamManager.startDream(wakefulness == WAKEFULNESS_DOZING);
}
isDreaming = mDreamManager.isDreaming();
} else {
isDreaming = false;
}
```

此处判断屏保服务是否启动了，而在屏保服务实在系统启动时在StartOtherService中随系统启动，在次重启屏保服务

```
if (isItBedTimeYetLocked()) {
goToSleepNoUpdateLocked(SystemClock.uptimeMillis(),
PowerManager.GO_TO_SLEEP_REASON_TIMEOUT, 0, Process.SYSTEM_UID);
updatePowerStateLocked();
} else {
wakeUpNoUpdateLocked(SystemClock.uptimeMillis(), "android.server.power:DREAM",
Process.SYSTEM_UID, mContext.getOpPackageName(), Process.SYSTEM_UID);
updatePowerStateLocked();
} else if (wakefulness == WAKEFULNESS_DOZING) {
if (isDreaming) {
return; // continue dozing
}
```

如果屏保结束，判断是进入唤醒状态还是进入睡眠状态，然后更新电源状态。若是还为到时，且wakefulness为WAKEFULNESS_DOZING则返回，继续处于dozing状态。

```
reallyGoToSleepNoUpdateLocked(SystemClock.uptimeMillis(), Process.SYSTEM_UID)
;
updatePowerStateLocked();
```

最后时间及到，真正进入睡眠状态调用reallyGoToSleepNoUpdateLocked，将mWakefulness置为WAKEFULNESS_ASLEEP，再次更新电源状态时将wakelock锁尽数释放，屏幕完全灭屏，进入睡眠状态。

8).finishWakefulnessChangeIfNeededLocked()方法未作实际操作，仅仅通知系统mWakefulness改变更新已经完成。

9)最后一个函数updateSuspendBlockerLocked();由于系统中可能需要释放最后一个维持CPU唤醒或者维持屏幕亮灭的Blocker，所以必须将所有事物处理完成，再执行该操作。由于该函数是由PowerManagerService调用到底层的唯一入口，所以十分重要：

```
private void updateSuspendBlockerLocked() {
final boolean needWakeLockSuspendBlocker = ((mWakeLockSummary & WAKE_LOCK_C
PU) != 0);
final boolean needDisplaySuspendBlocker = needDisplaySuspendBlockerLocked();
final boolean autoSuspend = !needDisplaySuspendBlocker;
final boolean interactive = mDisplayPowerRequest.isBrightOrDim();
```

needWakeLockSuspendBlocker变量判断wakelock是否带有WAKE_LOCK_CPU标志来决定是否为需要保持CPU唤醒，其中能维持CPU唤醒的wakelock类型有：PARTIAL_WAKE_LOCK，FULL_WAKE_LOCK，SCREEN_BRIGHT_WAKE_LOCK，SCREEN_DIM_WAKE_LOCK，DRAW_WAKE_LOCK。needDisplaySuspendBlocker 则是表示是否维持屏幕亮灭的变量，true表示维持屏幕亮，false表示可以关闭屏幕。

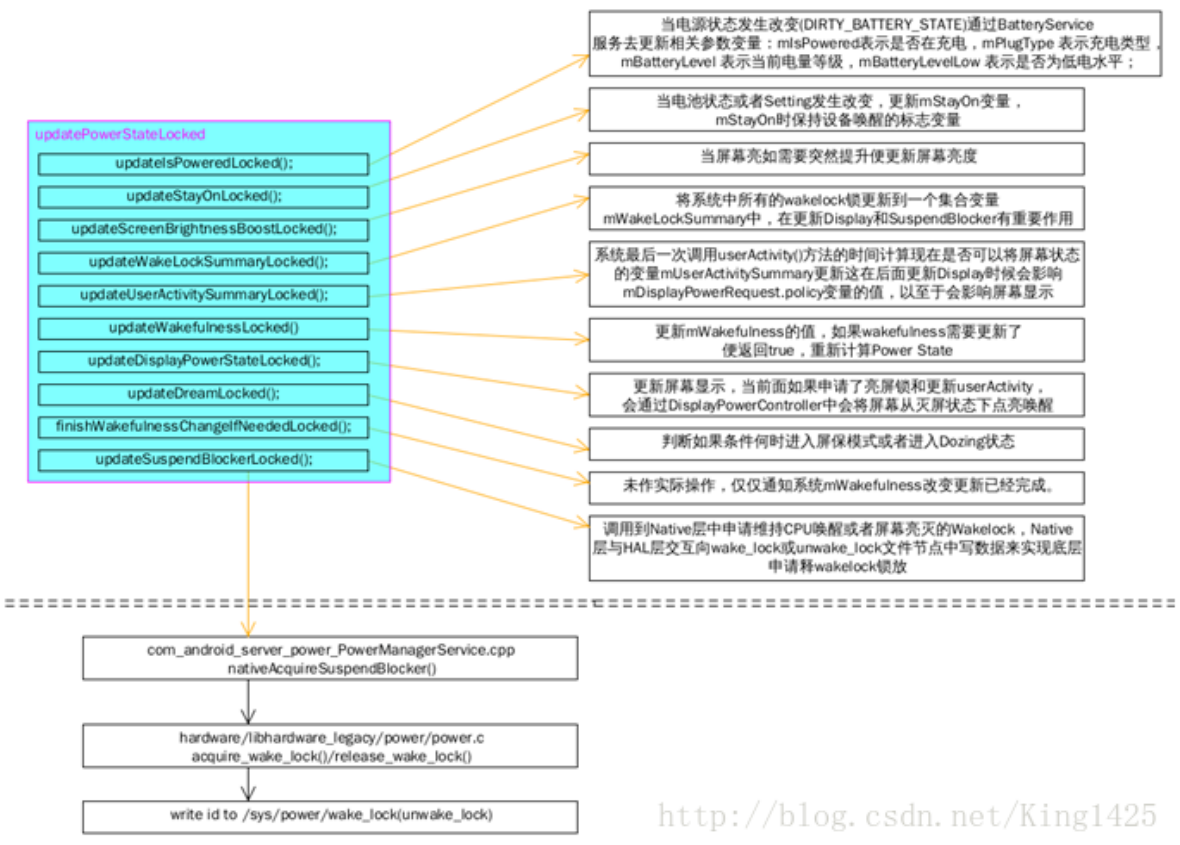
```
if (needWakeLockSuspendBlocker && !mHoldingWakeLockSuspendBlocker) {
mWakeLockSuspendBlocker.acquire();
mHoldingWakeLockSuspendBlocker = true;
}
if (needDisplaySuspendBlocker && !mHoldingDisplaySuspendBlocker) {
mDisplaySuspendBlocker.acquire();
mHoldingDisplaySuspendBlocker = true;
}
.....
if (!needWakeLockSuspendBlocker && mHoldingWakeLockSuspendBlocker) {
mWakeLockSuspendBlocker.release();
mHoldingWakeLockSuspendBlocker = false;
}
if (!needDisplaySuspendBlocker && mHoldingDisplaySuspendBlocker) {
mDisplaySuspendBlocker.release();
mHoldingDisplaySuspendBlocker = false;
}
```

上面两段代码是向下申请PowerManagerService.WakeLocks和PowerManagerService.Display类型的wakelock锁的入口，mHoldingWakeLockSuspendBlocker表示当前是否持有cpu唤醒锁，如果持有则不必向下继续申请锁，如果维持有，且需要维持CPU唤醒则需要，申请CPU唤醒锁。（Display逻辑与其相似，就不做分析）调用到 SuspendBlockerImpl的acquire()函数中

```
public void acquire() {
synchronized (this) {
mReferenceCount += 1;
if (mReferenceCount == 1) {
if (DEBUG_SPEW) {
Slog.d(TAG, "Acquiring suspend blocker \"" + mName + "\".");
}
Trace.asyncTraceBegin(Trace.TRACE_TAG_POWER, mTraceName, 0);
nativeAcquireSuspendBlocker(mName);
}
}
}
```

前面已经说过SuspendBlockerImpl维持了一个计数标志mReferenceCount 当为1的时候申请锁，为0时释放锁。nativeAcquireSuspendBlocker()函数通过JNI调用到native层的com_android_server_power_PowerManagerService.cpp文件的nativeAcquireSuspendBlocker函数，而后的详细流程在前面wakelock已经讲的很清楚了 。

至此电源管理核心函数updatePowerStateLocked()基本分析完成。
下图为概述电源管理核心函数处理流程及作用



总结

Android的电源管理提出wakelock的是一套全新的机制，跟我们C++里使用的智能指针（Smart pointer），借用智能指针的思想来设计电源的使用和分配。Smart Pointer都是引用，申请引用则它的引用计数会自动加1，取消引用则引用计数减1，使用了智能指针的对象，当它的引用

计数为0时，则该对象会被回收掉。同样， 我们的wake_lock也保持使用计数，只不过这种“智能指针”的所使用的资源不再是内存，而是电量。应用程序会通过特定的WakeLock去访问硬件，然后硬件会根据引用计数是否为0来决定是不是需要关闭这一硬件的供电。
电源管理于Framework层恰似为一个策略控制器，来掌控不同状态下的电源状态改变和更新，PowerManager作为一个重要服务在开机启动时便启动并注册到系统内，当上层应用程序需要使用该服务只需调用PowerManager开放接口即可控制系统某些电源状态的改变，而PowerManagerServcie提供服务端处理逻辑，在交互中做主要电源控制工作，其中与之交互的模块最频繁的为Display，Window，Light，和Battery等，其交互之复杂，联系之紧密实非一言以蔽之。