

Balancor

博客园

首页

新随笔

联系

订阅

管理

随笔 - 14 文章 - 0 评论 - 8

公告

昵称：Balancor
园龄：4年11个月
粉丝：14
关注：4
+加关注

<	2017年4月						>
日	一	二	三	四	五	六	
26	27	28	29	30	31	1	
2	3	4	5	6	7	8	
9	10	11	12	13	14	15	
16	17	18	19	20	21	22	
23	24	25	26	27	28	29	
30	1	2	3	4	5	6	

搜索

找找看


谷歌搜索

常用链接


我的随笔
我的评论
我的参与

Android Framework-----之PowerManagerService的功能

自从接触Android系统已经一年多了，这段时间内对于Android系统的Framework层的各个模块都有过接触，有时也做过分析，但是一直没能形成一个总结性的东西。这次下定决心，好好整理整理对于Android系统的学习梳理一下自己的思路。本文一方面是为了自己梳理下知识，文中涉及的内容，基本是拾人牙慧，很少有自己的东西，最多也就算是自己的总结；除此作用之外，如果能为后来者引玉，也算是一点功德吧。这次首先是对Android系统中的PowerManagerService进行下整理。之所以先选择PowerManagerService，是因为这个模块相对于Android系统中其他的模块而言，与系统其他的模块之间的交互较少，而且Framework中的PowerManagerService模块是由Google开发并维护的，虽然以Linux Kernel的Power为基础，但是它们之间的耦合度低，完全可以把两者分开，单独进行分析也不会造成困惑。接下来，我会从不同的角度，分别介绍下PowerManagerService的功能。还是先来看看PowerManagerService在Framework中的目录结构吧。PowerManagerService在Android4.2源码中的位置是：/frameworks/base/services/java/com/android/server/power/，在这个目录下有以下文件：



DisplayBlanker.java
DisplayPowerController.java
DisplayPowerRequest.java
DisplayPowerState.java
ElectronBeam.java
Notifier.java
PowerManagerService.java
RampAnimator.java
ScreenOnBlocker.java
ShutdownThread.java
SuspendBlocker.java
WirelessChargerDetector.java



最新评论

我的标签

随笔分类

Android(8)

随笔档案

2016年5月 (1)

2014年12月 (2)

2013年9月 (2)

2013年8月 (2)

2013年3月 (1)

2012年11月 (1)

2012年10月 (3)

2012年8月 (2)

Blog Background

最新评论

1. Re:Android Init语言

1) init.rc中import init.qcom.rc, 但是init.rc和init.qcom.rc中都有on post-fs-data Action, 这个跟你说的不能出现同名的action, 后.....

--者旨於陽

2. Re:Android Init语言

你好, 我这边遇到了一些问题, 想请教一下!

--者旨於陽

3. Re:Android Framework-----之

Input子系统

写的很好, 最后的两幅流程图很直观, 请问是用什么软件生成的, 还是手工画的?

--TaigaComplex

这些文件中, 个人认为对于PowerManagerService而言除了本身的代码, 较为重要的有DisplayPowerController.java, DisplayPowerState.java, Notifier.java.

而DisplayPowerRequest相当于一个辅助类, 用来存储一些统一的属性和变量, 让PowerManagerService和DisplayPowerController, DisplayPowerState交互时

能够使用统一的变量。另外, 还有几个接口文件SuspendBlocker.java, DisplayBlanker.java, ScreenOnBlanker.java。其次就是ShutdownThread.java,

WirelessChargerDetector.java, RampAnimator.java, ElectronBeam.java。下面, 就先逐一介绍下PowerManagerService在Framework中的这些文件:

PowerManagerService.java: 主要是计算系统中和Power相关的计算, 然后决策系统应该如何反应。同时协调Power如何与系统其它模块的交互, 比如没有用户

活动时, 屏幕变暗等等。

DisplayPowerController.java: 管理Display设备的电源状态。仅在PowerManagerService中实例化了一个对象, 它算是PowerManagerService的一部分, 只不过是独立出

来了而已。主要处理和距离传感器, 灯光传感器, 以及包括关屏在内的一些动画, 通过异步回调的方式来通知PowerManagerService某些事情发生了变化。

DisplayPowerState.java: 在本质上类似于View, 只不过用来描述一个display的属性, 当这些属性发生变化时, 可以通过一个序列化的命令, 让这些和display电源状态的属性

一起产生变化。这个类的对象只能被DisplayPowerController的Looper持有。而这个Looper应该就是PowerManagerService中新建的一个HandlerThread中的Looper。

和PowerManager相关的, 包括DisplayPowerState和DisplayPowerController相关的消息处理应该都可以通过这个HandlerThread进行运转的。

Notifier.java: 将Power Manager state的重要变化通过broadcast发送出去。

接下来就说说三个接口文件

SuspendBlocker.java: 相当于一个partial wake lock。在内部使用, 避免了使用上层的唤醒锁机制

DisplayBlanker.java: 主要功能一是BLANK DISPLAY: The display is blanked, but display memory is maintained and new data can be entered; 而是UNBLANK DISPLAY:

The display is restored/turned to active state. (不知道这两个英文的解释是否恰当, 还有待验证)。

4. Re:Android Framework-----之
Keyguard 简单分析
博主，您分析的很透彻，android4.4
的Keyguard确实改动挺大的，我最
近想自己整个点击更换主题Theme
时，拥有自己修改的解锁界面，我
写了一个简单的解锁界面叫做
keyguard_silder.....

--江山文笑

5. Re:Android Framework-----之
PowerManagerService的功能
言语罗嗦，逻辑混乱，前面说要把
一些东西放在后面讲，到最后也没
讲。

--云且留猪

阅读排行榜

- 1. Android Framework-----之
PowerManagerService的功能
(9574)
- 2. Android Framework-----之
Keyguard 简单分析(7898)
- 3. Android Framework-----之Input
子系统(3084)
- 4. Android4.0中MediaPlayer 和
MediaPlayerService(2412)
- 5. Android Framework-----之
ActivityManagerService与Activity之
间的通信(746)

评论排行榜

- 1. Android Framework-----之
PowerManagerService的功能(3)
- 2. Android Framework-----之
Keyguard 简单分析(2)
- 3. Android Init语言(2)

ScreenOnBlanker.java：描述了一种较为低级的blocker机制，主要用于关屏或者隐藏屏幕内容，直到window manager准备好新的内容

DisplayPowerRequest.java：描述了一些对于display电源状态的请求。

最后看看剩余的这些类：

ShutDownThread.java: 主要功能就是关机和重启，当我想要执行重启或者关机的时候，这个新城就会被启动。

ElectronBeam.java： 负责屏幕由开到关，或者由关到开的一些GL动画。在DisplayPowerController管理

WirelessChargerDetector.java：和无线充电相关的东西，没有细看。

每个文件的大致功能就是这样的，也许有些地方不是很恰当，还是需要仔细阅读源码，才能确切地知道到底是怎么回事，有些功能到底是如何实现的。

先从交互的角度去看看PowerManagerService的功能。在这里的交互是说PowerManagerService与应用程序或者Framework中其他的模块的交互，而不是指和用户之间的直接交互。和用户之间牵涉到交互的内容，在文章的最后也稍微有点介绍。下面就分成两个小节，对于PowerManagerService的交互作以总结。首先：

a). 与应用程序之间的交互

在Android中应用程序并不是直接同PowerManagerService交互的，而是通过PowerManager间接地与PowerManagerService打交道。不过在使用PowerManager和

PowerManager,WakeLock之前，我们要首先在APP中申请使用如下权限：

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

而APP能够与PowerManager做哪些交互，在Android提供的开发文档中给了我们答案。我们可以看到PowerManage提供了如下公共的接口：

PowerManager	PowerManagerService
goToSleep(long time)	goToSleep(long eventTime, int reason)
isScreenOn()	isScreenOn()
reboot(String reason)	reboot(boolean confirm, String reason, boolean wait)


4. Android Framework-----之Input
子系统(1)

推荐排行榜

- 1. Android Framework-----之Input
子系统(2)
- 2. Android Framework-----之
Keyguard 简单分析(1)
- 3. Android Framework-----之
PowerManagerService的功能(1)
- 4. Android Init语言(1)

userActivity(long when, boolean noChangeLights)	userActivity(long eventTime, int event, int flags)
wakeUp(long time)	wakeUp(long eventTime)

在这个表格中，仅仅列出了PowerManager的公开方法中的其中五个，同时列出在PowerManagerService中对应的方法。这里列出的，是与PowerManagerService关系比较紧密的方法，其余的和PowerManager相关的东西会在接下来，慢慢地都谈到的。如果阅读PowerManager的源码的话，你会很容易发现，其实PowerManager的方法在实现的过程中，都是通过调用PowerManagerService相应的函数来实现的。PowerManager就像是PowerManagerService的“代理类”。这里略过PowerManager是如何通过binder与PowerManagerService进行通信的。下面，我们逐一对PowerManagerService中这几个函数的实现进行下简单的分析，在对这些函数分析之前，我觉得还是先对代码中使用的一些变量作以简要的说明为好。其实，在PowerManagerService中绝大部分变量通过名字就能大概知道其意义，不过还有几个较为重要的还是仔细说说为好，首先是重要的变量mDirty,根据代码的注释是说，用来表示power state的变化，而这样的变化在系统中一共定义了12个，每一个state对应一个固定的数字，都是2的倍数。这样的话，当有若干个状态一起变化时，他们按位取或，这样得到的结果既是唯一的，又能准去标示出各个状态的变化。此外还有一个mWakefulness的变量，它用来标示的是device处于的一种状态，是醒着的还是睡眠中，或者处于两者之间的一种状态。这个状态是和 display的电源状态是不同的，display的电源状态是独立管理的。这个变量用来标示DIRTY_WAKEFULNESS这个power state下的一个具体的内容。比如说，系统从进入Draaming的时候，首先变化的是mDirty，在mDirty中对DIRTY_WAKEFULNESS位置位，这说明系统中的DIRTY_WAKEFULNESS发生了变化；此时，仅仅是知道DIRTY_WAKEFULNESS发生了变化，但是不知道wakefulness到底发生了怎样的变化，如果需要进一步知道系统的wakefulness变成了什么，就需要查看下mWakefulness的内容就知道了。相当于是对DIRTY_WAKEFULNESS的一个补充说明吧。 像这样的算是补充性质的变量还有mWakeLockSummary和mUserActivitySummary。好了，接下来我们可以从goToSleep(long eventTime, int reason)开始了，代码如下：



```
1  @Override // Binder call
2      public void goToSleep(long eventTime, int reason) {
3          if (eventTime > SystemClock.uptimeMillis()) {
4              throw new IllegalArgumentException("event time must not be in the future");
5          }
6          //权限检查
7          mContext.enforceCallingOrSelfPermission(android.Manifest.permission.DEVICE_POWER, null);
8
9          final long ident = Binder.clearCallingIdentity();
10         try {
11             goToSleepInternal(eventTime, reason); //这里会调用函数的实现，在PowerManagerService中有很多类似的使用方式，之后的代码中我会
直接列出对应方法的实现
12         } finally {
13             Binder.restoreCallingIdentity(ident);
14         }
15     }
16
17     private void goToSleepInternal(long eventTime, int reason) {
18         synchronized (mLock) {
```

```
19         if (goToSleepNoUpdateLocked(eventTime, reason)) {
20             updatePowerStateLocked();
21         }
22     }
23 }
```

对于文中的代码，我会在不影响阅读的情况下，尽量地少。在这段代码中，涉及到另个重要的函数goToSleepNoUpdateLocked()和updatePowerStateLocked()，而goToSleepNoUpdateLocked是goToSleep功能的计算者，来决定是否要休眠，而updatePowerStateLocked函数算是功能的执行者，而且这个执行者同时负责执行了很多其他的功能，在总结的时候会着重分析这个函数。这里先看 goToSleepNoUpdateLocked方法的代码：

⊕ goToSleepNoUpdateLocked

通过这段代码发现，其实这里并没有真正地让device进行sleep，仅仅只是把PowerManagerService中一些必要的属性进行了赋值，等会在分析updatePowerStateLocked的时候，再给出解释。在PowerManagerService的代码中，有很多的方法的名字中都含有xxxNoUpdateLocked这样的后缀，我觉得这样做大概是因为，都类似于goToSleepNoUpdateLocked方法，并没有真正地执行方法名字所描述的功能，仅仅是更新了一些必要的属性。所以在Android系统中可以把多个power state属性的多个变化放在一起共同执行的，而真正的功能执行者就是updatePowerStateLocked。

b).与系统其它模块之间的交互

PowerManagerService作为Android系统Framework中重要的能源管理模块，除了与应用程序交互之外，还要与系统中其它模块配合，在提供良好的能源管理同时提供友好的用户体验。Android系统除了提供公共接口与其它模块交互外，还提供BroadCast机制，用来对系统中发生的重要变化做出反应。下表列出了，在PowerManagerService中注册的Receiver，以及这些Receiver监听的事件，和处理方法：

BatteryReceiver	ACTION_BATTERY_CHANGED	handleBatterStateChangeLocked()
BootCompleteReceiver	ACTION_BOOT_COMPLETED	startWatchingForBootAnimationFinished()
userSwitchReceiver	ACTION_USER_SWITCHED	handleSettingsChangedLocked

DockReceiver	ACTION_DOCK_EVENT	updatePowerStateLocked
DreamReceiver	ACTION_DREAMING_STARTED ACTION_DREAMING_STOPPED	scheduleSandmanLocked

PowerManagerService中除了注册了这五个Receiver之外，还定义了一个SettingsObserver，用于监视系统中以下属性的变化：



SCREENSAVER_ENABLE，屏保的功能开启

SCREENSAVER_ACTIVE_ON_SLEEP，在睡眠时屏保启动

SCREENSAVER_ACTIVE_ON_DOCK，连接底座并且屏保启动

SCREEN_OFF_TIMEOUT，休眠时间

STAY_ON_PLUGGED_IN，有插入并且屏幕开启

SCREEN_BRIGHTNESS，屏幕的亮度

SCREEN_BRIGHTNESS_MODE，屏幕亮度的模式



当以上这些属性发生变化时，SettingObserver都会监视到，并且调用SettingObserver的onChange方法，

```
1 public void onChange(boolean selfChange, Uri uri) {  
2     synchronized (mLock) {  
3         handleSettingsChangedLocked();  
4     }  
5 }
```

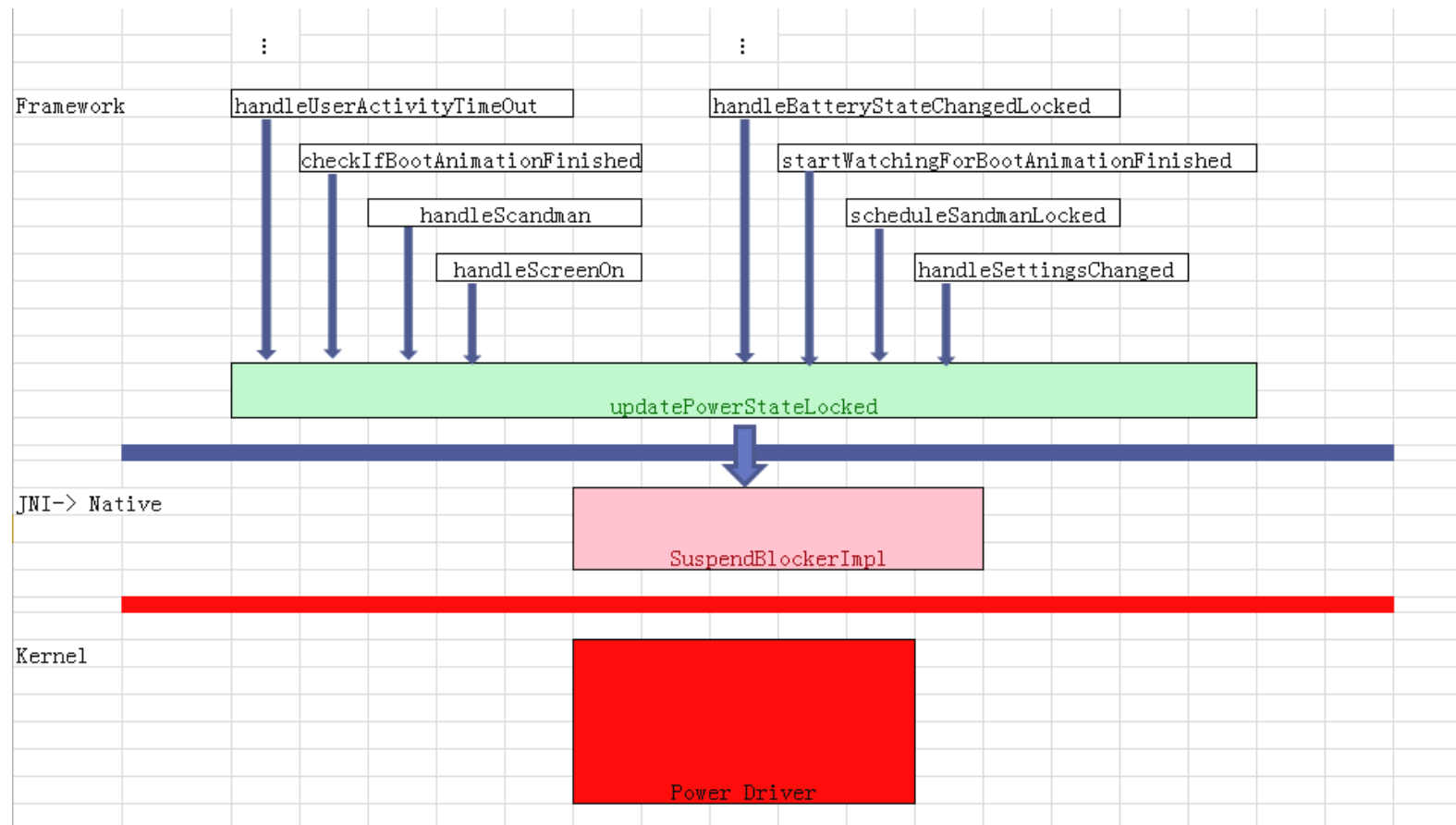
以上内容，说明PowerManagerService 不能能够接收用户的请求，被动地去做一些操作，还要主动监视系统中一些重要的属性的变化，和重要的事件的发生。

无论是处理主动还是被动的操作，在上面都一一列出了对应的处理函数。虽然对这些方法没有逐一说明，但是通过上面的goToSleepNoUpdateLocke的例子，

自己阅读下应该没有问题的。如果看过这些方法之后，你会发现一个很重要的共同点，就是PowerManagerService在处理各种各样的事件的时候，最终都会经过

这么一个方法updatePowerStateLocked。在上面这些内容中，我们说各种变化的时候，常用的一个词就是power state，而在updatePowerStateLocked方法

的名字中，我们很容易推测出这个方法要做的事情，就是把PowerManagerService中发生的变化，能够影响到Power Management的都要放在一起进行更新，让其真正地起作用。



这么说下去，还是有点空口白话的意味，我们还是从代码中一点一点去阅读效果会好些。

updatePowerStateLocked

```

1  /**
2   * Updates the global power state based on dirty bits recorded in mDirty.
3   *
4   * This is the main function that performs power state transitions.
  
```

```
5  * We centralize them here so that we can recompute the power state completely
6  * each time something important changes, and ensure that we do it the same
7  * way each time. The point is to gather all of the transition logic here.
8  */
9  private void updatePowerStateLocked() {
10     if (!mSystemReady || mDirty == 0) { //如果系统没有准备好, 或者power state没有发生任何变化, 这个方法可以不用执行的
11         return;
12     }
13
14     // Phase 0: Basic state updates.
15     updateIsPoweredLocked(mDirty);
16     updateStayOnLocked(mDirty);
17
18     // Phase 1: Update wakefulness.
19     // Loop because the wake lock and user activity computations are influenced
20     // by changes in wakefulness.
21     final long now = SystemClock.uptimeMillis();
22     int dirtyPhase2 = 0;
23     for (;;) {
24         int dirtyPhase1 = mDirty;
25         dirtyPhase2 |= dirtyPhase1;
26         mDirty = 0;
27
28         updateWakeLockSummaryLocked(dirtyPhase1); //在前面解释几个变量的时候, 就已经提到了WakeLockSummary和UserActivitySummary,
29         updateUserActivitySummaryLocked(now, dirtyPhase1); //在这里的两个方法中已经开始用到了。想必通过方法名, 大概也已经有所了解其
功能了。
30         if (!updateWakefulnessLocked(dirtyPhase1)) {
31             break;
32         }
33     }
34
35     // Phase 2: Update dreams and display power state.
36     updateDreamLocked(dirtyPhase2);
37     updateDisplayPowerStateLocked(dirtyPhase2);
38
39     // Phase 3: Send notifications, if needed.
40     if (mDisplayReady) {
41         sendPendingNotificationsLocked();
42     }
43
44     // Phase 4: Update suspend blocker.
45     // Because we might release the last suspend blocker here, we need to make sure
46     // we finished everything else first!
47     updateSuspendBlockerLocked();
48 }
```




从这段代码中，很容易就看出，这个方法对于power state的更新时分成四个阶段进行的。从注释中看到，第一阶段：基本状态的更新；

第二阶段：显示内容的更新； 第三阶段：dream和display状态的更新；第四阶段：suspend blocker的更新。之所以放在最后一步才进行suspend blocker的更新，

是因为在这里可能会释放suspend blocker。

对这个方法有了大概的了解之后，我们开始这个PowerManagerService中重要的函数进行分析吧，先看第一阶段的更新：

updateIsPoweredLocked开始，这个方法的功能是判断设备是否处于充电状态中，如果DIRTY_BATTERY_STATE发生了变化，说明设备的电池的状态有过改变，

然后通过对比和判断（通过电池的状态前后的变化和充电状态的变化来判断），确定是否处于在充电，充电方式的改变也会在mDirty中标记出来。同时根据充电状态

的变化进行一些相应的处理，同时是否在充电或者充电方式的改变都会认为是一次用户事件或者叫用户活动的发生。

updateStayOnLocked用来更新device是否开启状态。也是通过mStayOn的前后变化作为判断依据，如果device的属性Settings.Global.STAY_ON_WHILE_PLUGGED_IN

为置位，并且没有达到电池充电时持续开屏时间的最大值（也就是说，在插入电源后的一段时间内保持开屏状态），那么mStayOn为真。

上面这两个方法完成了第一阶段的更新，通过代码我们可以看到，主要是进行了充电状态的判断，然后根据充电的状态更新了一些必要的属性的变化，同时也在更新mDirty。

在看第二阶段是如何变化的：

在前面说到过mWakefulness是表示device处于的醒着或睡眠或两者之间的一种状态，这种状态会影响到wake lock和user activity的计算，所以要更新。第二阶段是通过一个

死循环进行了，只有当updateWakefulnessLocked返回为false的时候，才能跳出这个循环。刚刚进入这个循环的时候，把mDirty进行了重置，这点从侧面说明了这次updatePowerState

之后，会把前面所有发生的power state执行，不会让其影响到下一次的变化。同时也在为下一次的power state从头开始更新做好准备。updateWakeLockSummaryLocked和

updateUserActivitySummaryLocked代码很容易明白，要注意的是在updateUserActivitySummaryLocked在中锁屏时间和变暗时间的的比较。假如说在系统中设置的睡眠时间是30s，

而在PowerManagerService中默认的SCREEN_DIM_DURATION是7s，这就意味着：如果没有用户活动的话，在第23s，设备的屏幕开始变换，持续7s时间，然后屏幕开始关闭。

下面就是开始看看，在何时才能跳出这个循环，主要就是看updateWakefulnessLocked的返回值，先看看其代码：

```

1  /**
2   * Updates the wakefulness of the device.
3   *
4   * This is the function that decides whether the device should start napping//这个方法的功能是：根据当前的wakeLocks和用户的活动
情况，来决定设备是否需要小憩
5   * based on the current wake locks and user activity state. It may modify mDirty
6   * if the wakefulness changes.
7   *
8   * Returns true if the wakefulness changed and we need to restart power state calculation.
9   *///当wakefulness发生变化的时，返回true，同时也需要重新计算power state
10 private boolean updateWakefulnessLocked(int dirty) {
11     boolean changed = false;
12     if ((dirty & (DIRTY_WAKE_LOCKS | DIRTY_USER_ACTIVITY | DIRTY_BOOT_COMPLETED
13         | DIRTY_WAKEFULNESS | DIRTY_STAY_ON | DIRTY_PROXIMITY_POSITIVE
14         | DIRTY_DOCK_STATE)) != 0) {
15         if (mWakefulness == WAKEFULNESS_AWAKE && isItBedTimeYetLocked()) {
16             if (DEBUG_SPEW) {
17                 Slog.d(TAG, "updateWakefulnessLocked: Bed time...");
18             }
19             final long time = SystemClock.uptimeMillis();
20             if (shouldNapAtBedTimeLocked()) {
21                 changed = napNoUpdateLocked(time);
22             } else {
23                 changed = goToSleepNoUpdateLocked(time,
24                     PowerManager.GO_TO_SLEEP_REASON_TIMEOUT);
25             }
26         }
27     }
28     return changed;
29 }

```

先看函数isItBedTimeYetLocked，通过名字看，是在询问是否到了应该上床睡觉的时间了。然后结合line 15整个判断来看，如果现在设备处于醒着的状态，但是到了该睡眠的时间了，

就要进行如下操作。那么我们就来看看设备是判断是否该睡眠的：



```
1 private boolean isItBedTimeYetLocked() {
2     return mBootCompleted && !isBeingKeptAwakeLocked(); //个人认为mBootCompleted很重要，但是在设备正常使用过程中我们可以认为其值是true。现在还没有必要讨论其他的情况
3 }
4
5 /**
6  * Returns true if the device is being kept awake by a wake lock, user activity
7  * or the stay on while powered setting.
8  */
9 private boolean isBeingKeptAwakeLocked() {
10     return mStayOn
11         || mProximityPositive
12         || (mWakeLockSummary & WAKE_LOCK_STAY_AWAKE) != 0
13         || (mUserActivitySummary & (USER_ACTIVITY_SCREEN_BRIGHT
14             | USER_ACTIVITY_SCREEN_DIM)) != 0;
15 }
```



如果有应用程序持有wakelock，或者有用户活动的产生，或者处于充电状态，那么isBeingKeptAwakeLocked的返回值就是true，相应地isItBedTimeYetLocked

返回值就是false，说明还没有到睡眠的时间，因为还有wakelock没释放，或者有用户活动，或者是在充电等。但是，如果wakelock都释放了，并且也没有了用户活动了

也没有其他的顾虑了，那么就可以进入睡眠状态了。这时候我们就要考虑设备由醒着到睡眠的处理过程了。接着看代码updateWakefulnessLocked中的line20 ~ 25的内容，

在line 20中的方法代码如下

```
private boolean shouldNapAtBedTimeLocked() {
    return mDreamsActivateOnSleepSetting
        || (mDreamsActivateOnDockSetting
            && mDockState != Intent.EXTRA_DOCK_STATE_UNDOCKED);
}
```

mDreamsActivateOnSleepSetting的默认值为false，mDreamsActivateOnDockSetting的默认值为true。个人认为觉得Dock应该是类似于形似座充，或者能够接入汽车中的一个插孔吧，

具体是什么不是很了解。如果按我的理解，在一般的用户手中是没有接入Dock的，所以mDockState != Intent.EXTRA_DOCK_STATE_UNDOCKED应该为false。所以这个函数的返回值应该

是false的。这样的话，接下来执行的函数就是goToSleepNoUpdateLocked。这个方法在前面已经看到过了，当时是说这个方法只是更新了power state中一些必要的属性，并没有进行真正的

执行能够让device进入sleep的代码，真正的执行代码在updatePowerStateLocked方法中，可是现在到这里，又调用了goToSleepNoUpdateLocked方法，这不还是没能让设备进入sleep嘛，

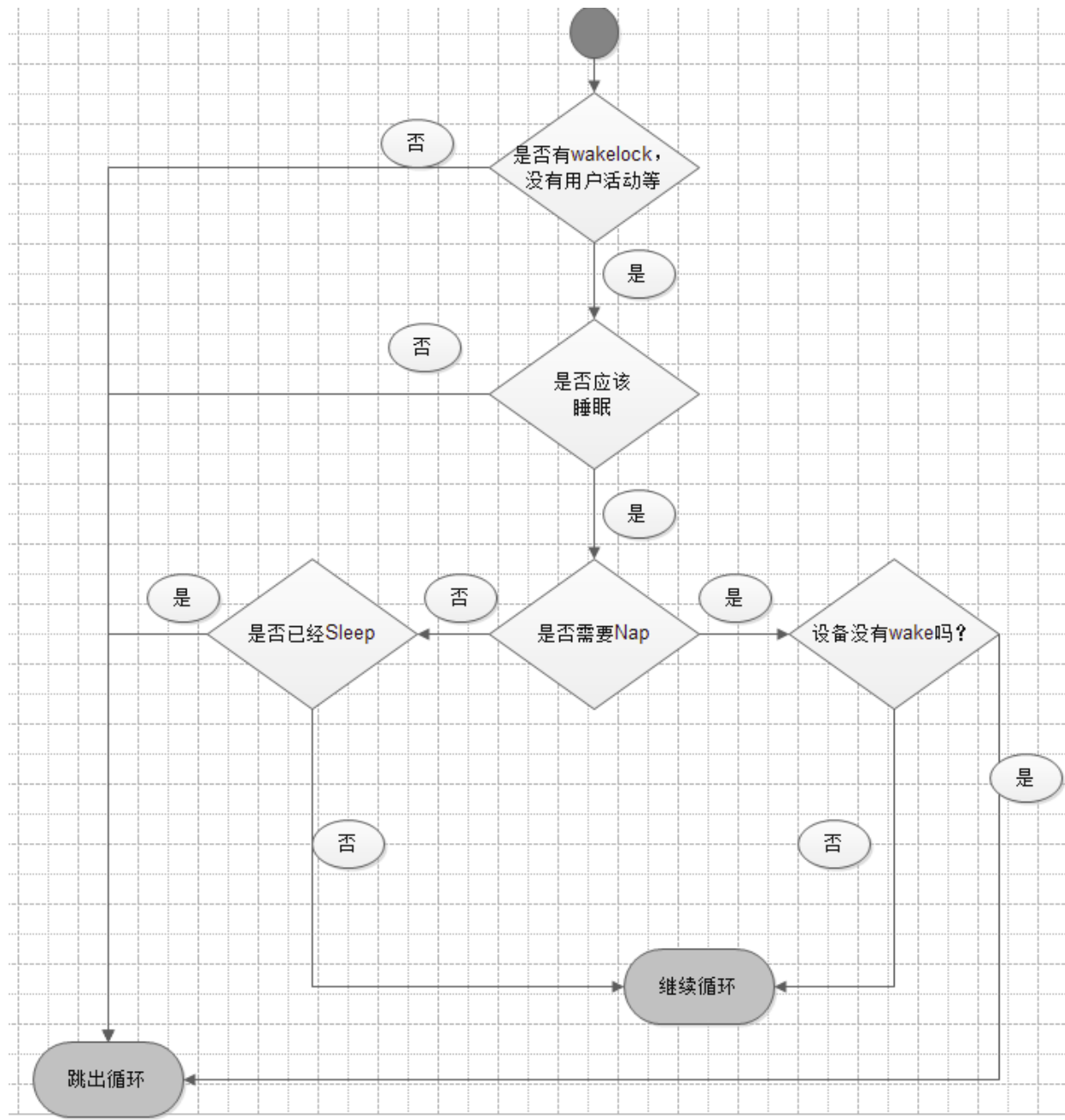
这到底是怎么回事？这个问题我们先放一放，接着往下看。大数学家华罗庚也经常使用这样的学习方法，如果在研究一个问题时，一时没能明白，不妨先放一放，接着往下读，说不定在研究后面的

问题时，你就会豁然开朗。让子弹飞一会。虽然我们一直假设shouldNapAtBedTimeLocked会返回false，但是他也是会返回为真的，比如只要开启Dreaming就行了，所以还有有必要看看

napNoUpdateLocked的实现挺简单的，就是在由醒着到应该睡眠之间的这个时间里，如果这个时间在醒着之前，或者设备不是醒着的状态，才会返回为false；其他情况都返回为true，所以我

个人认为在一般情况下这个方法是返回为true的。

到这里，对于第二阶段的power state更新就是叙述往了。整个过程，用下面一个图表，也许可以帮助大家的理解吧：



到这里为止，第二阶段的power state的更新叙述完成了。下面接着看看第三阶段的更新的内容吧：

updateDreamLocked(dirtyPhase2);根据mDirty的变化结合其他的属性一起判断是否要开始dreaming，其实就是开始屏保。如果需要开始屏保的话，通过DreamManagerService开始dreaming。

updateDisplayPowerStateLocked主要功能是每次都要重新计算一下display power state的值，即SCREEN_STATE_OFF,SCREEN_STATE_DIM,SCREEN_STATE_BRIGHT之一。此外，如果

在DisplayController中更新了display power state的话，DisplayController会发送消息通知我们，因此我们还要回来重新检查一次。我们可以看看updateDisplayPowerStateLocked是如何实现的：



```

1  private void updateDisplayPowerStateLocked(int dirty) {
2      if ((dirty & (DIRTY_WAKE_LOCKS | DIRTY_USER_ACTIVITY | DIRTY_WAKEFULNESS
3          | DIRTY_ACTUAL_DISPLAY_POWER_STATE_UPDATED | DIRTY_BOOT_COMPLETED
4          | DIRTY_SETTINGS | DIRTY_SCREEN_ON_BLOCKER_RELEASED)) != 0) {
5          int newScreenState = getDesiredScreenPowerStateLocked(); //获取display 的power state将要变成的状态
6          if (newScreenState != mDisplayPowerRequest.screenState) { //mDisplayPowerRequest.screenState是目前display所处的
power state
7              if (newScreenState == DisplayPowerRequest.SCREEN_STATE_OFF
8                  && mDisplayPowerRequest.screenState
9                  != DisplayPowerRequest.SCREEN_STATE_OFF) { //这个判断意味着：目前display的电源状态不是OFF，但是想要变
为OFF
10                 mLastScreenOffEventElapsedRealTime = SystemClock.elapsedRealtime();
11             }
12
13             mDisplayPowerRequest.screenState = newScreenState;
14             nativeSetPowerState(
15                 newScreenState != DisplayPowerRequest.SCREEN_STATE_OFF,
16                 newScreenState == DisplayPowerRequest.SCREEN_STATE_BRIGHT);
17         }
18
19         int screenBrightness = mScreenBrightnessSettingDefault;
20         float screenAutoBrightnessAdjustment = 0.0f;
21         boolean autoBrightness = (mScreenBrightnessModeSetting ==
22             Settings.System.SCREEN_BRIGHTNESS_MODE_AUTOMATIC); //获取屏幕亮度模式是否为自动变化
23         if (isValidBrightness(mScreenBrightnessOverrideFromWindowManager)) { //mScreenBrightnessOverrideFromWindowManager
是WindowManager设置的亮度大小，默认值为-1
24             screenBrightness = mScreenBrightnessOverrideFromWindowManager;
25             autoBrightness = false;
26         } else if (isValidBrightness(mTemporaryScreenBrightnessSettingOverride))
{ //mTemporaryScreenBrightnessSettingOverride在widget中设置的临时亮度大小，默认为-1
27             screenBrightness = mTemporaryScreenBrightnessSettingOverride;
28         } else if (isValidBrightness(mScreenBrightnessSetting)) { //在Settings中的设置的默认亮度，在android4.2中其值为102
29             screenBrightness = mScreenBrightnessSetting;

```

```
30     }
31     if (autoBrightness) { //如果亮度是自动调节的话
32         screenBrightness = mScreenBrightnessSettingDefault;
33         if (isValidAutoBrightnessAdjustment(
34             mTemporaryScreenAutoBrightnessAdjustmentSettingOverride)) {
35             screenAutoBrightnessAdjustment =
36                 mTemporaryScreenAutoBrightnessAdjustmentSettingOverride;
37         } else if (isValidAutoBrightnessAdjustment(
38             mScreenAutoBrightnessAdjustmentSetting)) {
39             screenAutoBrightnessAdjustment = mScreenAutoBrightnessAdjustmentSetting;
40         }
41     }
42     screenBrightness = Math.max(Math.min(screenBrightness,
43         mScreenBrightnessSettingMaximum), mScreenBrightnessSettingMinimum);
44     screenAutoBrightnessAdjustment = Math.max(Math.min(
45         screenAutoBrightnessAdjustment, 1.0f), -1.0f);
46     mDisplayPowerRequest.screenBrightness = screenBrightness; //从这行向下开始就是配置完成DisplayPowerRequest，然后以此为参
数通过requestPowerState方法进行设置。
47     mDisplayPowerRequest.screenAutoBrightnessAdjustment =
48         screenAutoBrightnessAdjustment;
49     mDisplayPowerRequest.useAutoBrightness = autoBrightness;
50
51     mDisplayPowerRequest.useProximitySensor = shouldUseProximitySensorLocked();
52
53     mDisplayPowerRequest.blockScreenOn = mScreenOnBlocker.isHeld();
54
55     mDisplayReady = mDisplayPowerController.requestPowerState(mDisplayPowerRequest,
56         mRequestWaitForNegativeProximity);
57     mRequestWaitForNegativeProximity = false;
58
59     if (DEBUG_SPEW) {
60         Slog.d(TAG, "updateScreenStateLocked: mDisplayReady=" + mDisplayReady
61             + ", newScreenState=" + newScreenState
62             + ", mWakefulness=" + mWakefulness
63             + ", mWakeLockSummary=0x" + Integer.toHexString(mWakeLockSummary)
64             + ", mUserActivitySummary=0x" + Integer.toHexString(mUserActivitySummary)
65             + ", mBootCompleted=" + mBootCompleted);
66     }
67 }
68 }
```

根据代码中的注释，这个方法阅读起来应该是没有问题的。其中代码line 55 ~56行实现了对屏幕亮度的请求，并改变了亮度。在这里还记得在前面说到的让子弹飞一会吗？现在子弹飞到这里了。

在goToSleepNoUpdateLocked函数中，我们把DisplayPowerState的状态更新为SCREEN_OFF了，然后就没做什么了，到这里之后，这个display state会因为requestPowerState的调用

而起作用。这个方法的具体实现时在DisplayPowerStateController中。在后面分析实例的时会说到这个方法。到这里，对于第三阶段的power state的更新已经完成了。

接下来就是最后一个阶段了的power state的更新了。这里对于SuspendBlocker的更新很简单，仅仅是判断现在device是否需要持有CPU或者是否需要CPU继续运行，如果有WakeLock没有释放，

或者还有用户活动的话，或者屏幕没有关闭的话等等，这是肯定是需要持有CPU的。所以这里就是更具需求去申请或者释放SuspendBlocker。

到这里，对于PowerManagerService的工作应该有了大致的了解，而且我们也知道了PowerManagerService在Framework中是如何实现的，代码是如何工作的。知道这些之后，对于后面这些问题的分析和解决都是非常有帮之的。

(2) 线程的角度

以上，叙述了这么多的内容，看起来十分地凌乱，不过总的来说，就是想告诉大家PowerManagerService的功能是什么，还有就是这么主要的功能是如何实现。了解了这些之后，我们不过是知道了

PowerManagerService这个类及其功能，就像是看到了一个事物，我们通过仔细观察，知道了这个事物大概能干些什么。也许，了解这些对于某些使用而言这就足够了，但是我们还不知道这个事物

从何而来，也不知道这个事物将向何处发展，如果不能了解从哪里来到哪里去的问题，很难从整体去把握整个事物的发展趋势，把握其内在的本质及规律。所以，为了了解PowerManagerService的

的运行流程，还需要换个角度去看待PowerManagerService。这次选择的就是从线程的角度去分析PowerManagerService。之所以是线程，是因为PowerManagerService是SystemServer进程中

的一部分，并没有独立的进程，但是PowerManagerService仍有一些单独的线程和Power处理相关的内容。好吧，我们从线程的角度是为了了解PowerManagerService的整个运行流程，所以就从

SystemServer中PowerManagerService对象的创建开始吧。

PowerManagerService的对象是在SystemServer创建的，然后在SystemServer的主线程中做了一些初始化工作，主要的初始化工作是通过以下这些方法完成的：

(1) power = new PowerManagerService();

(2) power.init(context, lights, ActivityManagerService.self(), battery, BatteryStatsService.getService(), display);

(3) power.systemReady(twilight, dreamy);

在systemServer的主线程中和PowerManagerService相关的内容大概就这些。上面这些方法中不包括把PowerManagerService作为参数来构造其他的对象，进而与PowerManagerService进行交互的。

在文章最后，在仔细分析使用power作为参数构造出的对象中对PowerManagerService的操作。接下来先看PowerManagerService的构造函数，其实在SystemServer的主线程中，和PowerManagerService

初始化相关的仅有两个方法，分别构造函数和init方法，我先来看看构造函数的代码，内容如下




```
1 public PowerManagerService() {
2     synchronized (mLock) { //下面是对于PowerManagerService中一些必要的变量进行的创建
3         mWakeLockSuspendBlocker = createSuspendBlockerLocked("PowerManagerService");
4         mWakeLockSuspendBlocker.acquire(); //申请持有SuspendBlocker，防止系统停止工作
5         mScreenOnBlocker = new ScreenOnBlockerImpl();
6         mDisplayBlanker = new DisplayBlankerImpl();
7         mHoldingWakeLockSuspendBlocker = true;
8         mWakefulness = WAKEFULNESS_AWAKE;
9     }
10
11     nativeInit(); //初始化native层的PowerManagerService，等会着重分析下这个函数
12     nativeSetPowerState(true, true); //这个方法是用来初始化两个全局变量的，一个是gScreenOn，一个是gScreenBright，用来标示屏幕是否开启，是否亮着
13 }
```




在上面的代码中，我们可以看出构造函数其实挺简单的，在初始化了必要的变量之后，还调用了一个native方法nativeInit()，想必是用来初始化native层的一些必要的属性的。

我还是先看看nativeInit的代码吧：



```
1 static void nativeInit(JNIEnv* env, jobject obj) {
2     gPowerManagerServiceObj = env->NewGlobalRef(obj);
3     //在阅读和HAL相关的代码是，经常见到如下这个方法，和hw_module_t这个结构体
4     status_t err = hw_get_module(POWER_HARDWARE_MODULE_ID, // #define POWER_HARDWARE_MODULE_ID "power"
5                                 (hw_module_t const**)&gPowerModule); // power_module* gPowerModule
6     if (!err) {
7         gPowerModule->init(gPowerModule);
8     } else {
9         ALOGE("Couldn't load %s module (%s)", POWER_HARDWARE_MODULE_ID, strerror(-err));
10     }
```

```
10     }  
11 }  

```

在阅读和HAL相关的代码时，经常会见到hw_get_module这个方法，以及hw_module_t这个结构体，这次在这里我们要仔细看看这个方法和结构体是怎么回事。

在hw_get_module之前，我们还是先弄清楚方法中的两个参数的含义为好。

(3) 与用户交互的角度

通过几个事件来分析，PowerManagerService与用户之间的交互，

首先是按下手机Power键之后的事件处理流程；

其次是调节Brightness与PowerManagerService之间的关系

最后，睡眠时间和没有用户输入时，手机逐渐进入睡眠状态的变化流程。

使用PowerManagerService作为参数构造出的对象中,对PowerManagerService的操作,这些对象如下 :

- 1) Watchdog.getInstance().init(context, battery, power, alarm,ActivityManagerService.self());
- 2)wm = WindowManagerService.main(context, power, display, inputManager, uiHandler, wmHandler, factoryTest != SystemServer.FACTORY_TEST_LOW_LEVEL,!firstBoot, onlyCore);
- 3)networkPolicy = **new** NetworkPolicyManagerService(context, ActivityManagerService.self(), power, networkStats, networkManagement);

分类: Android

好文要顶

关注我

收藏该文



Balancor

关注 - 4

粉丝 - 14

+加关注

1

« 上一篇 : Android4.0中MediaPlayer 和 MediaPlayerService

» 下一篇 : Android Framework-----之ActivityManagerService与Activity之间的通信

posted @ 2013-08-26 17:37 Balancor 阅读(9574) 评论(3) 编辑 收藏

评论列表

#1楼 2013-10-23 22:06 p10000n

这些内容是4.2或4.3的电源管理内容，分析的透彻。

支持(0) 反对(0)

#2楼 2014-03-11 09:48 lxlp

请问博主，显示超时后屏幕黑掉了，但是开启时没有显示锁屏界面，而是直接进到系统，按电源键是可以锁屏的，请问是什么原因

支持(0) 反对(0)

#3楼 2014-05-28 14:42 云且留猪

言语罗嗦，逻辑混乱，前面说要把一些东西放在后面讲，到最后也没讲。

支持(1) 反对(0)

[刷新评论](#) [刷新页面](#) [返回顶部](#)

注册用户登录后才能发表评论，请 [登录](#) 或 [注册](#)，[访问网站首页](#)。

【推荐】50万行VC++源码: 大型组态工控、电力仿真CAD与GIS源码库

【推荐】Google+滴滴联手打造Android开发工程师课程

【推荐】群英云服务器性价比王，2核4G5M BGP带宽 68元首月！



最新IT新闻:

- 这款雨伞无需用手支撑 雨天摄影利器
 - 加州车管局：苹果已被批准测试无人驾驶汽车
 - 在奥克兰的破仓库中 这些CEO和谷歌员工周末玩无人赛车
 - 亚马逊新计划大力推广智能扬声器 与谷歌对抗升级
 - 谷歌生命科学公司Verily发布健康跟踪智能手表
- » 更多新闻...



最新知识库文章:

- 程序员，如何从平庸走向理想？
- 我为什么鼓励工程师写blog
- 怎么轻松学习JavaScript
- 如何打好前端游击战
- 技术文章的阅读姿势
- » 更多知识库文章...

Copyright ©2017 Balancor