Android 电源管理:

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。

framework 层主要有这两个文件:

frameworks\base\core\java\android\os\PowerManager.java

frameworks\base\services\java\com\android\server**PowerManagerService.java**

其中 PowerManager.java 是提供给应用层调用的，最终的核心还是在 PowerManagerService.java。这个类的作用就是提供 PowerManager 的功能，以及整个电源管理状态机的运行。里面函数和类比较多，就从对外和对内分两块来说。

先说对外，PowerManagerService 如何来进行电源管理，那就要有外部事件的时候去通知它，这个主要是在 frameworks\base\services\java\com\android\server**WindowManagerService.java** 里面。WindowManagerService 会把用户的点击屏幕，按键等作为 user activity 事件来调用 userActivity 函数，PowerManagerService 就会在 userActivity 里面判断事件类型作出反映，是点亮屏幕提供操作，还是完全不理会，或者只亮一下就关掉。

供 WindowManagerService 调用的方法还有 **gotoSleep** 和其他一些获取电源状态的函数比如 **screenIsOn** 等等。

再说对内，作为对外接口的 userActivity 方法主要是通过 **setPowerState** 来完成功能。把要设置的电源状态比如开关屏幕背光什么的作为参数调用 setPowerState，setPowerState 先判断下所要的状态能不能完成，比如要点亮屏幕的话但是现在屏幕被 lock 了那就不能亮了，否则就可以调用

Power.setScreenState(true)来透过 jni 跑到 driver 里面去点亮屏幕了。

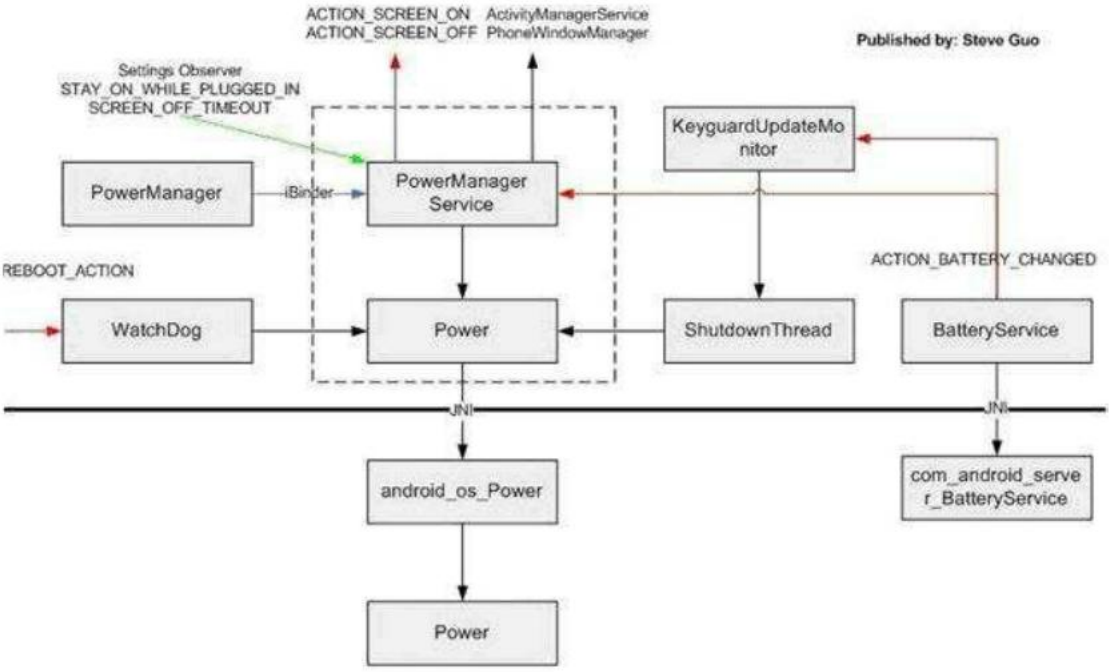
而电源的状态循环则主要是通过 Handler 来实现的。PowerManagerService 在 init 里面会启动一个 **HandlerThread** 一个后台消息循环来提供任务的延迟发送，就可以使用 Handler 来在定制推迟某一任务的执行时间，从而实现状态机的循环。比如 timeout，一段时间之后无操作要让屏幕变暗，然后关闭，反映在代码里如下:

userActivity 里面在调用 setPowerState 之后会用 **setTimeoutLocked** 来设置 timeout。然后在 setTimeoutLocked 里面会根据当前的状态来计算下一个状态以及时间，判断完再调用 mHandler.postAtTime(mTimeoutTask, when)来 post 一个

TimeoutTask。这样在 when 毫秒后就会执行 TimeoutTask。在 TimeoutTask 里面则根据设定的状态来调用 setPowerState 来改变电源状态，然后再设定新的状态，比如现在是把屏幕从亮改暗了，那就再用 setTimeoutLocked(now, SCREEN_OFF)来等下把屏幕完全关掉。如果这次已经是把屏幕关了，那这轮的 timeout 状态循环就算是结束了。

如果要定制的话，比如需求是在 timeout 屏幕关掉之后还要再关掉一些外围设备等等，那就在 TimeoutTask 里面把屏幕关掉之后再加上关闭其他设备的代码就好了。即使新的状态需求完全和原来的不一样，用 Handler 应该也不难。逻辑理清了把代码摆在合适的地方就好了。

总体上来说 Android 的电源管理还是比较简单的，主要就是通过**锁和定时器**来切换系统的状态,使系统的功耗降至最低,整个系统的电源管理架构图如下:



接下来我们从Java应用层面, Android framework 层面, Linux 内核层面分别进行详细的讨论:

应用层的使用:

Android 提供了现成 android.os.**PowerManager** 类,该类用于控制设备的电源状态的切换。

该类对外有三个接口函数:

1、 void goToSleep(long time); //强制设备进入 Sleep 状态

Note:

尝试在应用层调用该函数,却不能成功,出现的错误好象是权限不够, 但在 Framework 下面的 Service 里调用是可以的.

2、newWakeLock(int flags, String tag);　　//取得相应层次的锁

flags 参数说明:

PARTIAL_WAKE_LOCK　　// Screen off, keyboard light off

SCREEN_DIM_WAKE_LOCK　　// screen dim, keyboard light off

SCREEN_BRIGHT_WAKE_LOCK　　//screen bright, keyboard light off

FULL_WAKE_LOCK　　// screen bright, keyboard bright

ACQUIRE_CAUSES_WAKEUP　　//一旦有请求锁时强制打开 Screen 和 keyboard light

ON_AFTER_RELEASE　　// 在释放锁时 reset activity timer

Note:

如果申请了 partial wakelock,那么即使按 Power 键,系统也不会进 Sleep,如 Music 播放时

如果申请了其它的 wakelocks,按 Power 键,系统还是会进 Sleep

3、void userActivity(long when, boolean noChangeLights);
//User activity 事件发生,设备会被切换到 Full on 的状态,同时 Reset Screen off timer.

Sample code:

```
PowerManager pm = (PowerManager)getSystemService(Context.POWER_SERVICE);

PowerManager.WakeLock wl = pm.newWakeLock (PowerManager.SCREEN_DIM_WAKE_LOCK, “My Tag”);

wl.acquire();

…….

wl.release();
```

Note:

1. 在使用以上函数的应用程序中,必须在其 Manifest.xml 文件中加入下面的权限:

<uses-permission android:name="android.permission.WAKE_LOCK" />
<uses-permission android:name="android.permission.DEVICE_POWER" />

2. 所有的锁必须成对的使用,如果申请了而没有及时释放会造成系统故障.如申请了 partial wakelock,而没有及时释放,那系统就永远进不了 Sleep 模式.

Android Framework 层面:
其主要代码文件如下:
frameworks\base\core\java\android\os\PowerManager.java
frameworks\base\services\java\com\android\server\PowerManagerService.java
frameworks\base\core\java\android\os\Power.java
frameworks\base\core\jni\android_os_power.cpp
hardware\libhardware\power\power.c

其中 **PowerManagerService.java** 是核心, Power.java 提供底层的函数接口,与 JNI 层进行交互, JNI 层的代码主要在文件 android_os_Power.cpp 中,与 Linux kernel 交互是通过 Power.c 来实现的, Andriod 跟 Kernel 的交互主要是通过 **sys 文件**的方式来实现的,具体请参考 Kernel 层的介绍.

这一层的功能相对比较复杂,比如系统状态的切换,背光的调节及开关,Wake Lock 的申请和释放等等,但这一层跟硬件平台无关,而且由 Google 负责维护,问题相对会少一些,有兴趣的朋友可以自己查看相关的代码.

Kernel 层:
其主要代码在下列位置:
drivers/android/power.c
其对 Kernel 提供的接口函数有:

EXPORT_SYMBOL(android_init_suspend_lock);
//初始化 Suspend lock,在使用前必须做初始化

EXPORT_SYMBOL(android_uninit_suspend_lock);
//释放 suspend lock 相关的资源

EXPORT_SYMBOL(android_lock_suspend);
//申请 lock,必须调用相应的 unlock 来释放它

EXPORT_SYMBOL(android_lock_suspend_auto_expire);
//申请 partial wakelock, 定时时间到后会自动释放

EXPORT_SYMBOL(android_unlock_suspend); //释放 lock

EXPORT_SYMBOL(android_power_wakeup); //唤醒系统到 on

EXPORT_SYMBOL(android_register_early_suspend); //注册 early suspend 的驱动

EXPORT_SYMBOL(android_unregister_early_suspend);
//取消已经注册的 early suspend 的驱动

提供给 Android Framework 层的 proc 文件如下:

"/sys/android_power/acquire_partial_wake_lock"　　//申请 partial wake lock

"/sys/android_power/acquire_full_wake_lock"　　//申请 full wake lock

"/sys/android_power/release_wake_lock"　　//释放相应的 wake lock

"/sys/android_power/request_state"
//请求改变系统状态,进 standby 和回到 wakeup 两种状态

"/sys/android_power/state" //指示当前系统的状态

Android 的电源管理主要是通过 **Wake lock** 来实现的,在最底层主要是通过如下三个队列来实现其管理:
static LIST_HEAD(g_inactive_locks);
static LIST_HEAD(g_active_partial_wake_locks);
static LIST_HEAD(g_active_full_wake_locks);

所有初始化后的 lock 都会被插入到 g_inactive_locks 的队列中,而当前活动的 partial wake lock 都会被插入到 g_active_partial_wake_locks 队列中, 活动的 full wake lock 被插入到 g_active_full_wake_locks 队列中, 所有的 partial wake lock 和 full wake lock 在过期后或 unlock 后都会被移到 inactive 的队列,等待下次的调用.

在 Kernel 层使用 wake lock 步骤如下:

1. 调用函数 android_init_suspend_lock 初始化一个 wake lock
2. 调用相关申请 lock 的函数 android_lock_suspend 或 android_lock_suspend_auto_expire 请求 lock, 这里只能申请 partial wake lock, 如果要申请 Full wake lock, 则需要调用函数 android_lock_partial_suspend_auto_expire(该函数没有 EXPORT 出来),这个命名有点奇怪,不要跟前面的 android_lock_suspend_auto_expire 搞混了.

3. 如果是 auto expire 的 wake lock 则可以忽略,不然则必须及时的把相关的 wake lock 释放掉,否则会造成系统长期运行在高功耗的状态.

4. 在驱动卸载或不再使用 Wake lock 时请记住及时的调用 android_uninit_suspend_lock 释放资源.

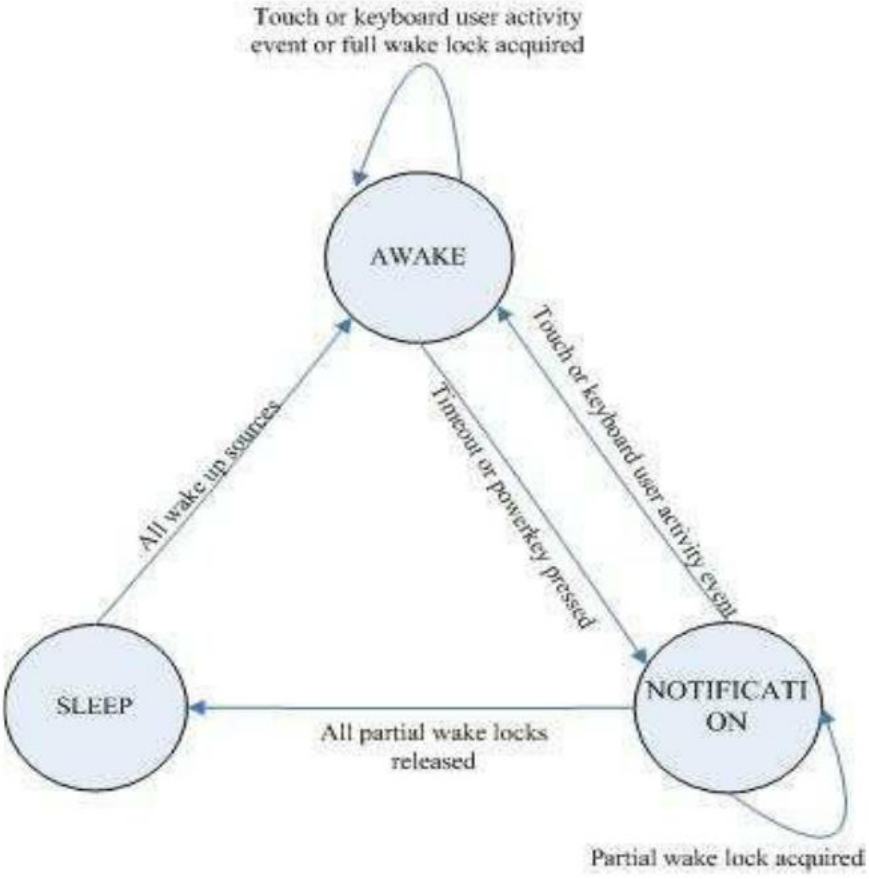
系统的状态:

USER_AWAKE, //Full on status

USER_NOTIFICATION, //Early suspended driver but CPU keep on

USER_SLEEP // CPU enter sleep mode

其状态切换示意图如下:



系统正常开机后进入到 AWAKE 状态, Backlight 会从最亮慢慢调节到用户设定的亮度,系统 screen off timer (settings->sound & display-> Display settings -> Screen timeout)开始计时,在计时时间到之前,如果有任何的 activity 事件发生,如 Touch click, keyboard pressed 等事件, 则将 Reset screen off timer, 系统保持在 AWAKE 状态. 如果有应用程序在这段时间内申请了 Full wake lock,那么系统也将保持在 AWAKE 状态, 除非用户按下 power key. 在 AWAKE 状态下如果电池电量低或者是用 AC 供电 screen off timer 时间到并且选中 Keep screen on while pluged in 选项,backlight 会被强制调节到 DIM 的状态.

如果 Screen off timer 时间到并且没有 Full wake lock 或者用户按了 power key, 那么系统状态将被切换到 NOTIFICATION, 并且调用所有已经注册的 g_early_suspend_handlers 函数, 通常会把 LCD 和 Backlight 驱动注册成 early

suspend 类型,如有需要也可以把别的驱动注册成 early suspend, 这样就会在第一阶段被关闭. 接下来系统会判断是否有 partial wake lock acquired, 如果有则等待其释放, 在等待的过程中如果有 user activity 事件发生,系统则马上回到 AWAKE 状态;如果没有 partial wake lock acquired, 则系统会马上调用函数 pm_suspend 关闭其它相关的驱动, 让 CPU 进入休眠状态.

系统在Sleep状态时如果检测到任何一个Wakeup source, 则CPU会从Sleep状态被唤醒,并且调用相关的驱动的 resume 函数,接下来马上调用前期注册的 early suspend 驱动的 resume 函数,最后系统状态回到 AWAKE 状态.这里有个问题就是所有注册过 early suspend 的函数在进 Suspend 的第一阶段被调用可以理解, 但是在 resume 的时候, Linux 会先调用所有驱动的 resume 函数,而此时再调用前期注册的 early suspend 驱动的 resume 函数有什么意义呢?个人觉得 android 的这个 early suspend 和 late resume 函数应该结合 Linux 下面的 suspend 和 resume 一起使用,而不是单独的使用一个队列来进行管理。