**系统电源管理简介**

# 概述

电源管理（PowerManager）在任何设备中都是最重要的组成部分之一，良好的电源管理方案可以达到节能、延长电池寿命、降低辐射、降温等目的。

移动设备的电量主要有两种元件消耗：**CPU和显示屏**。设法降低这两种元件的耗电量就是电源管理的关键。为移动设备设计的CPU大都有两种工作频率，为了省电，大部分时间CPU都工作在较低的频率下，只有进行密集计算时，如视频解码，才会切换到高频状态。而显示屏省电的方法是尽量减少亮屏的时间，但是显示屏的开关和应用有很大的关系，因此，系统需要有一套机制来控制显示屏的开关和亮度，这是电源管理模块的主要工作之一。

## 软件架构

Android的电源管理主要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，整个系统的框架可以分为四个层次，分别是应用程序层、Framework层、硬件抽象层（HAL）和内核层。Android电源管理框架如图1-1所示。

l 应用层：这里所谓的应用层主要是指应用程序和其他使用电源管理的Service，包括但不限于以下Services: PowerManagerService、BatteryService、LightService等。

l 框架层：在Android框架层包含了对应用层接口的API调用以及电源的协调工作，主要包含PowerManager.java、PowerManagerService.java、com\_android\_server\_PowerManagerService.cpp、Power.java、android\_os\_Power.cpp。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。PowerManager.java是提供给应用层调用的。android\_os\_power.cpp是jni交互文件。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。

l HAL层：该层只有一个Power.c文件，该文件通过sysfs的方式与kernel进行通信。主要功能有申请wake\_lock，释放wake\_lock，设置屏幕状态等。用户空间的native库绝不能直接调用Android电源管理（见下图）。绕过Android运行时的电源管理政策，将破坏该系统。所有对电源管理的调用应通过Android的PowerManagerAPI来完成。

l Kernel层：内核层的电源管理方案实现主要包含三部分：

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。

系统正常开机后，Brightness的亮度会设置成用户设定的亮度，系统Screen off timer开始计时，在计时时间到之前，如果有任何的userActivity事件发生，比如Touch click等事件，则将重新设置screen off timer，系统保持在Awake状态。如果应用程序在这段时间申请了Full wakelock，系统也将保持在Awake状态。在没有交互的情况下，首先进入到Awake状态，之后进入Dozing状态，最后进入Asleep状态。

PowerManagerService负责Andorid系统中电源管理方面的工作。作为系统核心服务之一，PowerManagerService与其他服务及HAL层等都有交互关系，所以PowerManagerService相对PackageManager来说，其社会关系更复杂，分析难度也会更大一些。

先来看直接与PowerManagerService有关的类家族成员，如图5-1所示

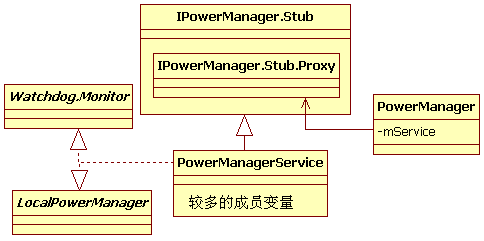


图5-1  PowerManagerService及相关类家族

由图5-1可知：

·  PowerManagerService从IPowerManager.Stub类派生，并实现了Watchdog.Monitor及LocalPowerManager接口。PowerManagerService内部定义了较多的成员变量，在后续分析中，我们会对其中比较重要的成员逐一进行介绍。

·  根据第4章介绍的知识，IPowerManager.Stub及内部类Proxy均由aidl工具处理PowerManager.aidl后得到。

·  **客户端使用PowerManager类，其内部通过代表BinderProxy端的mService成员变量与PowerManagerService进行跨Binder通信。**

现在开始PowerManagerService（以后简写为PMS）的分析之旅，先从它的调用流程入手。

**提示**PMS和BatteryService、BatteryStatsService均有交互关系，这些内容放在后面分析。

## 本章主要内容：

·  深入分析PowerManagerService

·  深入分析BatteryService和BatteryStatsService

## 本章所涉及的源代码文件名及位置：

·  PowerManagerService.java

frameworks/base/services/java/com/android/server/PowerManagerService.java

·  com\_android\_server\_PowerManagerService.cpp

frameworks/base/services/jni/com\_android\_server\_PowerManagerService.cpp

·  PowerManager.java

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

·  WorkSoure.java

frameworks/base/core/java/android/os/WorkSoure.java

·  Power.java

frameworks/base/core/java/android/os/Power.java

·  android\_os\_Power.cpp

frameworks/base/core/jni/android\_os\_Power.cpp

·  com\_android\_server\_InputManager.cpp

frameworks/base/services/jni/com\_android\_server\_InputManager.cpp

·  LightService.java

frameworks/base/services/java/com/android/server/LightService.java

·  com\_android\_server\_LightService.cpp

frameworks/base/services/jni/com\_android\_server\_LightService.cpp

·  BatteryService.java

frameworks/base/services/java/com/android/server/BatteryService.java

·  com\_android\_server\_BatteryService.cpp

frameworks/base/services/jni/com\_android\_server\_BatteryService.cpp

·  ActivityManagerService.java

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

·  BatteryStatsService.java

frameworks/base/services/java/com/android/server/am/BatteryStatsService.java

·  BatteryStatsImpl.java

frameworks/base/core/java/com/android/internal/os/BatteryStatsImpl.java

·  LocalPowerManager.java

frameworks/base/core/java/android/os/LocalPowerManager.java

## 问题背景

android 禁止电量不足自动关机

Android Low Battery 低电量处理流程

powermanagerservice

# 初识PowerManagerService

PowerManagerService是Android电源管理的核心服务，主要功能是控制系统的待机状态，控制显示屏的开关和亮度调节，以及查询和控制光线传感器和距离传感器等

PMS由SystemServer在ServerThread线程中创建。这里从中提取了4个关键调用点，如下所示：

## 启动入口

PowerManagerService也是在SystemServer中创建并加入到ServiceManager中的，代码如下：

**[java]** [view plain](http://blog.csdn.net/kitty_landon/article/details/47107045) [copy](http://blog.csdn.net/kitty_landon/article/details/47107045)

1. mPowerManagerService = mSystemServiceManager.startService(
2. PowerManagerService.**class**);

调用SystemServiceManager的startService方法创建PowerManagerService对象并注册到ServiceManager中。

## PowerManagerService构造函数

**[java]** [view plain](http://blog.csdn.net/kitty_landon/article/details/47107045) [copy](http://blog.csdn.net/kitty_landon/article/details/47107045)

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());
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**;
18. mWakefulness = WAKEFULNESS\_AWAKE;//设置PowerManagerService的状态
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：

l WAKEFULNESS\_ASLEEP：表示系统当前处于休眠状态，只能被wakeUp()调用唤醒。

l WAKEFULNESS\_AWAKE：表示系统目前处于正常运行状态。

l WAKEFULNESS\_DREAMING：表示系统当前正处于播放屏保的状态。

l 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。

## 系统准备工作—SystemReady

SystemServer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：

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. mDisplayManagerInternal =getLocalService(DisplayManagerInternal.**class**);//DisplayManagerService
7. mPolicy = getLocalService(WindowManagerPolicy.**class**);//WindowManagerPolicy
8. mBatteryManagerInternal =getLocalService(BatteryManagerInternal.**class**);//BatteryService
10. //获取最小、最大、默认屏幕亮度
11. 。。。。。。
12. //创建SensorManager对象，用于和SensorService交互
13. SensorManager sensorManager = **new** SystemSensorManager(mContext, mHandler.getLooper());
15. mBatteryStats = BatteryStatsService.getService();//获得BatteryStatsService的引用对象
16. mNotifier = **new** Notifier(Looper.getMainLooper(), mContext, mBatteryStats,
17. mAppOps,createSuspendBlockerLocked("PowerManagerService.Broadcasts"),
18. mPolicy);//创建Notifier对象
20. mWirelessChargerDetector = **new** WirelessChargerDetector(sensorManager,
21. createSuspendBlockerLocked("PowerManagerService.WirelessChargerDetector"),
22. mHandler);//创建检测无线充电的对象WirelessChargerDetector
23. mSettingsObserver = **new** SettingsObserver(mHandler);//创建监听系统设置项变化的对象
25. mLightsManager = getLocalService(LightsManager.**class**);//LightsManager对象
26. mAttentionLight =mLightsManager.getLight(LightsManager.LIGHT\_ID\_ATTENTION);
28. //初始化Power的管理模块
29. mDisplayManagerInternal.initPowerManagement(
30. mDisplayPowerCallbacks,mHandler, sensorManager);
32. 。。。。。。 //注册广播接收器
33. 。。。。。。//注册监听更多的settngs项的变化
34. // Go.
35. readConfigurationLocked();
36. updateSettingsLocked();
37. mDirty |= DIRTY\_BATTERY\_STATE;
38. updatePowerStateLocked();
39. }
40. }

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Notifer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：

**[java]** [view plain](http://blog.csdn.net/kitty_landon/article/details/47107045) [copy](http://blog.csdn.net/kitty_landon/article/details/47107045)

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：

### userActivityNoUpdateLocked

d

1. **private** **boolean** userActivityNoUpdateLocked(long eventTime,int event,int flags,int uid) {
2. **if** (eventTime< mLastSleepTime || eventTime < mLastWakeTime
3. || !mBootCompleted ||!mSystemReady) {
4. **return** **false**;
5. }
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. }
14. mNotifier.onUserActivity(event,uid);//发出通知
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
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：

### goToSleepInternal

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，

### goToSleepNoUpdateLocked

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。

### updatePowerStateLocked

## 控制系统的休眠机制acquireWakeLock

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。

### 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 (RemoteExceptionex) {

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：

### wakeUpNoUpdateLocked

private boolean wakeUpNoUpdateLocked(longeventTime,intuid) {

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：

### releaseWakeLock

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：

Override // Binder call

public void releaseWakeLock(IBinder lock, int flags) {

if (lock == null) {

throw new IllegalArgumentException("lock must not be null");

}

mContext.enforceCallingOrSelfPermission(android.Manifest.permission.WAKE\_LOCK, null);

final long ident = Binder.clearCallingIdentity();

try {

releaseWakeLockInternal(lock, flags);

} finally {

Binder.restoreCallingIdentity(ident);

}

}

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。

## 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 anduser 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。

booleandisplayBecameReady= updateDisplayPowerStateLocked(dirtyPhase2);

// Phase 3: Update dream state (depends on display readysignal).

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);

}

}

### updateIsPoweredLocked

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。

### updateStayOnLocked

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。

### updateScreenBrightnessBoostLocked

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；

### 无限for循环

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。

### 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。

### updateWakefulnessLocked()

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(intdirty) {

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 &&isItBedTimeYetLocked()) {

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isItBedTimeYetLocked()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isItBedTimeYetLocked方法什么情况下返回true。

private boolean isItBedTimeYetLocked() {

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，isItBedTimeYetLocked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(longeventTime,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(intwakefulness,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。

### updateDisplayPowerStateLocked()

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。

### updateDreamLocked()

用来启动屏保。

### updateSuspendBlockerLocked()

方法。来决定系统是休眠还是唤醒。

## 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。

### DisplayPowerController.requestPowerState

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

DisplayManagerService. requestPowerState

mDisplayPowerController.requestPowerState(request,

waitForNegativeProximity);

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chenge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updateSyspendBlockerLocked()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。

### DisplayPowerController.updatePowerState

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。

# WakeLock的native层实现

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()。在构造方法调用了nativeInit()方法

## nativeInit()

static void nativeInit(JNIEnv\* env, jobject obj) {

gPowerManagerServiceObj = env->NewGlobalRef(obj);

status\_t err = hw\_get\_module(POWER\_HARDWARE\_MODULE\_ID,

(hw\_module\_t const\*\*)&gPowerModule);

if (!err) {

gPowerModule->init(gPowerModule);

} else {

ALOGE("Couldn't load %s module (%s)", POWER\_HARDWARE\_MODULE\_ID, strerror(-err));

}

}

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 acruire()方法流程图

## acquire()

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。

## acquire\_wake\_lock

/core/jni/com\_android\_server\_PowerManagerService.cpp：

static void nativeAcquireSuspendBlocker(JNIEnv \*env, jclass clazz, jstring nameStr) {

ScopedUtfChars name(env, nameStr);

**acquire\_wake\_lock(PARTIAL\_WAKE\_LOCK, name.c\_str());**

}

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：

// hardware\libhardware\_legacy\power

const char \* const OLD\_PATHS[] = {

"/sys/android\_power/acquire\_partial\_wake\_lock",

"/sys/android\_power/release\_wake\_lock",

};

const char \* const NEW\_PATHS[] = {

"/sys/power/wake\_lock",

"/sys/power/wake\_unlock",

};

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：

static inline void

initialize\_fds(void)

{

// XXX: should be this:

//pthread\_once(&g\_initialized, open\_file\_descriptors);

// XXX: not this:

if (g\_initialized == 0) {

if(open\_file\_descriptors(NEW\_PATHS) < 0)

open\_file\_descriptors(OLD\_PATHS);

g\_initialized = 1;

}

}

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。

[-->SystemServer.java]

......//ServerThread的run函数

power =new PowerManagerService();//①创建PMS对象

ServiceManager.addService(Context.POWER\_SERVICE, power);//注册到SM中

......

//②调用PMS的init函数

power.init(context,lights, ActivityManagerService.self(), battery);

......//其他服务

power.systemReady();//③调用PMS的systemReady

......//系统启动完毕，会收到ACTION\_BOOT\_COMPLETED广播

//④PMS处理ACTION\_BOOT\_COMPLETED广播

先从第一个关键点即PMS的构造函数开始分析。

## PMS构造函数分析

# 电池管理服务BS

## 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：

源码结构

Android电池服务的源码结构

Framework\base\services\java\com\android\server

├── SystemServer.java

创建BatteryServices、PowerManagerService、ActivityManagerService

├── BatterySevices.java

监听底层上报的battery事件，广播电池发生改变的消息，提供接口用于获取电池信息，充电状态等

Framework\base\services\java\com\android\server\am

├── ActivityManagerService.java

创建BatteryStatsService

├── BatteryStatsService.java

统计和记录电池参数的信息，主要用做用电统计，通过它可知谁是系统中的耗电大户。

Framework\base\services\java\com\android\server\power

├── PowerManagerService.java

监听电池发生变化的广播消息，并调节系统的电源状态，例如亮屏

Framework\base\core\java\com\internal\os\

├── BatteryStatsImpl.java

统计和记录电池参数的信息，并通知其他模块

System\core\healthd

├── healthd.cpp

创建uevent socket，监听内核上报的内核事件

├── BatteryMonitor.cpp

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

## BatteryService分析

BatteryService由SystemServer创建，代码如下：

battery = new BatteryService(context, lights);

ServiceManager.addService("battery",battery);

### BatteryService的构造函数

public BatteryService(Context context,LightsService lights) {

mContext =context;

mLed = newLed(context, lights);//提示灯控制，感兴趣的读者可自行阅读相关代码

//BatteryService也需要和BatteryStatsService交互

mBatteryStats = BatteryStatsService.getService();

//获取一些配置参数

mCriticalBatteryLevel = mContext.getResources().getInteger(

com.android.internal.R.integer.config\_criticalBatteryWarningLevel);

mLowBatteryWarningLevel = mContext.getResources().getInteger(

com.android.internal.R.integer.config\_lowBatteryWarningLevel);

mLowBatteryCloseWarningLevel = mContext.getResources().getInteger(

com.android.internal.R.integer.config\_lowBatteryCloseWarningLevel);

//启动uevent监听对象，监视power\_supply信息

mPowerSupplyObserver.startObserving("SUBSYSTEM=power\_supply");

//如果下列文件存在，那么启动另一个uevent监听对象。该uevent事件来自invalid charger

//switch设备（即不匹配的充电设备）

if (newFile("/sys/devices/virtual/switch/invalid\_charger/state").exists()) {

mInvalidChargerObserver.startObserving(

"DEVPATH=/devices/virtual/switch/invalid\_charger");

}

update();//①查询HAL层，获取此时的电池信息

}

BatteryService定义了3个非常重要的阈值，分别是：

· **mCriticalBatteryLevel表示严重低电**，其值为4。当电量低于该值时会强制关机。该值由config.xml中的config\_criticalBatteryWarningLevel控制。

· mLowBatteryWarningLevel表示低电，值为15，当电量低于该值时，系统会报警，例如闪烁LED灯。该值由config.xml中的config\_lowBatteryWarningLevel控制。

· mLowBatteryCloseWarningLevel表示一旦电量大于此值，就脱离低电状态，即可停止警示灯。该值为20，表示由config.xml中的config\_lowBatteryCloseWarningLevel控制。

### BatteryService.java::update

private synchronized final void update() {

native\_update();//到Native层查询并更新内部变量的值

processValues();//处理更新后的状态

}

1. native\_update函数分析

native\_update的实现代码如下：

[-->com\_android\_server\_BatteryService.cpp]

static voidandroid\_server\_BatteryService\_update(JNIEnv\* env, jobject obj)

{

setBooleanField(env, obj, gPaths.acOnlinePath, gFieldIds.mAcOnline);

......//获取电池信息，并通过JNI设置到Java层对应的变量中

setIntField(env, obj, gPaths.batteryTemperaturePath,

gFieldIds.mBatteryTemperature);

constint SIZE = 128;

charbuf[SIZE];

//获取信息，以下参数并不是所有手机都支持的

if(readFromFile(gPaths.batteryStatusPath, buf, SIZE) > 0)

env->SetIntField(obj, gFieldIds.mBatteryStatus,getBatteryStatus(buf));

else

env->SetIntField(obj, gFieldIds.mBatteryStatus,

gConstants.statusUnknown);

......

}

一共有哪些电池信息呢？如表5-4所示

表5-4  Android系统中的电池信息

|  |  |  |
| --- | --- | --- |
| 变量名 | 功能 | 备注 |
| mAcOnline | 是否用外接充电器充电 | 即用交流电充电 |
| mUsbOnline | 是否用USB供电 | 即用USB供电 |
| mBatteryStatus | 电池状态 | 共有5个状态，详细内容可参考com\_android\_server\_BatteryService.cpp中BatteryManagerConstants的定义 |
| mBatteryHealth | 电池健康状态 | 共7个状态，详细内容可参考com\_android\_server\_BatteryService.cpp中BatteryManagerConstants的定义 |
| mBatteryPresent | 是否使用电池 | 有些手机在没有电池的情况下可直接利用USB/交流供电 |
| mBatteryLevel | 电池电量 |  |
| mBatteryVoltage | 电池电压 |  |
| mBatteryTemperature | 电池温度 |  |
| mBatteryTechnology | 电池制造技术 | 一般为“Li-poly”即锂电池技术 |

mBatteryStatus和mBatteryHealth均有几种不同状态，详细信息可查看getBatteryStatus和getBatteryHealth函数的实现。

上述信息均通过从/sys/class/power\_supply目录读取对应文件得到。和以往使用固定路径（可能是Android 2.2版本之前）不同的是，先读取power\_supply目录中各个子目录中的type文件，然后根据type文件的内容，再做对应处理：

·  如果type文件的内容为“Mains”：则读取对应子目录中的online文件，可判断是否为AC充电。

·  如果type文件的内容为“Battery”：则从对应子目录中其他的文件中读取电池相关的信息，例如从temp文件获取电池温度，从technology文件读取电池制造技术等。

·  如果type文件的内容为“USB”：读取该子目录中的online文件内容，可判断是否为USB充电。

### BatteryService.java::processValues

获取了电池信息后，BatteryService就要做一些处理，此项工作通过processValues完成，其代码如下：

[-->BatteryService.java::processValues]

private void processValues() {

longdischargeDuration = 0;

mBatteryLevelCritical = mBatteryLevel <= mCriticalBatteryLevel;

if (mAcOnline) {

mPlugType = BatteryManager.BATTERY\_PLUGGED\_AC;

} elseif (mUsbOnline) {

mPlugType = BatteryManager.BATTERY\_PLUGGED\_USB;

} else {

mPlugType = BATTERY\_PLUGGED\_NONE;

}

//通知BatteryStatsService，该函数以后再分析

mBatteryStats.setBatteryState(mBatteryStatus, mBatteryHealth,

mPlugType, mBatteryLevel, mBatteryTemperature, mBatteryVoltage

);

shutdownIfNoPower();//如果电量不够，弹出关机对话框

shutdownIfOverTemp();//如果电池过热，弹出关机对话框

......//根据当前电池信息与上次电池信息比较，判断是否需要发送广播等

if (比较前后两次电池信息是否发生变化) {

......//记录信息到日志文件

Intent statusIntent = new Intent();

statusIntent.setFlags(

Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY\_BEFORE\_BOOT);

if (mPlugType != 0 && mLastPlugType ==0) {

statusIntent.setAction(Intent.ACTION\_POWER\_CONNECTED);

mContext.sendBroadcast(statusIntent);

}......

if(sendBatteryLow) {

mSentLowBatteryBroadcast = true;//发送低电提醒

statusIntent.setAction(Intent.ACTION\_BATTERY\_LOW);

mContext.sendBroadcast(statusIntent);

} ......

mLed.updateLightsLocked();//更新LED灯状态

mLastBatteryStatus= mBatteryStatus;//保存新的电池信息

......

}

processValues函数非常简单，此处不再详述。另外，当电池信息发生改变时，系统会发送uevent事件给BatteryService，此时BatteryService只要重新调用update即可完成工作。

## BatteryStatsService/BSS

atteryStatsService（为书写方便，以后简称BSS）主要功能是收集系统中各模块和应用进程用电量情况。抽象地说，BSS就是一块电表，不过这块电表不只是显示总的耗电量，而是分门别类地显示耗电量，力图做到更为精准。

和其他服务不太一样的是，BSS的创建和注册是在ActivityManagerService中进行的，相关代码如下：

[-->ActivityManagerService.java::ActivityManagerService构造函数]

private ActivityManagerService() {

......//创建BSS对象，传递一个File对象，指向/data/system/batterystats.bin

mBatteryStatsService= new BatteryStatsService(new File(

systemDir, "batterystats.bin").toString());

}

[-->ActivityManagerService.java::main]

//调用BSS的publish函数，在内部将其注册到ServiceManager

m.mBatteryStatsService.publish(context);

下面来分析BSS的构造函数，见识一下这块电表的样子。

### 构造函数

BSS其实只是一个壳，具体功能委托BatteryStatsImpl（以后简称BStatsImpl）来实现，代码如下

BatteryStatsService(String filename) {

mStats = new BatteryStatsImpl(filename);

//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");

}

}

图5-2展示了BSS及BStatsImpl的家族图谱。

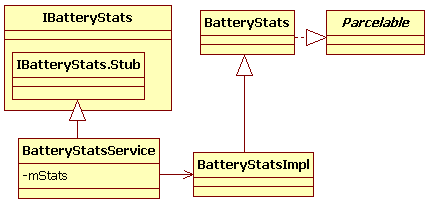


图5-2 BSS及BStatsImpl家族图谱

由图5-2可知：

· BSS通过成员变量mStats指向一个BStatsImpl类型的对象。

· BStatsImpl从BatteryStats类派生。更重要的是，该类实现了Parcelable接口，由此可知，BStatsImpl对象的信息可以写到Parcel包中，从而可通过Binder在进程间传递。实际上，在Android手机的设置中查到的用电信息就是来自BStatsImpl的。

BSS的getStatistics函数提供了查询系统用电信息的接口，代码如下：

public byte[] getStatistics() {

mContext.enforceCallingPermission(//检查调用进程是否有BATTERY\_STATS权限

android.Manifest.permission.BATTERY\_STATS, null);

Parcel out= Parcel.obtain();

mStats.writeToParcel(out, 0);//将BStatsImpl信息写到数据包中

byte[]data = out.marshall();//序列化为一个buffer，然后通过Binder传递

out.recycle();

returndata;

}

由此可以看出，电量统计的核心类是BStatsImpl，下面就来分析它。

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，事件发生时会调用该对象的onUEvent设备文件方法，如下：

privatefinal UEventObservermInvalidChargerObserver =newUEventObserver() {

@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。

publicint getInvalidCharger(){

synchronized (mLock) {

return mInvalidCharger;

}

}

### onStart

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：

privatefinal class BatteryListenerextendsIBatteryPropertiesListener.Stub {

@Override

public void batteryPropertiesChanged(BatteryPropertiesprops) {

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：

publicclass BatteryPropertiesimplementsParcelable {

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

}

### BStatsImpl流程分析

#### BStatsImpl

BStatsImpl功能是进行电量统计，那么是否存在计量工具呢？答案是肯定的，并且BStatsImpl使用了不止一种的计量工具。

#### 计量工具和统计对象介绍

BStatsImpl一共使用了4种计量工具，如图5-3所示。

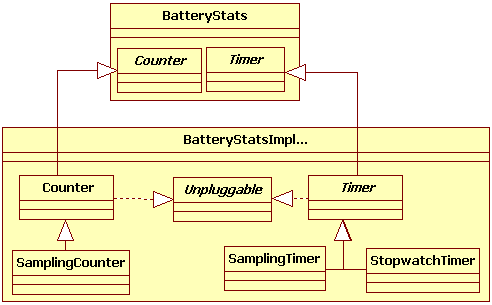


图5-3 计量工具图例

由图5-3可知：

· 一共有两大类计量工具，Counter用于计数，Timer用于计时。

· BStatsImpl实现了StopwatchTimer（即所谓的秒表）、SamplingTimer（抽样计时）、Counter和SamplingCounter（抽样计数）等4个具体的计量工具。

· BStatsImpl中定义了一个Unpluggable接口。当手机插上USB线充电（不论是由AC还是由USB供电）时，该接口的plug函数被调用。反之，当拔去USB线时，该接口的unplug函数被调用。设置这个接口的目的是为了满足BStatsImpl对各种情况下系统用电量的统计要求。关于Unpluggable接口的作用，在后续内容中可以能见到。

虽然只有4种计量工具（笔者觉得已经相当多了），但是可以在很多地方使用它们。下面先来认识部分被挂牌要求统计用电量的对象，如表5-5所示。

|  |  |  |
| --- | --- | --- |
| 成员变量名 | 类型 | 备注 |
| mScreenOnTimer | StopwatchTimer | 统计屏幕开启耗电量 |
| mScreenBrightnessTimer[] | StopwatchTimer | 统计各级屏幕亮度（共5级）情况下的耗电量 |
| mInputEventCounter | Counter | 统计输入事件耗电量 |
| mPhoneOnTimer | StopwatchTimer | 统计通话耗电量 |
| mPhoneSignalStrengthsTimer[] | StopwatchTimer | 统计手机信号各级强度耗电量，共5级 |
| mPhoneSignalScanningTimer | StopwatchTimer | 统计搜索手机信号耗电量 |
| mPhoneDataConnectionsTimer[] | StopwatchTimer | 统计手机使用各种数据通信方式（如GPRS、CDMA等）的用电量，一共15级 |
| mWifiOnTimer | StopwatchTimer | Wifi用电量（包括使用网络和开启Wifi功能却没有使用网络的情况） |
| mGlobalWifiRunningTimer | StopwatchTimer | 使用Wifi的用电量 |
| mAudioOnTimer | StopwatchTimer | 使用Audio的耗电量 |
| mVideoOnTimer | StopwatchTimer | 使用Video的耗电量 |

表5-5中的电量统计项已经够多了吧？还不止这些，为了做到更精确，Android还希望能统计每个进程在各种情况下的耗电量。这是一项庞大的工程，怎么做到的呢？来看下一节的内容

#### BatteryStats.Uid介绍

在Android 4.0中，和进程相关的用电量统计并非以单个PID为划分单元，而是以Uid为组，相关类结构如图5-4所示。

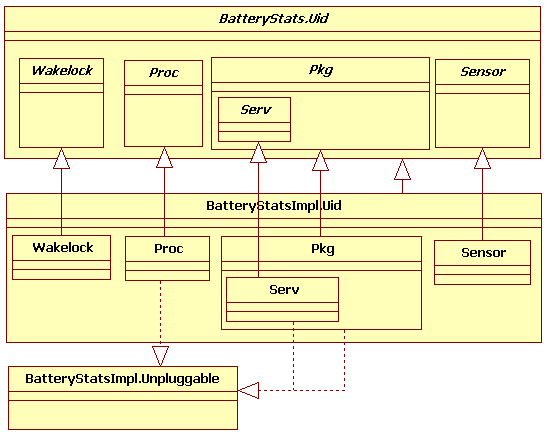


图5-4  BatteryStats.Uid家族

由图5-4可知：

· Wakelock用于统计该Uid对应进程使用wakeLock的情况。

· Proc用于统计Uid中某个进程的电量使用情况。

· Pkg用于统计某个特定Package的使用情况，其内部类Serv用于统计该Pkg中Service的用电情况。

· Sensor用于统计传感器用电情况。

基于以上的了解，以后分析将会轻松很多，下面来分析它的代码。

#### 构造函数分析

先分析构造函数，代码如下：

[-->BatteryStatsImpl.java::BatteryStatsImpl构造函数]

public BatteryStatsImpl(String filename) {

//JournaledFile为日志文件对象，内部包含两个文件，原始文件和临时文件。目的是双备份，

//以防止在读写过程中文件信息丢失或出错

mFile =new JournaledFile(new File(filename), new File(filename + ".tmp"));

mHandler= new MyHandler();//创建一个Handler对象

mStartCount++;

//创建表5-5中的用电统计项对象

mScreenOnTimer = new StopwatchTimer(null, -1, null, mUnpluggables);

for (inti=0; i<NUM\_SCREEN\_BRIGHTNESS\_BINS; i++) {

mScreenBrightnessTimer[i] = new StopwatchTimer(null, -100-i, null,

mUnpluggables);

}

mInputEventCounter = new Counter(mUnpluggables);

......

mOnBattery= mOnBatteryInternal = false;//设置这两位成员变量为false

initTimes();//①初始化统计时间

mTrackBatteryPastUptime = 0;

mTrackBatteryPastRealtime = 0;

mUptimeStart= mTrackBatteryUptimeStart =

SystemClock.uptimeMillis()\* 1000;

mRealtimeStart= mTrackBatteryRealtimeStart =

SystemClock.elapsedRealtime()\* 1000;

mUnpluggedBatteryUptime = getBatteryUptimeLocked(mUptimeStart);

mUnpluggedBatteryRealtime = getBatteryRealtimeLocked(mRealtimeStart);

mDischargeStartLevel = 0;

mDischargeUnplugLevel = 0;

mDischargeCurrentLevel = 0;

initDischarge(); //②初始化和电池level有关的成员变量

clearHistoryLocked();//③删除用电统计的历史记录

}

要看懂这段代码比较困难，主要原因是变量太多，并且没有注释说明。只能根据名字来推测了。在以上代码中除了计量工具外，还出现了三大类变量：

用于统计时间的变量，例如mUptimeStart、mTrackBatteryPastUptime等。这些参数的初始化函数为initTimes。注意，系统时间分为uptime和realtime。uptime和realtime的时间起点都从系统启动开始算（since the system was booted），但是uptime不包括系统休眠时间，而realtime包括系统休眠时间[③]。

· 用于记录各种情况下电池电量的变量，如mDischargeStartLevel、mDischargeCurrentLevel等，这些成员变量的初始化函数为initDischarge。

· 用于保存历史记录的HistroryItem，在clearHistoryLocked函数中初始化，主要有mHistory、mHistoryEnd等成员变量（这些成员在clearHistoryLocked函数中出现）。

上述这些成员变量的具体作用，只有通过后文的分析才能弄清楚。这里先介绍StopwacherTimer。

//调用方式

mPhoneSignalScanningTimer = newStopwatchTimer(null, -200+1,

null,mUnpluggables);

//mUnpluggables类型为ArrayList<Unpluggable>，用于保存插拔USB线时需要对应更新用电

//信息的统计对象

// StopwatchTimer的构造函数

StopwatchTimer(Uid uid, int type,ArrayList<StopwatchTimer> timerPool,

ArrayList<Unpluggable>unpluggables) {

//在本例中，uid为0，type为负数，timerPool为空，unpluggables为mUnpluggables

super(type, unpluggables);

mUid =uid;

mTimerPool = timerPool;

}

// Timer的构造函数

Timer(int type, ArrayList<Unpluggable>unpluggables) {

mType =type;

mUnpluggables = unpluggables;

unpluggables.add(this);

}

在StopwatchTimer中比较难理解的就是unpluggables，根据注释说明，当拔插USB线时，需要更新用电统计的对象，应该将其加入到mUnpluggables数组中。

在启动秒表时，调用它的startRunningLocked函数，并传入BStatsImpl实例，代码如下：

void startRunningLocked(BatteryStatsImpl stats) {

if(mNesting++ == 0) {//嵌套调用控制

// getBatteryRealtimeLocked函数返回总的电池使用时间

mUpdateTime = stats.getBatteryRealtimeLocked(

SystemClock.elapsedRealtime()\* 1000);

if (mTimerPool != null) {//不讨论这种情况

}

mCount++;

mAcquireTime = mTotalTime;//计数控制，请读者阅读相关注释说明

}

}

当停用秒表时，调用它的stopRunningLocked函数，代码如下：

void stopRunningLocked(BatteryStatsImpl stats) {

if (mNesting == 0) {

return; //嵌套控制

}

if(--mNesting == 0) {

if(mTimerPool != null) {//不讨论这种情况

}else {

final long realtime = SystemClock.elapsedRealtime() \* 1000;

//计算此次启动/停止周期的时间

final long batteryRealtime = stats.getBatteryRealtimeLocked(realtime);

mNesting = 1;

//mTotalTime代表从启动开始该秒停表一共记录的时间

mTotalTime = computeRunTimeLocked(batteryRealtime);

mNesting = 0;

}

if (mTotalTime == mAcquireTime) mCount--;

}

}

在StopwatchTimer中定义了很多的时间参数，无非就是用于记录各种时间，例如总耗时、最近一次工作周期的耗时等。如果不是工作需要（例如研究Settings应用中和BatteryInfo相关的内容），读者仅需了解它的作用即可。

### ActivityManagerService和BSS交互

ActivityManagerService创建BSS后，还要进行几项操作，具体代码分别如下：

[-->ActivityManagerService.java::ActivityManagerService构造函数]

mBatteryStatsService = new BatteryStatsService(newFile(

systemDir, "batterystats.bin").toString());

//操作通过BStatsImpl创建的JournaledFile文件

mBatteryStatsService.getActiveStatistics().readLocked();

mBatteryStatsService.getActiveStatistics().writeAsyncLocked();

//BStatsImpl的getIsOnBattery返回mOnBattery变量，初始化值为false

mOnBattery= DEBUG\_POWER ? true

: mBatteryStatsService.getActiveStatistics().getIsOnBattery();

//设置回调，该回调也是用于信息统计，只能留到介绍ActivityManagerService时再来分析了

mBatteryStatsService.getActiveStatistics().setCallback(this);

[-->ActivityManagerService.java::main函数]

m.mBatteryStatsService.publish(context);

[-->BatteryStatsService.java::publish]

public void publish(Context context) {

mContext =context;

//注意，BSS服务叫做batteryinfo，而BatteryService服务叫做battery

ServiceManager.addService("batteryinfo", asBinder());

//PowerProfile见下文解释

mStats.setNumSpeedSteps(new PowerProfile(mContext).getNumSpeedSteps());

//设置通信信号扫描超时时间

mStats.setRadioScanningTimeout(mContext.getResources().getInteger(

com.android.internal.R.integer.config\_radioScanningTimeout)

\* 1000L);

}

在以上代码中，比较有意思的是PowerProfile类，它将解析Android 4.0源码/frameworks/base/core/res/res/xml/power\_profile.xml文件。此XML文件存储的是各种操作（和硬件相关）的耗电情况，如图5-5所示。

<https://android.googlesource.com/platform/frameworks/base/+/master/core/res/res/xml/power_profile.xml>

|  |  |
| --- | --- |
|  | <item name="screen.on">0.1</item> <!-- ~200mA -->  <item name="screen.full">0.1</item> <!-- ~300mA --> |
|  | <item name="bluetooth.active">0.1</item> <!-- Bluetooth data transfer, ~10mA --> |
|  | <item name="bluetooth.on">0.1</item> <!-- Bluetooth on & connectable, but not connected, ~0.1mA --> |
|  | <item name="wifi.on">0.1</item> <!-- ~3mA --> |
|  | <item name="wifi.active">0.1</item> <!-- WIFI data transfer, ~200mA --> |
|  | <item name="wifi.scan">0.1</item> <!-- WIFI network scanning, ~100mA --> |
|  | <item name="dsp.audio">0.1</item> <!-- ~10mA --> |
|  | <item name="dsp.video">0.1</item> <!-- ~50mA --> |
|  | <item name="camera.flashlight">0.1</item> <!-- Avg. power for camera flash, ~160mA --> |
|  | <item name="camera.avg">0.1</item> <!-- Avg. power use of camera in standard usecases, ~550mA --> |
|  | <item name="gps.on">0.1</item> <!-- ~50mA --> |

由图5-5可知，该文件保存了各种操作的耗电情况，以mAh（毫安）为单位。PowerProfile的getNumSpeedSteps将返回CPU支持的频率值，目前在该XML中只定义了一个值，即400MHz。

注意在编译时，各厂家会将特定硬件平台的power\_profile.xml复制到输出目录。此处展示的power\_profile.xml和硬件平台无关。通常在

device/rockchip/common/overlay/framworks/base/core/res/xml/power\_profile.xml

### BatteryService和BSS交互

#### BatteryService在它的processValues

BatteryService在它的processValues函数中和BSS交互，代码如下：

[-->BatteryService.java]

private void processValues() {

   ......

   mBatteryStats.setBatteryState(mBatteryStatus,mBatteryHealth, mPlugType,

                 mBatteryLevel, mBatteryTemperature,mBatteryVoltage);

}

BSS的工作由BStatsImpl来完成，所以直接setBatteryState函数的代码：

[-->BatteryStatsImpl.java::setBatteryState]

public void setBatteryState(int status, inthealth, int plugType, int level,

                                int temp, int volt) {

  synchronized(this) {

      boolean onBattery = plugType == BATTERY\_PLUGGED\_NONE;//判断是否为电池供电

       intoldStatus = mHistoryCur.batteryStatus;

       ......

        if(onBattery) {

            //mDischargeCurrentLevel记录当前使用电池供电时的电池电量

            mDischargeCurrentLevel = level;

            mRecordingHistory = true;//mRecordingHistory表示需要记录一次历史值

         }

       //此时,onBattery为当前状态，mOnBattery为历史状态

      if(onBattery != mOnBattery) {

          mHistoryCur.batteryLevel = (byte)level;

          mHistoryCur.batteryStatus = (byte)status;

           mHistoryCur.batteryHealth = (byte)health;

           ......//更新mHistoryCur中的电池信息

               setOnBatteryLocked(onBattery, oldStatus, level);

           } else {

               boolean changed = false;

               if (mHistoryCur.batteryLevel != level) {

                   mHistoryCur.batteryLevel = (byte)level;

                   changed = true;

               }

               ......//判断电池信息是否发生变化

               if (changed) {//如果发生变化，则需要增加一次历史记录

                   addHistoryRecordLocked(SystemClock.elapsedRealtime());

               }

           }

           if (!onBattery && status == BatteryManager.BATTERY\_STATUS\_FULL){

               mRecordingHistory = false;

           }

        }

    }

setBatteryState函数的工作主要有两项：

·  判断当前供电状态是否发生变化，由onBattery和mOnBattery进行比较。其中onBattery用于判断当前是否为电池供电，mOnBattery为上次调用该函数时得到的判断值。如果供电状态发生变化（其实就是经历一次USB拔插过程），则调用setOnBatteryLocked函数。

·  如果供电状态未发生变化，则需要判断电池信息是否发生变化，例如电量和电压等。如果发生变化，则调用addHistoryRecordLocked。该函数用于记录一次历史信息。

接下来看setOnBatteryLocked函数的代码：

#### BatteryStatsImpl.java::setOnBatteryLocked

void setOnBatteryLocked(boolean onBattery, intoldStatus, int level) {

boolean doWrite = false;

//发送一个消息给mHandler，将在内部调用ActivityManagerService设置的回调函数

Message m= mHandler.obtainMessage(MSG\_REPORT\_POWER\_CHANGE);

m.arg1 =onBattery ? 1 : 0;

mHandler.sendMessage(m);

mOnBattery = mOnBatteryInternal = onBattery;

longuptime = SystemClock.uptimeMillis() \* 1000;

longmSecRealtime = SystemClock.elapsedRealtime();

longrealtime = mSecRealtime \* 1000;

if(onBattery) {

//关于电量信息统计，有一个值得注意的地方：当oldStatus为满电状态，或当前电量

//大于90，或mDischargeCurrentLevel小于20并且当前电量大于80时，要清空统计

//信息，以开始新的统计。也就是说在满足特定条件的情况下，电量使用统计信息会清零并重

//新开始。读者不妨用自己手机一试

if(oldStatus == BatteryManager.BATTERY\_STATUS\_FULL || level >= 90

|| (mDischargeCurrentLevel < 20 && level >= 80)) {

doWrite = true;

resetAllStatsLocked();

mDischargeStartLevel = level;

}

//读取/proc/wakelock文件，该文件反映了系统wakelock的使用状态，

//感兴趣的读者可自行研究

updateKernelWakelocksLocked();

mHistoryCur.batteryLevel = (byte)level;

mHistoryCur.states &= ~HistoryItem.STATE\_BATTERY\_PLUGGED\_FLAG;

//添加一条历史记录

addHistoryRecordLocked(mSecRealtime);

//mTrackBatteryUptimeStart表示使用电池的开始时间，由uptime表示

mTrackBatteryUptimeStart = uptime;

// mTrackBatteryRealtimeStart表示使用电池的开始时间，由realtime表示

mTrackBatteryRealtimeStart = realtime;

//mUnpluggedBatteryUptime记录总的电池使用时间（不论中间插拔多少次）

mUnpluggedBatteryUptime = getBatteryUptimeLocked(uptime);

// mUnpluggedBatteryRealtime记录总的电池使用时间

mUnpluggedBatteryRealtime = getBatteryRealtimeLocked(realtime);

//记录电量

mDischargeCurrentLevel =mDischargeUnplugLevel = level;

if(mScreenOn) {

mDischargeScreenOnUnplugLevel = level;

mDischargeScreenOffUnplugLevel = 0;

}else {

mDischargeScreenOnUnplugLevel = 0;

mDischargeScreenOffUnplugLevel = level;

}

mDischargeAmountScreenOn = 0;

mDischargeAmountScreenOff = 0;

//调用doUnplugLocked函数

doUnplugLocked(mUnpluggedBatteryUptime, mUnpluggedBatteryRealtime);

}else {

......//处理使用USB充电的情况，请读者在上面讨论的基础上自行分析

}

......//记录信息到文件

}

}

doUnplugLocked函数将更新对应信息，该函数比较简单，无须赘述。另外，addHistoryRecordLocked函数用于增加一条历史记录（由HistoryItem表示），读者也可自行研究。

从本节的分析可知，Android将电量统计分得非常细，例如由电池供电的情况需要统计，由USB/AC充电的情况也要统计，因此有setBatteryState函数的存在。

### PowerManagerService和BSS交互

PMS和BSS交互是最多的，此处以noteScreenOn和noteUserActivity为例，来介绍BSS到底是如何统计电量的。

先来看noteScreenOn函数。当开启屏幕时，PMS会调用BSS的noteScreenOn以通知屏幕开启，该函数在内部调用BStatsImpl的noteScreenOnLocked，其代码如下：

#### BatteryStatsImpl.java::noteScreenOnLocked

public void noteScreenOnLocked() {

if(!mScreenOn) {

mHistoryCur.states |= HistoryItem.STATE\_SCREEN\_ON\_FLAG;

//增加一条历史记录

addHistoryRecordLocked(SystemClock.elapsedRealtime());

mScreenOn = true;

//启动mScreenOnTime秒停表，内部就是记录时间，读者可自行研究

mScreenOnTimer.startRunningLocked(this);

if(mScreenBrightnessBin >= 0)//启动对应屏幕亮度的秒停表（参考表5-5）

mScreenBrightnessTimer[mScreenBrightnessBin].startRunningLocked(this);

//屏幕开启也和内核WakeLock有关，所以这里一样要更新WakeLock的用电统计

noteStartWakeLocked(-1, -1, "dummy", WAKE\_TYPE\_PARTIAL);

if(mOnBatteryInternal)

updateDischargeScreenLevelsLocked(false, true);

}

}

再来看noteUserActivity，当有输入事件触发PMS的userActivity时，该函数被调用,代码如下,：

#### BatteryStatsImpl.java::noteUserActivityLocked

//BSS的noteUserActivity将调用BStatsImpl的noteUserActivityLocked

public void noteUserActivityLocked(int uid, intevent) {

getUidStatsLocked(uid).noteUserActivityLocked(event);

}

先是调用getUidStatsLocked以获取一个Uid对象，如果该Uid是首次出现的，则要在内部创建一个Uid对象。直接来了解Uid的noteUserActivityLocked函数：

public void noteUserActivityLocked(int type) {

if(mUserActivityCounters == null) {

initUserActivityLocked();

}

if (type< 0) type = 0;

else if(type >= NUM\_USER\_ACTIVITY\_TYPES)

type= NUM\_USER\_ACTIVITY\_TYPES-1;

// noteUserActivityLocked只是调用对应type的Counter的stepAtomic函数

//每个Counter内部都有个计数器，stepAtomic使该计数器增1

mUserActivityCounters[type].stepAtomic();

}

mUserActivityCounters为一个7元Counter数组，该数组对应7种不同的输入事件类型，在代码中，由BStatsImpl的成员变量USER\_ACTIVITY\_TYPES表示，如下所示：

static final String[] USER\_ACTIVITY\_TYPES = {

"other", "cheek", "touch","long\_touch", "touch\_up", "button", "unknown"

};

~~另外，在LocalPowerManager中，也定义了相关的type值，如下所示：~~

~~[-->LocalPowerManager.java]~~

~~public interface LocalPowerManager {~~

~~publicstatic final int OTHER\_EVENT = 0;~~

~~publicstatic final int BUTTON\_EVENT = 1;~~

~~publicstatic final int TOUCH\_EVENT = 2; //目前只使用这三种事件~~

~~......~~

~~}~~

# 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分组，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：

healthd

## 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。

## BatteryMonitor类-读取电池的各种参数

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。

# BatteryService及BatteryStatsService总结

本节重点讨论了BatteryService和BatteryStatsService。其中，BatteryService和系统中的供电系统交互，通过它可获取电池状态等信息。而BatteryStatsService用于统计系统用电量的情况。就难度而言，BSS较为复杂，原因是Android试图对系统耗电量作非常细致的统计，导致统计项非常繁杂。另外，电量统计大多采用被动通知的方式（即需要其他服务主动调用BSS提供的noteXXXOn/noteXXXOff函数），这种实现方法一方面加重了其他服务的负担，另一方面影响了这些服务未来的功能扩展。

**注意**虽然Google费尽心血来完善电量统计，但这并不是解决耗电量大的根本途径。另外，读者可分析Settings程序中电量统计图的绘制以加深对各种统计对象的理解。Settings中和电量相关的文件在Android 4.0源码的/packages/apps/Settings/src/com/android/settings/fuelgauge/目录中。

## 5.6  本章学习指导

本章的难度其实在BSS中，而PMS和BatteryService相对较简单。在这三项服务中， PMS是核心。读者在研究PMS时，要注意把握以下几个方面：

·  PMS的初期工作流程，即构造函数、init函数、systemReady函数和BootCompleted函数等。

·  PMS功能在于根据当前系统状态（包括mUserState和mWakeLockState）去操作屏幕和灯光。而触发状态改变的有WakeLock的获取和释放，userActivity函数的调用，因此读者也要搞清楚PMS在这两个方面的工作原理。

·  PMS还有一部分功能和传感器有关，其功能无非还是根据状态操作屏幕和灯光。除非工作需要，否则只需要简单了解这部分的工作流程即可。

对BSS来说，复杂之处在于它定义了很多成员变量和数据类型，并且没有一份电量统计标准的说明文档，因此笔者认为，读者只要搞清楚那几个计量工具和各个统计项的作用即可，如果在其他服务的代码中看到和BSS交互的函数，那么只需知道原因和目的即可。

另外，电源管理需要HAL层和Linux内核提供支持，感兴趣的读者不妨以本章知识为切入点，对底层技术进行一番深入剖析。

## 5.7  小结

电源管理系统的核心是PowerManagerService，还包括BatteryService和BatteryStatsService。本章对Android平台中的电源管理系统进行了较详细的分析，其中：

·  对于PMS，本章分析了它的初始化流程、WakeLock获取流程、userActivity函数的工作流程及Power按键处理流程。

·  BatteryService功能较为简单，读者大概了解即可。

·  对于BatteryStatsService，本章对它内部的数据结构、统计对象等进行了较详细的介绍，并对其工作流程展开了分析。建议读者结合Settings应用中的相关代码，加深对其中各种计量工具及统计对象的理解。

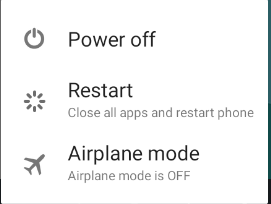
[①] config.xml文件的全路径是4.0源码/frameworks/base/core/res/res/values/config.xml。

[②]必须在一定时间内完成按下和松开Power键的操作，否则系统会认为是关机操作。详情将在卷Ⅲ输入系统一章的分析。

[③]读者可阅读SDK文档中关于SystemClock类的说明。

# 关机详解

## 长按关机键，弹关机对话框



选择Power Off则开启关机。

### 源码

长按关机键(KEYCODE\_POWER)，PhoneWindowManager.java拦截该事件，做出处理

|  |
| --- |
| /\*\* {@inheritDoc} \*/  @Override  public int interceptKeyBeforeQueueing(KeyEvent event, int policyFlags) {  if (!mSystemBooted) {  // If we have not yet booted, don't let key events do anything.  return 0;  }  final boolean interactive = (policyFlags & FLAG\_INTERACTIVE) != 0;  final boolean down = event.getAction() == KeyEvent.ACTION\_DOWN;  final boolean canceled = event.isCanceled();  final int keyCode = event.getKeyCode();  switch(keycode){  case KeyEvent.KEYCODE\_POWER: {  result &= ~ACTION\_PASS\_TO\_USER;  isWakeKey = false; // wake-up will be handled separately  if (down) {  interceptPowerKeyDown(event, interactive);  } else {  interceptPowerKeyUp(event, interactive, canceled);  }  break;  }  case KeyEvent.KEYCODE\_SLEEP: {  result &= ~ACTION\_PASS\_TO\_USER;  isWakeKey = false;  if (!mPowerManager.isInteractive()) {  useHapticFeedback = false; // suppress feedback if already non-interactive  }  if (down) {  sleepPress(event.getEventTime());  } else {  sleepRelease(event.getEventTime());  }  break;  }  case KeyEvent.KEYCODE\_WAKEUP: {  result &= ~ACTION\_PASS\_TO\_USER;  isWakeKey = true;  break;  }  }  } |

down事件：

|  |
| --- |
| private void interceptPowerKeyDown(KeyEvent event, boolean interactive) {  //如果电源键尚未处理，则检测是短按，长按，或多按操作中的哪一种，再决定做下一步处理。  mPowerKeyHandled = hungUp || mScreenshotChordVolumeDownKeyTriggered  || mScreenshotChordVolumeUpKeyTriggered;  if (!mPowerKeyHandled) {  //手机处于激活(亮屏)状态  if (interactive) {  // When interactive, we're already awake.  // Wait for a long press or for the button to be released to decide what to do.  //检查是否是长按操作  //hasLongPressOnPowerBehavior=getResolvedLongPressOnPowerBehavior() != LONG\_PRESS\_POWER\_NOTHING;  if (hasLongPressOnPowerBehavior()) {//true  Message msg = mHandler.obtainMessage(MSG\_POWER\_LONG\_PRESS);  msg.setAsynchronous(true);  //发送消息通知主线程处理长按操作  mHandler.sendMessageDelayed(msg,  ViewConfiguration.get(mContext).getDeviceGlobalActionKeyTimeout());  }  }  //手机待机状态  else {  //唤醒手机  wakeUpFromPowerKey(event.getDownTime());  if (mSupportLongPressPowerWhenNonInteractive && hasLongPressOnPowerBehavior()) {  Message msg = mHandler.obtainMessage(MSG\_POWER\_LONG\_PRESS);  msg.setAsynchronous(true);  mHandler.sendMessageDelayed(msg,  ViewConfiguration.get(mContext).getDeviceGlobalActionKeyTimeout());  mBeganFromNonInteractive = true;  } else {  final int maxCount = getMaxMultiPressPowerCount();  if (maxCount <= 1) {  mPowerKeyHandled = true;  } else {  mBeganFromNonInteractive = true;  }  }  }  }  } |

主线程接收到长按的消息并做出下一步处理

|  |
| --- |
| private class PolicyHandler extends Handler {  @Override  public void handleMessage(Message msg) {  switch (msg.what) {  case MSG\_POWER\_LONG\_PRESS:  //调用powerPress()函数  powerLongPress();  break;  }  }  } |

那接着来看powerLongPress()函数是如何处理长按事件的

|  |
| --- |
| private void powerLongPress() {  final int behavior = getResolvedLongPressOnPowerBehavior();  switch (behavior) {  case LONG\_PRESS\_POWER\_NOTHING:  break;  case LONG\_PRESS\_POWER\_GLOBAL\_ACTIONS:  mPowerKeyHandled = true;  if (!performHapticFeedbackLw(null, HapticFeedbackConstants.LONG\_PRESS, false)) {  performAuditoryFeedbackForAccessibilityIfNeed();  }  showGlobalActionsInternal();  break;  case LONG\_PRESS\_POWER\_SHUT\_OFF:  case LONG\_PRESS\_POWER\_SHUT\_OFF\_NO\_CONFIRM:  mPowerKeyHandled = true;  performHapticFeedbackLw(null, HapticFeedbackConstants.LONG\_PRESS, false);  sendCloseSystemWindows(SYSTEM\_DIALOG\_REASON\_GLOBAL\_ACTIONS);  mWindowManagerFuncs.shutdown(behavior == LONG\_PRESS\_POWER\_SHUT\_OFF);  break;  }  } |

这里根据长按的行为behavior来分情况处理的。

情况一：LONG\_PRESS\_POWER\_NOTHING：不处理

情况二：LONG\_PRESS\_POWER\_GLOBAL\_ACTIONS:正常关机流程

|  |
| --- |
| void showGlobalActionsInternal() {  //请求ActivityManagerNative关闭系统所有窗口  sendCloseSystemWindows(SYSTEM\_DIALOG\_REASON\_GLOBAL\_ACTIONS);  if (mGlobalActions == null) {  //初始化GlobalActions  mGlobalActions = new GlobalActions(mContext, mWindowManagerFuncs);  }  final boolean keyguardShowing = isKeyguardShowingAndNotOccluded();  //显示关机对话框  mGlobalActions.showDialog(keyguardShowing, isDeviceProvisioned());  if (keyguardShowing) {  // since it took two seconds of long press to bring this up,  // poke the wake lock so they have some time to see the dialog.  mPowerManager.userActivity(SystemClock.uptimeMillis(), false);  }  } |

#### sendCloseSystemWindows()

(1)sendCloseSystemWindows()；//发送请求关闭系统的对话框

void sendCloseSystemWindows(String reason) {

PhoneWindow.sendCloseSystemWindows(mContext, reason);

}

* 1
* 2
* 3

调用的是PhoneWindow.java的方法，如下：

public static void sendCloseSystemWindows(Context context, String reason) {

if (ActivityManagerNative.isSystemReady()) {

try {

ActivityManagerNative.getDefault().closeSystemDialogs(reason);

} catch (RemoteException e) {

}

}

}

继续调用ActivityManagerNative.java的方法，如下：

public void closeSystemDialogs(String reason) throws RemoteException {

Parcel data = Parcel.obtain();

Parcel reply = Parcel.obtain();

data.writeInterfaceToken(IActivityManager.descriptor);

data.writeString(reason);

mRemote.transact(CLOSE\_SYSTEM\_DIALOGS\_TRANSACTION, data, reply, 0);

reply.readException();

data.recycle();

reply.recycle();

}

mRemote.transact(CLOSE\_SYSTEM\_DIALOGS\_TRANSACTION, data, reply, 0)

@Override

public boolean onTransact(int code, Parcel data, Parcel reply, int flags)

throws RemoteException {

switch (code) {

case CLOSE\_SYSTEM\_DIALOGS\_TRANSACTION: {

data.enforceInterface(IActivityManager.descriptor);

String reason = data.readString();

closeSystemDialogs(reason);

reply.writeNoException();

return true;

}

}

}

#### mGlobalActions.showDialog();//显示关机对话框

##### mGlobalActions.showDialog();//显示关机对话框

/\*\*

\* Show the global actions dialog (creating if necessary)

\* @param keyguardShowing True if keyguard is showing

\*/

public void showDialog(boolean keyguardShowing, boolean isDeviceProvisioned) {

mKeyguardShowing = keyguardShowing;

mDeviceProvisioned = isDeviceProvisioned;

if (mDialog != null) {

mDialog.dismiss();

mDialog = null;

// Show delayed, so that the dismiss of the previous dialog completes

mHandler.sendEmptyMessage(MESSAGE\_SHOW);

} else {

handleShow();

}

}

##### 调用HandleShow()方法创建GlobalActionsDialog对话框

private void handleShow() {

awakenIfNecessary();

//创建关机对话框

mDialog = createDialog();

prepareDialog();

// 对话框视图只有一个item,当手单按时回调onPress()函数

if (mAdapter.getCount() == 1

&& mAdapter.getItem(0) instanceof SinglePressAction

&& !(mAdapter.getItem(0) instanceof LongPressAction)) {

((SinglePressAction) mAdapter.getItem(0)).onPress();

}

//否则显示关机对话框，设置对话框参数属性

else {

WindowManager.LayoutParams attrs = mDialog.getWindow().getAttributes();

attrs.setTitle("GlobalActions");

mDialog.getWindow().setAttributes(attrs);

mDialog.show();

mDialog.getWindow().getDecorView().setSystemUiVisibility(View.STATUS\_BAR\_DISABLE\_EXPAND);

}

}

/\*\*

\* Create the global actions dialog.

\* @return A new dialog.

\*/

private GlobalActionsDialog createDialog() {

// Simple toggle style if there's no vibrator, otherwise use a tri-state

if (!mHasVibrator) {

mSilentModeAction = new SilentModeToggleAction();

} else {

mSilentModeAction = new SilentModeTriStateAction(mContext, mAudioManager, mHandler);

}

//飞行模式

mAirplaneModeOn = new ToggleAction(

R.drawable.ic\_lock\_airplane\_mode,

R.drawable.ic\_lock\_airplane\_mode\_off,

R.string.global\_actions\_toggle\_airplane\_mode,

R.string.global\_actions\_airplane\_mode\_on\_status,

R.string.global\_actions\_airplane\_mode\_off\_status) {

void onToggle(boolean on) {

if (mHasTelephony && Boolean.parseBoolean(

SystemProperties.get(TelephonyProperties.PROPERTY\_INECM\_MODE))) {

mIsWaitingForEcmExit = true;

// Launch ECM exit dialog

Intent ecmDialogIntent =

new Intent(TelephonyIntents.ACTION\_SHOW\_NOTICE\_ECM\_BLOCK\_OTHERS, null);

ecmDialogIntent.addFlags(Intent.FLAG\_ACTIVITY\_NEW\_TASK);

mContext.startActivity(ecmDialogIntent);

} else {

changeAirplaneModeSystemSetting(on);

}

}

@Override

protected void changeStateFromPress(boolean buttonOn) {

if (!mHasTelephony) return;

// In ECM mode airplane state cannot be changed

if (!(Boolean.parseBoolean(

SystemProperties.get(TelephonyProperties.PROPERTY\_INECM\_MODE)))) {

mState = buttonOn ? State.TurningOn : State.TurningOff;

mAirplaneState = mState;

}

}

public boolean showDuringKeyguard() {

return true;

}

public boolean showBeforeProvisioning() {

return false;

}

};

//飞行模式改变

onAirplaneModeChanged();

mItems = new ArrayList<Action>();

//framework/base/core/res/res/values/config.xml(Global Actions:power、bugreport、users、reboot、airplane)

String[] defaultActions = mContext.getResources().getStringArray(

com.android.internal.R.array.config\_globalActionsList);

ArraySet<String> addedKeys = new ArraySet<String>();

//循环给对话框的的列表添加item

for (int i = 0; i < defaultActions.length; i++) {

String actionKey = defaultActions[i];

if (addedKeys.contains(actionKey)) {

// If we already have added this, don't add it again.

continue;

}

if (GLOBAL\_ACTION\_KEY\_POWER.equals(actionKey)) {

//关机

mItems.add(new PowerAction());

} else if (GLOBAL\_ACTION\_KEY\_AIRPLANE.equals(actionKey)) {

//飞行模式

mItems.add(mAirplaneModeOn);

} else if (GLOBAL\_ACTION\_KEY\_BUGREPORT.equals(actionKey)) {

if (Settings.Global.getInt(mContext.getContentResolver(),

Settings.Global.BUGREPORT\_IN\_POWER\_MENU, 0) != 0 && isCurrentUserOwner()) {

mItems.add(getBugReportAction());

}

} else if (GLOBAL\_ACTION\_KEY\_SILENT.equals(actionKey)) {

//静音

if (mShowSilentToggle) {

mItems.add(mSilentModeAction);

}

} else {

Log.e(TAG, "Invalid global action key " + actionKey);

}

// Add here so we don't add more than one.

addedKeys.add(actionKey);

}

//创建适配器

mAdapter = new MyAdapter();

AlertParams params = new AlertParams(mContext);

params.mAdapter = mAdapter;

params.mOnClickListener = this;

params.mForceInverseBackground = true;

//创建GlobalActionsDialog对象

GlobalActionsDialog dialog = new GlobalActionsDialog(mContext, params);

//触摸对话框外部，对话框消失

dialog.setCanceledOnTouchOutside(false); // Handled by the custom class.

//给对话框列表注册监听事件

dialog.getListView().setItemsCanFocus(true);

dialog.getListView().setLongClickable(true);

dialog.getListView().setOnItemLongClickListener(

new AdapterView.OnItemLongClickListener() {

@Override

public boolean onItemLongClick(AdapterView<?> parent, View view, int position,

long id) {

final Action action = mAdapter.getItem(position);

if (action instanceof LongPressAction) {

return ((LongPressAction) action).onLongPress();

}

return false;

}

});

dialog.getWindow().setType(WindowManager.LayoutParams.TYPE\_KEYGUARD\_DIALOG);

dialog.setOnDismissListener(this);

return dialog;

}

149

##### 关机Action:PowerAction

private final class PowerAction extends SinglePressAction implements LongPressAction {

private PowerAction() {

super(com.android.internal.R.drawable.ic\_lock\_power\_off,

R.string.global\_action\_power\_off);

}

@Override

public boolean onLongPress() {

UserManager um = (UserManager) mContext.getSystemService(Context.USER\_SERVICE);

if (!um.hasUserRestriction(UserManager.DISALLOW\_SAFE\_BOOT)) {

mWindowManagerFuncs.rebootSafeMode(true);

return true;

}

return false;

}

@Override

public boolean showDuringKeyguard() {

return true;

}

@Override

public boolean showBeforeProvisioning() {

return true;

}

@Override

public void onPress() {

// shutdown by making sure radio and power are handled accordingly.

mWindowManagerFuncs.shutdown(false /\* confirm \*/);

}

}

##### mWindowManagerFuncs.shutdown

onPress()和onLongpress()是分别处理单击和长按的方法，那么WindownManagerService.java实现了WindowManagerPolicy.WindowManagerFuncs接口，所以实际上调用的是WindownManagerService.java类的shutdown()、rebootSafeMode()方法。

framework/base/services/core/java/com/android/server/wm/WindowManagerService.java

最终都是调用ShutdownThread.java的方法。

### ShutdownThread

该类路径:framework/base/services/core/java/com/android/server/power/ShutdownThread.java

最后调用shutdownInner()

|  |
| --- |
| static void shutdownInner(final Context context, boolean confirm) {  // ensure that only one thread is trying to power down.  // any additional calls are just returned  synchronized (sIsStartedGuard) {  if (sIsStarted) {  Log.d(TAG, "Request to shutdown already running, returning.");  return;  }  }  final int longPressBehavior = context.getResources().getInteger(  com.android.internal.R.integer.config\_longPressOnPowerBehavior);  final int resourceId = mRebootSafeMode  ? com.android.internal.R.string.reboot\_safemode\_confirm  : (longPressBehavior == 2  ? com.android.internal.R.string.shutdown\_confirm\_question  : com.android.internal.R.string.shutdown\_confirm);  Log.d(TAG, "Notifying thread to start shutdown longPressBehavior=" + longPressBehavior);  if (confirm) {  final CloseDialogReceiver closer = new CloseDialogReceiver(context);  if (sConfirmDialog != null) {  sConfirmDialog.dismiss();  }  //创建关机确认对话框  sConfirmDialog = new AlertDialog.Builder(context)  .setTitle(mRebootSafeMode  ? com.android.internal.R.string.reboot\_safemode\_title  : com.android.internal.R.string.power\_off)  .setMessage(resourceId)  .setPositiveButton(com.android.internal.R.string.yes, new DialogInterface.OnClickListener() {  public void onClick(DialogInterface dialog, int which) {  //进入关机流程  beginShutdownSequence(context);  }  })  .setNegativeButton(com.android.internal.R.string.no, null)  .create();  closer.dialog = sConfirmDialog;  sConfirmDialog.setOnDismissListener(closer);  sConfirmDialog.getWindow().setType(WindowManager.LayoutParams.TYPE\_KEYGUARD\_DIALOG);  sConfirmDialog.show();  } else {  //不需要关机确认对话框，直接进入关机流程  beginShutdownSequence(context);  }  } |

### beginShutdownSequence开始进入关机流程

|  |
| --- |
| private static void beginShutdownSequence(Context context) {  synchronized (sIsStartedGuard) {  if (sIsStarted) {  Log.d(TAG, "Shutdown sequence already running, returning.");  return;  }  sIsStarted = true;  }  // Throw up a system dialog to indicate the device is rebooting / shutting down.  ProgressDialog pd = null;  // Path 1: Reboot to recovery and install the update  // Condition: mRebootReason == REBOOT\_RECOVERY and mRebootUpdate == True  // (mRebootUpdate is set by checking if /cache/recovery/uncrypt\_file exists.)  // UI: progress bar  //  // Path 2: Reboot to recovery for factory reset  // Condition: mRebootReason == REBOOT\_RECOVERY  // UI: spinning circle only (no progress bar)  //  // Path 3: Regular reboot / shutdown  // Condition: Otherwise  // UI: spinning circle only (no progress bar)  //检查是否是定时关机  if(!sIsPowerOffAlarm){  //非定时关机  pd = new ProgressDialog(context);  if (PowerManager.REBOOT\_RECOVERY.equals(mRebootReason)) {  mRebootUpdate = new File(UNCRYPT\_PACKAGE\_FILE).exists();  if (mRebootUpdate) {  //Reboot to Recovery Progress Dialog. This is shown before it reboots to recovery  pd.setTitle(context.getText(com.android.internal.R.string.reboot\_to\_update\_title));  pd.setMessage(context.getText(  com.android.internal.R.string.reboot\_to\_update\_prepare));  pd.setMax(100);  pd.setProgressNumberFormat(null);  pd.setProgressStyle(ProgressDialog.STYLE\_HORIZONTAL);  pd.setProgress(0);  pd.setIndeterminate(false);  } else {  // Factory reset path. Set the dialog message accordingly.  //恢复出厂设置对话框属性设置Reboot to Recovery for factory reset  pd.setTitle(context.getText(com.android.internal.R.string.reboot\_to\_reset\_title));  pd.setMessage(context.getText(  com.android.internal.R.string.reboot\_to\_reset\_message));  pd.setIndeterminate(true);  }  } else {  //关机Shutdown Progress Dialog. This is shown if the user chooses to power off the phone  pd.setTitle(context.getText(com.android.internal.R.string.power\_off));  pd.setMessage(context.getText(com.android.internal.R.string.shutdown\_progress));  pd.setIndeterminate(true);  }  pd.setCancelable(false);  pd.getWindow().setType(WindowManager.LayoutParams.TYPE\_KEYGUARD\_DIALOG);  }  String[] bootcmd = {"bootanimation",  "shutdown"} ;  try {  ////执行关机动画  Log.i(TAG, "exec the bootanimation ");  SystemProperties.set("service.bootanim.exit", "0");  Runtime.getRuntime().exec(bootcmd);  } catch (Exception e){  Log.e(TAG,"bootanimation command exe err!");  }  }  //初始化关机线程  sInstance.mProgressDialog = pd;  sInstance.mContext = context;  sInstance.mPowerManager = (PowerManager)context.getSystemService(Context.POWER\_SERVICE);  // make sure we never fall asleep again  sInstance.mCpuWakeLock = null;  try {  sInstance.mCpuWakeLock = sInstance.mPowerManager.newWakeLock(  PowerManager.PARTIAL\_WAKE\_LOCK, TAG + "-cpu");  sInstance.mCpuWakeLock.setReferenceCounted(false);  sInstance.mCpuWakeLock.acquire();  } catch (SecurityException e) {  Log.w(TAG, "No permission to acquire wake lock", e);  sInstance.mCpuWakeLock = null;  }  // also make sure the screen stays on for better user experience  sInstance.mScreenWakeLock = null;  if (sInstance.mPowerManager.isScreenOn()) {  try {  sInstance.mScreenWakeLock = sInstance.mPowerManager.newWakeLock(  PowerManager.FULL\_WAKE\_LOCK, TAG + "-screen");  sInstance.mScreenWakeLock.setReferenceCounted(false);  sInstance.mScreenWakeLock.acquire();  } catch (SecurityException e) {  Log.w(TAG, "No permission to acquire wake lock", e);  sInstance.mScreenWakeLock = null;  }  }  // start the thread that initiates shutdown  //启动关机线程  if(sIsPowerOffAlarm){  Looper.prepare();  sInstance.mHandler = new Handler();  Log.d(TAG,">>> before start <<<");  sInstance.start();  Looper.loop();  }else{  sInstance.mHandler = new Handler() {  };  sInstance.start();  }  } |

### run()

|  |
| --- |
| /\*\*  \* Makes sure we handle the shutdown gracefully.  \* Shuts off power regardless of radio and bluetooth state if the alloted time has passed.  \*/  public void run() {  BroadcastReceiver br = new BroadcastReceiver() {  @Override public void onReceive(Context context, Intent intent) {  // We don't allow apps to cancel this, so ignore the result.  //用于接收关机广播，bu,不允许appcancel this,忽略  actionDone();  }  };  /\*  \* Write a system property in case the system\_server reboots before we  \* get to the actual hardware restart. If that happens, we'll retry at  \* the beginning of the SystemServer startup.  \*/  {  String reason = (mReboot ? "1" : "0") + (mRebootReason != null ? mRebootReason : "");  SystemProperties.set(SHUTDOWN\_ACTION\_PROPERTY, reason);  }  /\*  \* If we are rebooting into safe mode, write a system property  \* indicating so.  \*/  if (mRebootSafeMode) {  SystemProperties.set(REBOOT\_SAFEMODE\_PROPERTY, "1");  }  Log.i(TAG, "Sending shutdown broadcast...");  // First send the high-level shut down broadcast.发送关机广播  mActionDone = false;  Intent intent = new Intent(Intent.ACTION\_SHUTDOWN);  intent.addFlags(Intent.FLAG\_RECEIVER\_FOREGROUND);  mContext.sendOrderedBroadcastAsUser(intent,  UserHandle.ALL, null, br, mHandler, 0, null, null);  //等待10s，前面定义的广播接收器收到关机广播时mActionDone设置为true，同时取消等待  final long endTime = SystemClock.elapsedRealtime() + MAX\_BROADCAST\_TIME;  synchronized (mActionDoneSync) {  while (!mActionDone) {  long delay = endTime - SystemClock.elapsedRealtime();  if (delay <= 0) {  Log.w(TAG, "Shutdown broadcast timed out");  break;  } else if (mRebootUpdate) {  int status = (int)((MAX\_BROADCAST\_TIME - delay) \* 1.0 \*  BROADCAST\_STOP\_PERCENT / MAX\_BROADCAST\_TIME);  sInstance.setRebootProgress(status, null);  }  try {  mActionDoneSync.wait(Math.min(delay, PHONE\_STATE\_POLL\_SLEEP\_MSEC));  } catch (InterruptedException e) {  }  }  }  if (mRebootUpdate) {  sInstance.setRebootProgress(BROADCAST\_STOP\_PERCENT, null);  }  Log.i(TAG, "Shutting down activity manager...");  //10s内关闭ActivityManager服务  final IActivityManager am =  ActivityManagerNative.asInterface(ServiceManager.checkService("activity"));  if (am != null) {  try {  am.shutdown(MAX\_BROADCAST\_TIME);  } catch (RemoteException e) {  }  }  if (mRebootUpdate) {  sInstance.setRebootProgress(ACTIVITY\_MANAGER\_STOP\_PERCENT, null);  }  Log.i(TAG, "Shutting down package manager...");  //关掉PackageManager服务  final PackageManagerService pm = (PackageManagerService)  ServiceManager.getService("package");  if (pm != null) {  pm.shutdown();  }  if (mRebootUpdate) {  sInstance.setRebootProgress(PACKAGE\_MANAGER\_STOP\_PERCENT, null);  }  // Shutdown radios.12秒内关闭收音机  shutdownRadios(MAX\_RADIO\_WAIT\_TIME);  if (mRebootUpdate) {  sInstance.setRebootProgress(RADIO\_STOP\_PERCENT, null);  }  // Shutdown MountService to ensure media is in a safe state  IMountShutdownObserver observer = new IMountShutdownObserver.Stub() {  public void onShutDownComplete(int statusCode) throws RemoteException {  Log.w(TAG, "Result code " + statusCode + " from MountService.shutdown");  actionDone();  }  };  Log.i(TAG, "Shutting down MountService");  // Set initial variables and time out time.  //20s内关闭MountService服务  mActionDone = false;  final long endShutTime = SystemClock.elapsedRealtime() + MAX\_SHUTDOWN\_WAIT\_TIME;  synchronized (mActionDoneSync) {  try {  final IMountService mount = IMountService.Stub.asInterface(  ServiceManager.checkService("mount"));  if (mount != null) {  mount.shutdown(observer);  } else {  Log.w(TAG, "MountService unavailable for shutdown");  }  } catch (Exception e) {  Log.e(TAG, "Exception during MountService shutdown", e);  }  while (!mActionDone) {  long delay = endShutTime - SystemClock.elapsedRealtime();  if (delay <= 0) {  Log.w(TAG, "Shutdown wait timed out");  break;  } else if (mRebootUpdate) {  int status = (int)((MAX\_SHUTDOWN\_WAIT\_TIME - delay) \* 1.0 \*  (MOUNT\_SERVICE\_STOP\_PERCENT - RADIO\_STOP\_PERCENT) /  MAX\_SHUTDOWN\_WAIT\_TIME);  status += RADIO\_STOP\_PERCENT;  sInstance.setRebootProgress(status, null);  }  try {  mActionDoneSync.wait(Math.min(delay, PHONE\_STATE\_POLL\_SLEEP\_MSEC));  } catch (InterruptedException e) {  }  }  }  if (mRebootUpdate) {  sInstance.setRebootProgress(MOUNT\_SERVICE\_STOP\_PERCENT, null);  // If it's to reboot to install update, invoke uncrypt via init service.  uncrypt();  }  //重启或关机  rebootOrShutdown(mContext, mReboot, mRebootReason);  } |

### 重启或关机

/\*\*

\* Do not call this directly. Use {@link #reboot(Context, String, boolean)}

\* or {@link #shutdown(Context, boolean)} instead.

\*

\* @param context Context used to vibrate or null without vibration

\* @param reboot true to reboot or false to shutdown

\* @param reason reason for reboot

\*/

public static void rebootOrShutdown(final Context context, boolean reboot, String reason) {

if (reboot) {

Log.i(TAG, "Rebooting, reason: " + reason);

PowerManagerService.lowLevelReboot(reason);

Log.e(TAG, "Reboot failed, will attempt shutdown instead");

} else if (SHUTDOWN\_VIBRATE\_MS > 0 && context != null) {

// vibrate before shutting down

//关机前震动

Vibrator vibrator = new SystemVibrator(context);

try {

//震动时间、属性设置

vibrator.vibrate(SHUTDOWN\_VIBRATE\_MS, VIBRATION\_ATTRIBUTES);

} catch (Exception e) {

// Failure to vibrate shouldn't interrupt shutdown. Just log it.

Log.w(TAG, "Failed to vibrate during shutdown.", e);

}

// vibrator is asynchronous so we need to wait to avoid shutting down too soon.

//先休眠一会，震动结束再关机。

try {

Thread.sleep(SHUTDOWN\_VIBRATE\_MS);

} catch (InterruptedException unused) {

}

}

// Shutdown power

Log.i(TAG, "Performing low-level shutdown...");

//调用PowerManagerService的方法关闭电源。

PowerManagerService.lowLevelShutdown();

}

run()方法主要完成以下一些工作：   
(1)发送关机广播ACTION\_SHUTDOWN；   
(2)关闭服务；   
(3)关闭Radio   
(4)设置关机震动   
(5)调用PowerManagerService的方法关闭电源。

### PowerManagerService

路径：

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

/\*\*

\* Low-level function turn the device off immediately, without trying

\* to be clean. Most people should use {@link ShutdownThread} for a clean shutdown.

\*/

public static void lowLevelShutdown() {

**SystemProperties.set("sys.powerctl", "shutdown");**

}

**"", "shutdown**

## 系统应用广播关机

这个需要系统权限

*//"android.intent.action.ACTION\_REQUEST\_SHUTDOWN"*

*//*/\*是否显示关机确认对话框 \*/ Intent sdIntent = **new** Intent(Intent.ACTION\_REQUEST\_SHUTDOWN);  
sdIntent.putExtra(Intent.EXTRA\_KEY\_CONFIRM, **false**);  
sdIntent.setFlags(Intent.***FLAG\_ACTIVITY\_NEW\_TASK***);  
getContext().startActivity(sdIntent);

2.这个也需要

PowerManager pm = (PowerManager)DJIApplication.*getAppContext*().getSystemService(Context.***POWER\_SERVICE***);  
pm.reboot(**"recovery"**);

Neither user 10038 nor current process has android.permission.REBOOT.

Root后，执行cmd，常用于c语言

adb shell am broadcast -a android.intent.action.ACTION\_REQUEST\_SHUTDOWN --ez android.intent.extra.KEY\_CONFIRM false

root权限！下面可以行

adb shell am start -a android.intent.action.ACTION\_REQUEST\_SHUTDOWN

--ez android.intent.extra.KEY\_CONFIRM false

## BatteryService低电量自动关机

 Android 通过BatteryService对电量进行自动管理。在BatteryService.java中从jni层获得电量mBatteryLevel,  并根据mBatteryLevel判断手机电量是否过低，然后发出警告或声音提醒，并且太低时还会自动关机。

BatteryService.shutdownIfNoPower();//如果电量不够，弹出关机对话框

在BatteryService.java中判断当前电量是否过低:

1. /\* The ACTION\_BATTERY\_LOW broadcast is sent in these situations:
2. \* - is just un-plugged (previously was plugged) and battery level is
3. \*   less than or equal to WARNING, or
4. \* - is not plugged and battery level falls to WARNING boundary
5. \*   (becomes <= mLowBatteryWarningLevel).
6. \*/
7. **final** **boolean** sendBatteryLow = !plugged
8. && mBatteryStatus != BatteryManager.BATTERY\_STATUS\_UNKNOWN
9. && mBatteryLevel <= mLowBatteryWarningLevel
10. && (oldPlugged || mLastBatteryLevel > mLowBatteryWarningLevel);

如果当前电量小于警告电量（在config.xml中 <integer name="config\_lowBatteryWarningLevel">15</integer>）则弹出电量低提示。

或者电量为0（当然这个有误差也可能是5%时就自动关机）时自动关机。本质也是基于广播完成的

**[java]** [view plain](http://blog.csdn.net/wangbaochu/article/details/44406681) [copy](http://blog.csdn.net/wangbaochu/article/details/44406681)

1. **private** **void** More ...shutdownIfNoPowerLocked() {
2. // shut down gracefully if our battery is critically low and we are not powered.
3. // wait until the system has booted before attempting to display the shutdown dialog.
4. **if** (mBatteryLevel == 0 && !isPoweredLocked(BatteryManager.BATTERY\_PLUGGED\_ANY)) {
5. mHandler.post(**new** Runnable() {
6. @Override
7. **public** **void** More ...run() {
8. **if** (ActivityManagerNative.isSystemReady()) {
9. Intent intent = **new** Intent(Intent.ACTION\_REQUEST\_SHUTDOWN);
10. intent.putExtra(Intent.EXTRA\_KEY\_CONFIRM, **false**);
11. intent.setFlags(Intent.FLAG\_ACTIVITY\_NEW\_TASK);
12. mContext.startActivityAsUser(intent, UserHandle.CURRENT);
13. }
14. }
15. });
16. }
17. }

## 脚本关闭

setprop sys.powerctl shutdown！！！

adb reboot

整个android系统，只有PowerMS的，recover，system/core

这是本质，系统应用层最终都是这里，最终由init进程完成关机指令！

## Init关机

在属性系统里面会解析sys前缀的属性，并执行powerctl的参数shutdown

//device/rockship/zs600/init.rc

sys.powerctl 的配置在init.rc 当中，可以参考  
on property:sys.powerctl=\*  
  powerctl ${sys.powerctl}

=》在init.rc中和一个相关的on 触发器  
396 on property:sys.powerctl=\*  
397    powerctl ${sys.powerctl}

=》会调用 builtins.c 的，参考间reboot,recovery  
int do\_powerctl(int nargs, char \*\*args)

=>  
return android\_reboot(cmd, 0, reboot\_target);  
cmd = ANDROID\_RB\_RESTART2 //在android\_reboot.h中#define ANDROID\_RB\_RESTART2 0xDEAD0003  
reboot\_target =第二个参考recovery

 =》  
/system/core/libcutils/android\_reboot.c 中的执行android\_reboot();

case ANDROID\_RB\_RESTART2:                                                                                                                          
121            ret = \_\_reboot(LINUX\_REBOOT\_MAGIC1, LINUX\_REBOOT\_MAGIC2,  
122                            LINUX\_REBOOT\_CMD\_RESTART2, arg); //arg = recovery

=>  
这个在bionic/libc/include/sys/reboot.h中定义的。说明这是一个标准的系统调用  
extern int \_\_reboot(int, int, int, void \*);

具体位于bionic/libc/arch-arm/syscalls/\_\_reboot.S  
  1 /\* autogenerated by gensyscalls.py \*/  
  2 #include <asm/unistd.h>  
  3 #include <linux/err.h>  
  4 #include <machine/asm.h>  
  5                                                                                                                                                            
  6 ENTRY(\_\_reboot)  
  7    mov    ip, r7  
  8    ldr    r7, =\_\_NR\_reboot  
  9    swi    #0  
10    mov    r7, ip  
11    cmn    r0, #(MAX\_ERRNO + 1)  
12    bxls    lr  
13    neg    r0, r0  
14    b      \_\_set\_errno  
15 END(\_\_reboot)

=》最后会进入Kernel  
kernel/sys.c  
    case LINUX\_REBOOT\_CMD\_RESTART2:  
          if (strncpy\_from\_user(&buffer[0], arg, sizeof(buffer) - 1) < 0) {  
              ret = -EFAULT;  
              break;  
          }  
          buffer[sizeof(buffer) - 1] = '\0';

          kernel\_restart(buffer);  //buffer = arg= recovery  
          break;

=>machine\_restart()

=>arch/arm/kernel/process.c  
void machine\_restart(char \*cmd)  
{  
    local\_irq\_disable();  
    smp\_send\_stop();

    /\* Flush the console to make sure all the relevant messages make it  
    \* out to the console drivers \*/  
    arm\_machine\_flush\_console();

    arm\_pm\_restart(reboot\_mode, cmd);

    /\* Give a grace period for failure to restart of 1s \*/  
    mdelay(1000); // 1s之内没有restart完成就reboot失败。

    /\* Whoops - the platform was unable to reboot. Tell the user! \*/  
    printk("Reboot failed -- System halted\n");  
    local\_irq\_disable();  
    while (1);  
}

=》  
void (\*arm\_pm\_restart)(char str, const char \*cmd) = arm\_machine\_restart;  
=》  
void arm\_machine\_restart(char mode, const char \*cmd)  
{  
...  
aml\_write\_reg32(P\_AO\_RTI\_STATUS\_REG1, reboot\_reason); //这一个标志寄存器  
...

arch\_reset(mode, cmd); // mode = 'h' cmd = "recovery"  
}

差不多了，尝试就如此吧。

那么有必要了解这个寄存器P\_AO\_RTI\_STATUS\_REG1在下次启动的时候，uboot是什么时候读。这就可以实现一些关机之后的状态保存。

## 其他关机调用grep

|  |
| --- |
| key@Linux-in:~/work/rkandroid5.1\_v1.00$ sgrep sys.powerctl  ./system/core/include/cutils/android\_reboot.h:28:#define ANDROID\_RB\_PROPERTY "sys.powerctl"  ./bootable/recovery/uncrypt/uncrypt.c:357: property\_set("sys.powerctl", "reboot,recovery");  ./frameworks/base/services/core/java/com/android/server/power/PowerManagerService.java:2733: SystemProperties.set("sys.powerctl", "shutdown");  ./frameworks/base/services/core/java/com/android/server/power/PowerManagerService.java:2751: // setting sys.powerctl directly we'll start the  ./frameworks/base/services/core/java/com/android/server/power/PowerManagerService.java:2761: SystemProperties.set("sys.powerctl", "reboot," + reason);  key@Linux-in:~/work/rkandroid5.1\_v1.00$ sgrep ANDROID\_RB\_PROPERTY  ./system/vold/cryptfs.c:307: property\_set(ANDROID\_RB\_PROPERTY, "reboot");  ./system/vold/cryptfs.c:311: property\_set(ANDROID\_RB\_PROPERTY, "reboot,recovery");  ./system/vold/cryptfs.c:315: property\_set(ANDROID\_RB\_PROPERTY, "shutdown");  ./system/core/reboot/reboot.c:66: ret = property\_set(ANDROID\_RB\_PROPERTY, property\_val);  ./system/core/adb/services.c:130: ret = property\_set(ANDROID\_RB\_PROPERTY, property\_val);  ./system/core/include/cutils/android\_reboot.h:28:#define ANDROID\_RB\_PROPERTY "sys.powerctl"  ./bootable/recovery/recovery.cpp:2968: property\_set(ANDROID\_RB\_PROPERTY, "shutdown,");  ./bootable/recovery/recovery.cpp:2973: property\_set(ANDROID\_RB\_PROPERTY, "reboot,bootloader");  ./bootable/recovery/recovery.cpp:2978: property\_set(ANDROID\_RB\_PROPERTY, "reboot,");  key@Linux-in:~/work/rkandroid5.1\_v1.00$  key@Linux-in:~/work/rkandroid5.1\_v1.00$ jgrep ACTION\_REQUEST\_SHUTDOWN  ./opt/DJIService/src/android/rockchip/update/service/FirmwareUpdatingActivity.java:407: Intent intent = new Intent("android.intent.action.ACTION\_REQUEST\_SHUTDOWN");  ./packages/services/Telephony/src/com/android/phone/OtaUtils.java:1013: Intent shutdown = new Intent(Intent.ACTION\_REQUEST\_SHUTDOWN);  ./frameworks/base/services/core/java/com/android/server/BatteryService.java:275: Intent intent = new Intent(Intent.ACTION\_REQUEST\_SHUTDOWN);  ./frameworks/base/services/core/java/com/android/server/BatteryService.java:294: Intent intent = new Intent(Intent.ACTION\_REQUEST\_SHUTDOWN);  ./frameworks/base/packages/SystemUI/src/com/android/systemui/statusbar/oswin/WinMetroWindow.java:412: intent = new Intent(Intent.ACTION\_REQUEST\_SHUTDOWN);  ./frameworks/base/core/java/android/content/Intent.java:2046: public static final String ACTION\_REQUEST\_SHUTDOWN = "android.intent.action.ACTION\_REQUEST\_SHUTDOWN";  ./frameworks/base/core/java/android/content/Intent.java:3246: \* Set to true in {@link #ACTION\_REQUEST\_SHUTDOWN} to request confirmation from the user  -------------------------  onOtaCloseSpcNotice()  key@Linux-in:~/work/rkandroid5.1\_v1.00$ jgrep onOtaCloseSpcNotice  ./packages/services/Telephony/src/com/android/phone/OtaUtils.java:1011: public void onOtaCloseSpcNotice() {  ./packages/services/Telephony/src/com/android/phone/OtaUtils.java:1012: if (DBG) log("onOtaCloseSpcNotice(), send shutdown intent"); |

### 系统自动关机

# Android7.1 亮度自动调节

https://blog.csdn.net/Kitty\_Landon/article/details/64128140

# 电量变化监听

# Asdf

06-12 15:54:35.032 4240-4240/com.dpad.launcher D/BatteryView2: @@@dji@@@handleLevelChanged,charging=1,level=41

06-12 15:54:43.063 4240-4240/com.dpad.launcher D/BatteryView2: @@@dji@@@handleLevelChanged,charging=1,level=41

06-12 15:54:51.085 4240-4240/com.dpad.launcher D/BatteryView2: @@@dji@@@handleLevelChanged,charging=1,level=41

06-12 15:54:59.112 4240-4240/com.dpad.launcher D/BatteryView2: @@@dji@@@handleLevelChanged,charging=1,level=41

06-12 15:55:15.167 4240-4240/com.dpad.launcher D/BatteryView2: @@@dji@@@handleLevelChanged,charging=1,level=41

06-12 15:55:23.193 4240-4240/com.dpad.launcher D/BatteryView2: @@@dji@@@handleLevelChanged,charging=1,level=41

06-12 15:55:31.219 4240-4240/com.dpad.launcher D/BatteryView2: @@@dji@@@handleLevelChanged,charging=1,level=41

06-12 15:55:32.847 4240-4240/com.dpad.launcher D/BatteryView2: @@@dji@@@handleLevelChanged,charging=1,level=41

06-12 15:55:39.246 4240-4240/com.dpad.launcher D/BatteryView2: @@@dji@@@handleLevelChanged,charging=1,level=41

## 调用关机命令

# REF

[第5章 深入理解 PowerManagerService](http://wiki.jikexueyuan.com/project/deep-android-v2/powermanagerservice.html)

[Android Low Battery 低电量处理流程](http://blog.csdn.net/wangbaochu/article/details/44406681)

[Android5.1--PowerManagerService电源管理](http://blog.csdn.net/kitty_landon/article/details/47107045)

# Task