自從接觸Android系統已經一年多了，這段時間內對於Android系統的Framework層的各個模塊都有過接觸，有時也做過分析，但是一直沒能形成一個總結性的東西。這次下定決心，好好整理整理對於Android系統的學習梳理一下自己的思路。本文一方面是為了自己梳理下知識，文中涉及的內容，基本是拾人牙慧，很少有自己的東西，最多也就算是自己的總結；除此作用之外，如果能為後來者引玉，也算是一點功德吧。這次首先是對Android系統中的PowerManagerService進行下整理。之所以先選擇PowerManagerService，是因為這個模塊相對於Android系統中其他的模塊而言，與系統其他的模塊之間的交互較少，而且Framework中的PowerManagerService模塊是由Google開發並維護的，雖然以Linux Kernel的Power為基礎，但是它們之間的耦合度低，完全可以把兩者分開，單獨進行分析也不會造成困惑。接下來，我會從不同的角度，分別介紹下PowerManagerService的功能。還是先來看看PowerManagerService在Framework中的目錄結構吧。PowerManagerService在Android4.2源碼中的位置是：/frameworks/base/services/java/com/android/server/power/，在這個目錄下有以下文件：

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

這些文件中，個人認為對於PoweManagerService而言除了本身的代碼，較為重要的有DisplayPowerController.java,DisPlayPowerState.java， Notifier.java.
而DisplayPowerRequest相當於一個輔助類，用來存儲一些統一的屬性和變量，讓PowerManagerService和DisplayPoerController, DisplayPowerState交互時
能夠使用統一的變量。另外，還有及個接口文件SuspendBlocker.java,DisplayBlanker.java, ScreenOnBlanker.java。其次就是ShutdownThread.java, WirelessChargerDetector.java,RampAnimator.java, ElectronBeam.java. 下面，就先逐一介紹下PowerManagerService在Framework中的這些文件：
PowerManagerService.java: 主要是計算系統中和Power相關的計算，然後決策系統應該如何反應。同時協調Power如何與系統其它模塊的交互，比如沒有用戶
活動時，屏幕變暗等等。
DisplayPowerController.java：管理Display設備的電源狀態。僅在PowerManagerService中實例化了一

個對象，它算是PowerManagerService的一部分，只不過是獨立出來了而已。主要處理和距離傳感器，燈光傳感器，以及包括關屏在內的一些動畫，通過異步回調的方式來通知PowerManagerService某些事情發生了變化。

DisPlayPowerState.java：在本質上類似於View，只不過用來描述一個display的屬性，當這些屬性發生變化時，可以通過一個序列化的命令，讓這些和display電源狀態的屬性一起產生變化。這個類的對像只能被DispalyPowerController的Looper持有。而這個Looper應該就是PowerManagerService中新建的一個HandlerThread中的Looper。和PowerManager相關的，包括DisplayPowerState和DisplayPowerController相關的消息處理應該都可以通過這個HandlerThread進行運轉的。

Notifier.java: 將Power Manager state的重要變化通過broadcast發送出去。

接下來就說說三個接口文件

SuspendBlocker.java： 相當於一個partial wake lock。在內部使用，避免了使用上層的喚醒鎖機制

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.（不知道這兩個英文的解釋是否恰當，還有待驗證）。

ScreenOnBlanker.java：描述了一種較為低級的blocker機制，主要用於關屏或者隱藏屏幕內容，直到window manager準備好新的內容

DisplayPowerRequest.java：描述了一些對於display電源狀態的請求。

最後看看剩餘的這些類：

ShutDownThread.java: 主要功能就是關機和重啟，當我想要執行重啟或者關機的時候，這個新城就會被啟動。

ElectronBeam.java： 負責屏幕由開到關，或者由關到開的一些GL動畫。在DisplayPowerController管理

WirelessChargerDetector.java：和無線充電相關的東西，沒有細看。

每個文件的大致功能就是這樣的，也許有些地方不是很恰當，還是需要仔細閱讀源碼，才能確切地知道到底是怎麼回事，有些功能到底是如何實現的。

先從交互的角度去看看PowerManagerService的功能。在這裡的交互是說PowerManagerService與應用程序或者Framework中其他的模塊的交互，而不是指和用戶之間的直接交互。和用戶之間牽涉到交互的內容，在文章的最後也稍微有點介紹。下面就分成兩個小節，對於PowerManagerService的交互作以總結。首先：

a). 與應用程序之間的交互

在Android中應用程序並不是直接同PowerManagerService交互的，而是通過PowerManager間接地與PowerManagerService打交道。不過在使用PowerManager和PowerManager,WakeLock之前，我們要首先在APP中申請使用如下權限：

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

而APP能夠與PowerManager做哪些交互，在Android提供的開發文檔中給了我們答案。我們可以看到

PowerManage提供瞭如下公共的接口：

PowerManager	PowerManagerService
goToSleep(long time)	goToSleep(long eventTime, int reason)
isScreenOn()	isScreenOn()
reboot(String reason)	reboot(boolean confirm, String reason, boolean wait)
userActivity(long when, boolean noChangeLights)	userActivity(long eventTime, int event, int flags)
wakeUp(long time)	wakeUp(long eventTime)

在這個表格中，僅僅列出了PowerManager的公開方法中的其中五個，同時列出在PowerManagerService中對應的方法。這裡列出的，是與PowerManagerService關 系比較緊密的方法，其餘的和PowerManager相關的東西會在接下來，慢慢地都談到的。如果閱讀PowerManager的源碼的話，你會很容易發現，其實PowerManager的方法在實現的過程中，都是通過調用PowerManagerService相應的函數來實現的。PowerManager就像是PowerManagerService的"代理類"。這裡略過PowerManager是如何通過binder與PowerManagerService進行通信的。下面，我們逐一對PowerManagerService中這幾個函數的實現進行下簡單的分析，在對這些函數分析之前，我覺得還是先對代碼中使用的一些變量作以簡要的說明為好。其實，在PowerManagerService中絕大部分變量通過名字就能大概知道其意義，不過還有幾個較為重要的還是仔細說說為好，首先是重要的變量mDirty,根據代碼的註釋是說，用來表示power state的變化，而 這樣的變化在系統中一共定義了12個，每一個state對應一個固定的數字，都是2的倍數。這樣的話，當有若干個狀態一起變化時，他們按位取或，這樣得到的結果既是唯一的，又能准去標示出各個狀態的變化。此外還有一個mWakefulness的變量，它用來標示的是device處於的一種狀態，是醒著的還是睡眠中，或者處於兩者之間的一種狀態。這個狀態是和display的電源狀態是不同的，display的電源狀態是獨立管理的。這個變量用來標示DIRTY_WAKEFULNESS這個power state下的一個具體的內容。比如說，系統從進入Draaming的時候，首先變化的是mDirty，在mDirty中對DIRTY_WAKEFULNESS位置位，這說明系統中的DIRTY_WAKEFULNESS發生了變化；此時，僅僅是知道DIRTY_WAKEFULNESS發生了變化，但是不知道wakefulness到底發生了怎樣的變化，如果需要進一步知道系統的wakefulness變成了什麼，就需要查看下mWakefulness的內容就知道了。相當於是對DIRTY_WAKEFULNESS的一個補充說明吧。像這樣的算是補充性質的變量還有mWakeLockSummary和mUserActivitySummary。好了，接下來我們可以從goToSleep(long eventTime, int reason)開始了，代碼如下：

```
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               //權限檢查
7
mContext.enforceCallingOrSelfPermission(android.Manifest.permission.DEVICE_POWER, null);
8
9 final long ident = Binder.clearCallingIdentity();
10 try {
11 goToSleepInternal(eventTime, reason); //這裡會調用函數的實現，在PowerManagerService中有很多類似的使用方式，之後的代碼中我會直接列出對應方法的實現
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 }
```

對於文中的代碼，我會在不影響閱讀的情況下，儘量地少。在這段代碼中，涉及到另個重要的函數goToSleepNoUpdateLocked() 和updatePowerStateLocked()，而goToSleepNoUpdateLocked是goToSleep功能的計算者，來決定是否要休眠，而updatePowerStateLocked函數算是功能的執行者，而且這個執行者同時負責執行了很多其他的功能，在總結的時候會著重分析這個函數。這裡先看goToSleepNoUpdateLocked方法的代碼：

```
1     private boolean goToSleepNoUpdateLocked( long eventTime, int reason) {
2         if (DEBUG_SPEW) {
3             Slog.d(TAG, "goToSleepNoUpdateLocked: eventTime=" + eventTime + ", reason=" + reason);
4         }
5
6         if (eventTime < mLastWakeTime || mWakefulness == WAKEFULNESS_ASLEEP
7                 || !mBootCompleted || ! mSystemReady) {
```

```
 8            return false ;
 9        }
 10
11        switch (reason) {
 12            case PowerManager.GO_TO_SLEEP_REASON_DEVICE_ADMIN:
 13                Slog.i(TAG, "Going to sleep due to device administration policy..." );
 14                break ;
 15            case PowerManager.GO_TO_SLEEP_REASON_TIMEOUT:
 16                Slog.i(TAG, "Going to sleep due to screen timeout..." );
 17                break ;
 18            default :
 19                Slog.i (TAG, "Going to sleep by user request..." );
 20                reason = PowerManager.GO_TO_SLEEP_REASON_USER;
 21                break ;
 22        }
 23
24        sendPendingNotificationsLocked();
 25        mNotifier.onGoToSleepStarted(reason);
 26          mSendGoToSleepFinishedNotificationWhenReady = true ;
 27
28          mLastSleepTime = eventTime;
 29          mDirty |= DIRTY_WAKEFULNESS;
 30          mWakefulness = WAKEFULNESS_ASLEEP;
 31
32        // Report the number of wake locks that will be cleared by going to sleep.
33        int numWakeLocksCleared = 0 ;
 34        final int numWakeLocks = mWakeLocks.size();
 35        for ( int i = 0; i < numWakeLocks; i++ ) {
 36            final WakeLock wakeLock = mWakeLocks.get(i);
 37            switch (wakeLock.mFlags &
PowerManager.WAKE_LOCK_LEVEL_MASK) {
 38                case PowerManager.FULL_WAKE_LOCK:
 39                case PowerManager.SCREEN_BRIGHT_WAKE_LOCK :
 40                case PowerManager.SCREEN_DIM_WAKE_LOCK:
 41                    numWakeLocksCleared += 1 ;
 42                    break ;
 43            }
 44        }
 45        EventLog.writeEvent(EventLogTags.POWER_SLEEP_REQUESTED,
numWakeLocksCleared);
 46          return true ;
 47    }
```

通過這段代碼發現，其實這裡並沒有真正地讓device進行sleep，僅僅只是把PowerManagerService中一些必要的屬性進行了賦值，等會在分析updatePowerStateLocked的時候，再給出解釋。在PowerManagerService的代碼中，有很多的方法的名字中都含有xxxNoUpdateLocked這樣的後綴，我覺得這樣做大概是因為，都類似於goToSleepNoUpdateLocked方法，並沒有真正地執行方法名字所描述的功能，僅僅是更新了一些必要的屬性。所以在Android系統中可以把多個power state屬性的多個變化放在一起共同執行的，而真正的功能執行者就是updatePowerStateLocked。

b).與系統其它模塊之間的交互

PowerManagerService作為Android系統Framework中重要的能源管理模塊，除了與應用程序交互之外，還要與系統中其它模塊配合，在提供良好的能源管理同時提供友好的用戶體驗。

Android系統除了提供公共接口與其它模塊交互外，還提供BroadCast機制，用來對系統中發生的重要變化做出反應。下表列出了，在PowerManagerService中註冊的Receiver，以及這些Receiver監聽的事件，和處理方法：

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中除了註冊了這五個Receiver之外，還定義了一個SettingsObserver，用於監視系統中以下屬性的變化：

SCREENSAVER_ENABLE，屏保的功能開啟

SCREENSAVER_ACTIVE_ON_SLEEP，在睡眠時屏保啟動

SCREENSAVER_ACTIVE_ON_DOCK，連接底座並且屏保啟動

SCREEN_OFF_TIMEOUT，休眠時間

STAY_ON_PLUGGED_IN，有插入並且屏幕開啟

SCREEN_BRIGHTNESS，屏幕的亮度

SCREEN_BRIGHTNESS_MODE，屏幕亮度的模式

當以上這些屬性發生變化時，SettingObserver都會監視到，並且調用SettingObserver的onChange方法，

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

以上內容，說明PowerManagerService 不能能夠接收用戶的請求，被動地去做一些操作，還要主動監視系統中一些重要的屬性的變化，和重要的事件的發生。

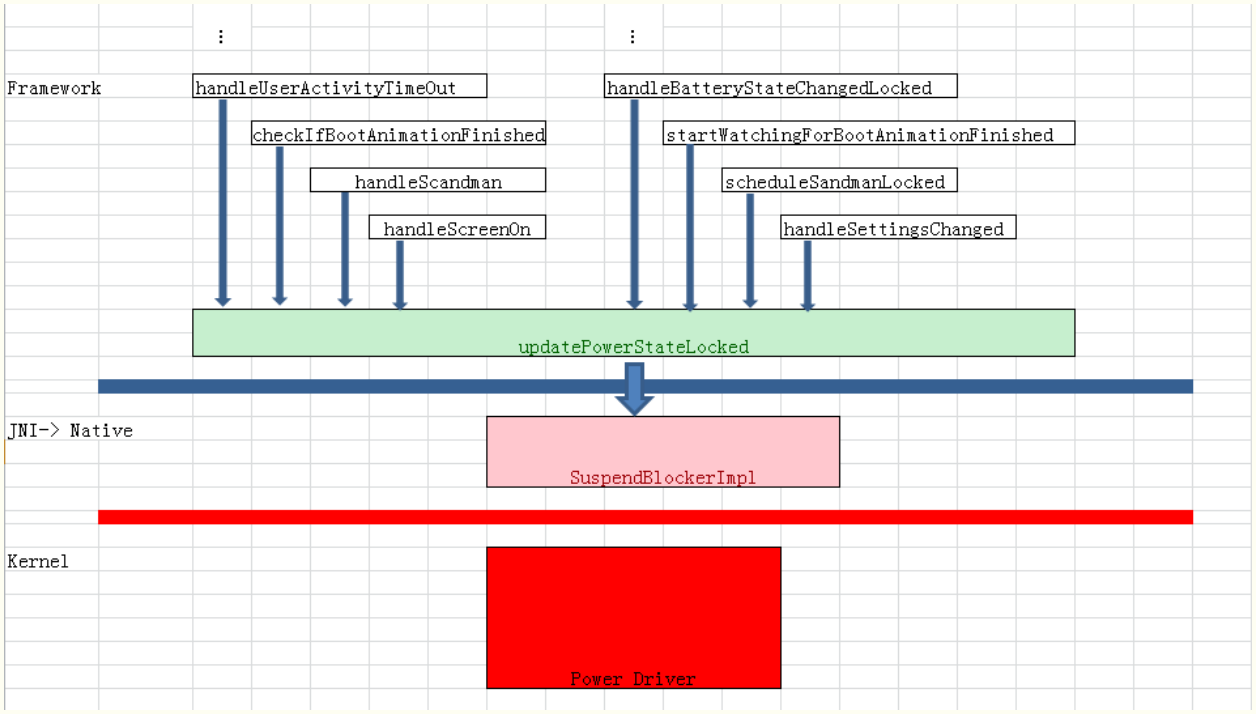
無論是處理主動還是被動的操作，在上面都一一列出了對應的處理函數。雖然對這些方法沒有逐一說明，但是通過上面的goToSleepNoUpdateLocke的例子，

自己閱讀下應該沒有問題的。如果看過這些方法之後，你會發現一個很重要的共同點，就是PowerManagerService在處理各種各樣的事件的時候，最終都會經過

這麼一個方法updatePowerStateLocked。在上面這些內容中，我們說各種變化的時候，常用的一個詞就是power state，而在updatePowerStateLocked方法

的名字中，我們很容易推測出這個方法要做的事情，就是把PowerManagerService中發生的變化，能夠影響到Power Management的都要放在一起進行更新，讓

其真正地起作用。



這麼說下去，還是有點空口白話的意味，我們還是從代碼中一點一點去閱讀效果會好些。

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 ) { //如果系統沒有準備好，或者power state沒有發生任何變化，這個方法可以不用執行的
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);// 在前面解釋幾個變量的
時候，就已經提到了WakeLockSummary和UserActivitySummary，
29          updateUserActivitySummaryLocked(now, dirtyPhase1);// 在這裡的兩個方
法中已經開始用到了。想必通過方法名，大概也已經有所瞭解其功能了。
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     }
```

從這段代碼中，很容易就看出，這個方法對於power state的更新時分成四個階段進行的。從註釋中看到，第一階段： 基本狀態的更新；

第二階段：顯示內容的更新； 第三階段：dream和display狀態的更新；第四階段：suspend blocker的更新。之所以放在最後一步才進行suspend blocker的更新，

是因為在這裡可能會釋放suspend blocker。

對這個方法有了大概的瞭解之後，我們開始這個PowerManagerService中重要的函數進行分析吧，先看第

一階段的更新：
updateIsPoweredLocked開始，這個方法的功能是判斷設備是否處於充電狀態中，如果DIRTY_BATTERY_STATE發生了變化，說明設備的電池的狀態有過改變，
然後通過對比和判斷（通過電池的狀態前後的變化和充電狀態的變化來判斷），確定是否處於在充電，充電方式的改變也會在mDirty中標記出來。同時根據充電狀態
的變化進行一些相應的處理，同時是否在充電或者充電方式的改變都會認為是一次用戶事件或者叫用戶活動的發生。
updateStayOnLocked用來更新device是否開啟狀態。也是通過mStayOn的前後變化作為判斷依據，如果device的屬性Settings.Global.STAY_ON_WHILE_PLUGGED_IN
為置位，並且沒有達到電池充電時持續開屏時間的最大值（也就是說，在插入電源後的一段時間內保持開屏狀態），那麼mStayOn為真。
上面這兩個方法完成了第一階段的更新，通過代碼我們可以看到，主要是進行了充電狀態的判斷，然後根據充電的狀態更新了一些必要的屬性的變化，同時也在更新mDirty。
在看第二階段是如何變化的：
在前面說到過mWakefuless是表示device處於的醒著或睡眠或兩者之間的一種狀態，這種狀態會影響到wake lock和user activity的計算，所以要進行更新。第二階段是通過一個
死循環進行了，只有當updateWakefulnessLocked返回為false的時候，才能跳出這個循環。剛剛進入這個循環的時候，把mDirty進行了重置，這點從側面說明了這次updatePowerState
之後，會把前面所有發生的power state執行，不會讓其影響到下一次的變化。同時也在為下一次的power state從頭開始更新做好準備。updateWakeLockSummaryLocked和
updateUserActivitySummaryLocked代碼很容易明白，要注意的是在updateUserActivitySummaryLocked在中鎖屏時間和變暗時間的的比較。假如說在系統中設置的睡眠時間是30s，
而在PowerManagerService中默認的SCREEN_DIM_DURATION是7s，這就意味著：如果沒有用戶活動的話，在第23s，設備的屏幕開始變換，持續7s時間，然後屏幕開始關閉。
下面就是開始看看，在何時才能跳出這個循環，主要就是看updateWakefulnessLocked的返回值，先看看其代碼：

```
1    /**
2     * Updates the wakefulness of the device.
3     *
4     * This is the function that decides whether the device should start napping// 這個方法的功能是：根據當前的wakeLocks和用戶的活動情況，來決定設備是否需要小憩
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      *// / 當wakefulness發生變化的時，返回true，同時也需要重新計算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     }
```

先看函數isItBedTimeYetLocked，通過名字看，是在詢問是否到了應該上床睡覺的的時間了。然後結合line 15整個判斷來看，如果現在設備處於醒著的狀態，但是到了該睡眠的時間了，
就要進行如下操作。那麼我們就來看看設備是判斷是否該睡眠的：

```
1     private boolean isItBedTimeYetLocked() {
 2         return mBootCompleted && ! isBeingKeptAwakeLocked();// 個人認為
mBootCompleted很重要，但是在設備正常使用的過程中我們可以認為其值是true。現在還
沒有必要討論其他的情況
 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    }
```

如果有應用程序持有wakelock，或者有用戶活動的產生，或者處於充電狀態，那麼isBeingKeptAwakeLocked的返回值就是true，相應地isItBedTimeYetLocked返回值就是false，說明還沒有到睡眠的時間，因為還有wakelock沒釋放，或者有用戶活動，或者是在充電等。但是，如果wakelock都釋放了，並且也沒有了用戶活動了也沒有其他的顧慮了，那麼就可以進入睡眠狀態了。這時候我們就要考慮設備由醒著到睡眠的處理過程了。接著看代碼updateWakefulnessLocked中的line20 ~ 25的內容，

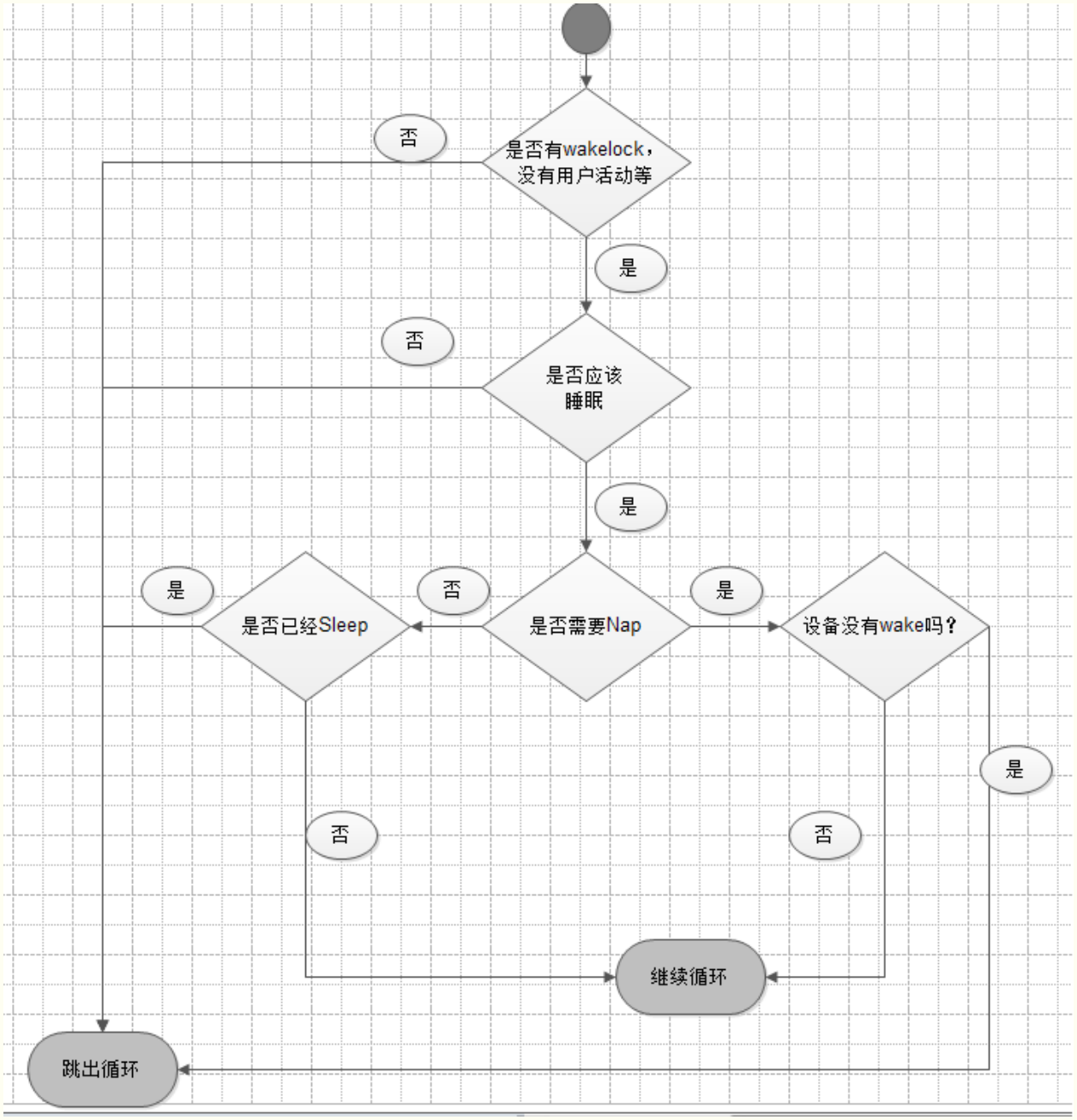
在line 20中的方法代碼如下

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

mDreamsActivateOnSleepSetting的默認值為false， mDreamsActivateOnDockSetting的默認值為true。個人認為覺得Dock應該是類似於形似座充，或者能夠接入汽車中的一個插孔吧，具體是什麼不是很瞭解。如果按我的理解，在一般的用戶手中是沒有接入Dock的，所以mDockState != Intent.EXTRA_DOCK_STATE_UNDOCKED應該為false。所以這個函數的返回值應該是false的。這樣的話，接下來執行的函數就是goToSleepNoUpdateLocked。這個方法在前面已經看到過了，當時是說這個方法只是更新了下power state中一些必要的屬性，並沒有進行真正的執行能夠讓device進入sleep的代碼，真正的執行代碼在updatePowerStateLocked方法中，可是現在到這裡，又調用了goToSleepNoUpdateLocked方法，這不還是沒能讓設備進入sleep嘛，這到底是怎麼回事？這個問題我們先放一放，接著往下看。大數學家華羅庚也經常使用這樣的學習方法，如果在研究一個問題時，一時沒能明白，不妨先放一放，接著往下讀，說不定在研究後面的問題時，你就會豁然開朗。讓子彈飛一會。雖然我們一直假設shouldNapAtBedTimeLocked會返回false，但是他也是會返回為真的，比如只要開啟Dreaming就行了，所以還有有必要看看napNoUpdateLocked的實現挺簡單的，就是在由醒著到應該睡眠之間的這個時間裡，如果這個時間在醒著之前，或者設備不是醒著的狀態，才會返回為false；其他情況都返回為true ，所以我個人認為在一般情況下這個方法是返回為true的。

到這裡，對於第二階段的power state更新就是敘述往了。整個過程，用下面一個圖表，也許可以幫助大家

的理解吧：



到這裡為止，第二階段的power state的更新敘述完成了。下面接著看看第三階段的更新的內容吧：updateDreamLocked(dirtyPhase2);根據mDirty的變化結合其他的屬性一起判斷是否要開始dreaming，其實就是開始屏保。如果需要開始屏保的話，通過DreamManagerService開始dreaming。updateDisplayPowerStateLocked主要功能是每次都要重新計算一下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的話，DisplayController會發送消息通知我們，因此我們還要回來重新檢查一次。我們可以看看updateDisplayPowerStateLocked是如何實現的：

```
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(); //獲取
display的power state將要變成的狀態
 6             if (newScreenState != mDisplayPowerRequest.screenState) {
//mDisplayPowerRequest.screenState是目前display所處的power state
 7                 if (newScreenState == DisplayPowerRequest.SCREEN_STATE_OFF
 8                         && mDisplayPowerRequest.screenState
 9                                 != DisplayPowerRequest .SCREEN_STATE_OFF) {// 這個判
斷意味著：目前display的電源狀態不是OFF，但是想要變為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);// 獲
取屏幕亮度模式是否為自動變化
 23              if (isValidBrightness(mScreenBrightnessOverrideFromWindowManager))
{// mScreenBrightnessOverideFromeWindowManager是WindowManager設置的亮度
大小，默認值為-1
 24                  screenBrightness =
mScreenBrightnessOverrideFromWindowManager;
 25                  autoBrightness = false ;
 26              } else if
(isValidBrightness(mTemporaryScreenBrightnessSettingOverride)) {//
mTemporaryScreenBrightnessSettingOverride在widget中中設置的臨時亮度大小，默認
```

```
為-1
 27                 screenBrightness = mTemporaryScreenBrightnessSettingOverride;
 28             } else if (isValidBrightness(mScreenBrightnessSetting)) {// 在Settings
中的設置的默認亮度，在android4.2中其值為102
 29                 screenBrightness = mScreenBrightnessSetting;
 30             }
 31             if (autoBrightness) {// 如果亮度是自動調節的話
 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;// 從這行
向下開始就是配置完成DisplayPowerRequest，然後以此為參數通過requestPowerSTate方
法進行設置。
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     }
```

根據代碼中的註釋，這個方法閱讀起來應該是沒有問題的。其中代碼line 55 ~56行實現了對屏幕亮度的請求，並改變了亮度。在這裡還記得在前面說到的讓子彈飛一會嗎？現在子彈飛到這裡了。

在goToSleepNoUpdateLocked函數中，我們把DisplayPowerState的狀態更新為SCREEN_OFF了，然後就沒做什麼了，到這裡之後，這個display state會因為requestPowerState的調用而起作用。這個方法的具體實現時在DisplayPowerStateController中。在後面分析實例的時會說到這個方法。到這裡，對於第三階段的power state的更新已經完成了。

接下來就是最後一個階段了的power state的更新了。這裡對於SuspendBlocker的更新很簡單，僅僅是判斷現在device是否需要持有CPU或者是否需要CPU繼續運行，如果有WakeLock沒有釋放，或者還有用戶活動的話，或者屏幕沒有關閉的話等等，這是肯定是需要持有CPU的。所以這裡就是更具需求去申請或者釋放SuspendBlocker。

到這裡，對於PowerManagerService的工作應該有了大致的瞭解，而且我們也知道了PowerManagerService在Framework中是如何實現的，代碼是如何工作的。知道這些之後，對於後面這些問題的分析和解決都是非常有幫之的。

(2) 線程的角度

以上，敘述了這麼多的內容，看起來十分地凌亂，不過總的來說，就是想告訴大家PowerManagerService的功能是什麼，還有就是這麼主要的功能是如何實現。瞭解了這些之後，我們不過是知道了PowerManagerService這個類及其功能，就像是看到了一個事物，我們通過仔細觀察，知道了這個事物大概能幹些什麼。也許，瞭解這些對於某些使用而言這就足夠了，但是我們還不知道這個事物從何而來，也不知道這個事物將向何處發展，如果不能瞭解從哪裡來到哪裡去的問題，很難從整體去把握整個事物的發展趨勢，把握其內在的本質及規律。所以，為了瞭解PowerManagerService的的運行流程，還需要換個角度去看待PowerManagerService。這次選擇的就是從線程的角度去分析PowerManagerService。之所以是線程，是因為PowerManagerService是SystemServer進程中的一部分，並沒有獨立的進程，但是PowerManagerService仍有一些單獨的線程和Power處理相關的內容。好吧，我們從線程的角度是為了瞭解PowerManagerService的整個運行流程，所以就從SystemServer中PowerManagerService對象的創建開始吧。

PowerManagerService的對像是在SystemServer創建的，然後在SystemServer的主線程中做了一些初始化工作，主要的初始化工作是通過以下這些方法完成的：

（1）power = new PowerManagerService();

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

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

在systemServer的主線程中和PowerManagerService相關的內容大概就這些。上面這些方法中不包括把PowerManagerService作為參數來構造其他的對象，進而與PowerManagerService進行交互的。

在文章最後，在仔細分析使用power作為參數構造出的對像中對PowerManagerService的操作。接下來先看PowerManagerService的構造函數，其實在SystemServer的主線程中，和PowerManagerService初始化相關的僅有兩個方法，分別構造函數和init方法，我先來看看構造函數的代碼，內容如下

```
1     public PowerManagerService() {
 2         synchronized (mLock) {// 下面是對於PowerManagerService中一些必要的變量進行的創建
 3             mWakeLockSuspendBlocker = createSuspendBlockerLocked("PowerManagerService" );
 4             mWakeLockSuspendBlocker.acquire();// 申請持有SuspendBlocker，防止系統停止工作
 5             mScreenOnBlocker = new ScreenOnBlockerImpl();
 6             mDisplayBlanker = new DisplayBlankerImpl();
 7             mHoldingWakeLockSuspendBlocker = true ;
 8             mWakefulness = WAKEFULNESS_AWAKE;
 9         }
10
11         nativeInit();// 初始化native層的PowerManagerService，等會著重分析下這個函數
12         nativeSetPowerState( true , true );// 這這個方法是用來初始化兩個全局變量的，一個是gScreenOn，一個是gScreenBright，用來標示屏幕是否開啟，是否亮著
13     }
```

在上面的代碼中，我們可以看出構造函數其實挺簡單的，在初始化了必要的變量之後，還調用了一個native方法nativeInit()，想必是用來初始化native層的一些必要的屬性的。

我還是先看看nativeInit的代碼吧：

```
1 static void nativeInit(JNIEnv* env, jobject obj) {
 2     gPowerManagerServiceObj = env-> NewGlobalRef(obj);
```

```
3       // 在閱讀和HAL相關 的代碼是，經常見到如下這個方法，和hw_module_t這個結構體
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    }
11 }
```

在閱讀和HAL相關的代碼時，經常會見到hw_get_module這個方法，以及hw_module_t這個結構體，這次在這裡我們要仔細看看這個方法和結構體是怎麼回事。

在hw_get_module之前，我們還是先弄清楚方法中的兩個參數的含義為好。

(3) 與用戶交互的角度

通過幾個事件來分析，PowerManagerService與用戶之間的交互，

首先是按下手機Power鍵之後的事件處理流程；

其次是調節Brightness與PowerManagerService之間的關係

最後，睡眠時間和沒有用戶輸入時，手機逐漸進入睡眠狀態的變化流程。

使用PowerManagerService作為參數構造出的對像中,對PowerManagerService的操作,這些對像如下：

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);