B

android 禁止电量不足自动关机

Android Low Battery 低电量处理流程

powermanagerservice

# 主要内容

## 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

# 概述

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

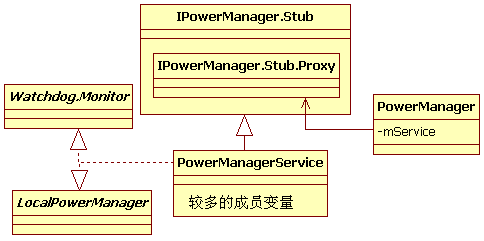


图5-1  PowerManagerService及相关类家族

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。

# 初识PowerManagerService

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：

[-->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

# BatteryService

从前面介绍PMS的代码中发现，PMS和系统中其他两个服务BatterService及BatteryStatsService均有交互，其中：

· BatteryService提供接口用于获取电池信息，充电状态等。

· BatteryStatsService主要用做用电统计，通过它可知谁是系统中的耗电大户。

下面先来介绍稍简单的BatteryService。

## 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

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。

## -->BatteryStatsService.java::BatteryStatsService构造函数

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

BatteryStatsService(String filename) {

mStats = new BatteryStatsImpl(filename);

}

图5-2展示了BSS及BSImpl的家族图谱。

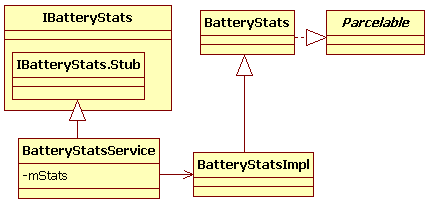


图5-2 BSS及BSImpl家族图谱

由图5-2可知：

· BSS通过成员变量mStats指向一个BSImpl类型的对象。

· BSImpl从BatteryStats类派生。更重要的是，该类实现了Parcelable接口，由此可知，BSImpl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BSImpl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BSImpl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BSImpl，下面就来分析它。

## BSImpl

BSImpl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BSImpl使用了不止一种的计量工具。

### 计量工具和统计对象介绍

BSImpl一共使用了4种计量工具，如图5-3所示。

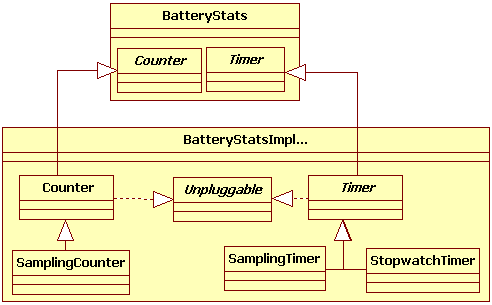


图5-3 计量工具图例

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。

· BSImpl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。

· BSImpl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BSImpl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。

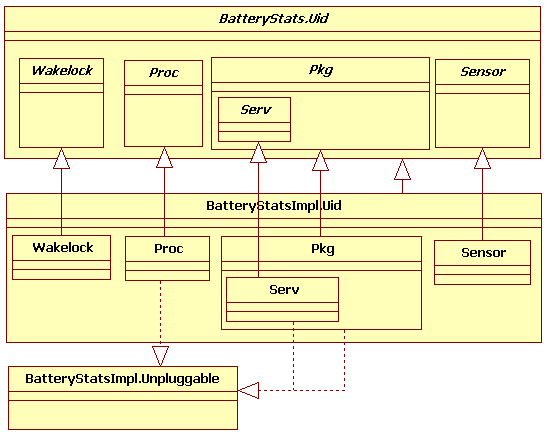


图5-4  BatteryStats.Uid家族

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。

## BSImpl流程分析

### 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BSImpl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BSImpl创建的JournaledFile文件

mBatteryStatsService.getActiveStatistics().readLocked();

mBatteryStatsService.getActiveStatistics().writeAsyncLocked();

//BSImpl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BSImpl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BSImpl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BSImpl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BSImpl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~~

~~......~~

~~}~~

# 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。

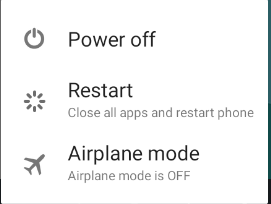
[①] config.xml文件的全路径是4.0源码/frameworks/base/core/res/res/values/config.xml。

[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

## 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) {

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8

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();

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

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);

## 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。

### 系统自动关机

## 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)

# Task

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

（android 关机/重启）Android关机/重启流程解析 （2）-----实例解析(多种实现方式)

# PowerManagerService

第5章 深入理解 PowerManagerService

<http://wiki.jikexueyuan.com/project/deep-android-v2/powermanagerservice.html>

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

<http://blog.csdn.net/kitty_landon/article/details/47107045>

# [Android5.1 PowerManagerService深入分析](http://blog.csdn.net/kc58236582/article/details/45537749)

<http://blog.csdn.net/kc58236582/article/details/45537749>

[Android5.1AlarmManagerService深入分析（Android4.4补充）](http://blog.csdn.net/kc58236582/article/details/45666709)

<http://blog.csdn.net/kc58236582/article/details/45666709>

# Android之AlarmManagerService源码分析

<http://www.procedurego.com/article/128389.html>

<http://blog.csdn.net/zhangyongfeiyong/article/details/52224300>

<https://github.com/clarkehe/Android/wiki/Android%E4%BC%91%E7%9C%A0%E6%9C%BA%E5%88%B6>

休眠机制

# [AlarmManager 实现闹钟的基本功能](http://www.cnblogs.com/rainly/p/3437627.html)