[高通平台Android源码分析之Linux内核设备树(DT - Device Tree)](http://huaqianlee.github.io/2015/08/19/Android/%E9%AB%98%E9%80%9A%E5%B9%B3%E5%8F%B0Android%E6%BA%90%E7%A0%81%E5%88%86%E6%9E%90%E4%B9%8BLinux%E5%86%85%E6%A0%B8%E8%AE%BE%E5%A4%87%E6%A0%91-DT-Device-Tree-dts%E6%96%87%E4%BB%B6/)

在kernel里面多了一种dts文件，因为当初自学Linux时和在第一家公司做物联网模型时都是用的比较老的内核，内核代码还比较混乱，没有采用dts这种方便简洁的格式。后面才知道这是因为Linus的一句”this whole arm thing is a fucking pain in ass“促进改革的，记得Linux早期代码里面板级细节都是在C文件中描述的，代码就显得十分臃肿和混乱。如此优化之后就显得简洁多了，并且也更易于学习、移植。

修复

# Fan

|  |  |  |
| --- | --- | --- |
| T | B | Speed |
| [0,53> |  | 0 |
| [53,58> | [0,80> | 0 |
|  | [80,100] | Max/2 |
| [58,63> | [0,50> | 0 |
|  | [50,80> | Max/2 |
|  | [80,100] | Max |

80du 从何而来，

怎么判断产品get\_broard\_ID

这么多宏定义，可读性非常差！怎么维护

风扇逻辑还是有问题。

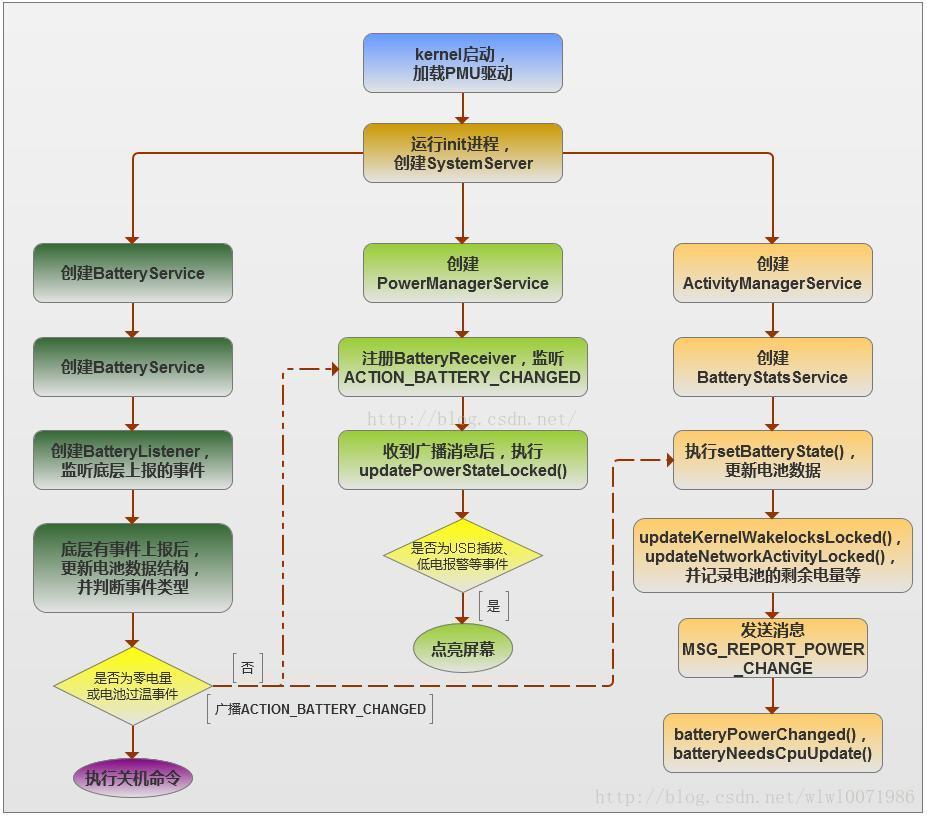
Bat:

# 电池管理

## Android 电池服务

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。

android电池服务的启动和运行流程：



Android电池服务的源码结构

         Framework\base\services\**[Java](http://lib.csdn.net/base/javase" \t "_blank" \o "Java SE知识库)**\com\android\server  
        ├── SystemServer.**[Java](http://lib.csdn.net/base/java" \t "_blank" \o "Java 知识库)**  
                 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

         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

         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

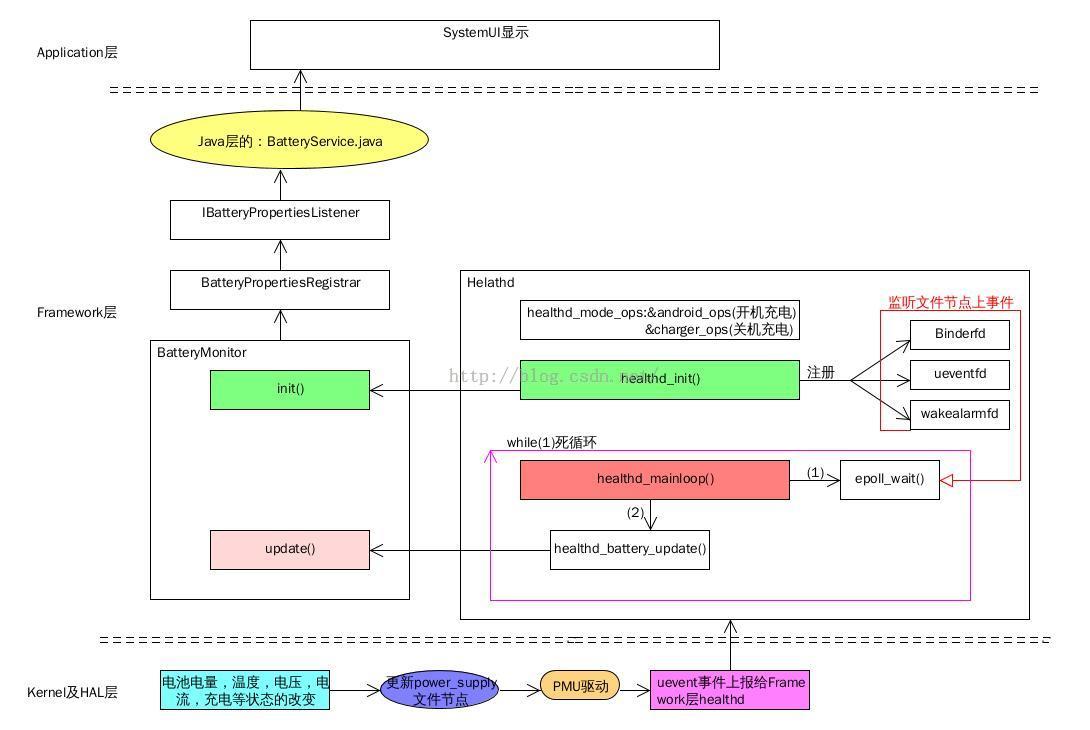
         Framework\base\core\java\com\internal\os\  
        ├── BatteryStatsImpl.java  
                 统计和记录电池参数的信息，并通知其他模块

         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](http://lib.csdn.net/base/datastructure" \t "_blank" \o "算法与数据结构知识库)**，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

## 二、Healthd

该模型向下监听来自底层的电池事件，向上传递电池数据信息给Framework层的BatteryService用来计算电池电量相关信息，

BatteryService通过传递来的数据来计算电池电量等信息，因此healthd在电池管理系统中起着承上启下的作用。



healthd的具体调用流程[深入分析android5.1 healthd](http://blog.csdn.net/kc58236582/article/details/47300413" \t "_blank)这篇文章讲得很清楚。

## 三、驱动

Android电源管理底层用的是**[Linux](http://lib.csdn.net/base/linux" \t "_blank" \o "Linux知识库)**

power\_supply框架，内核提供给电池驱动的接口是结构体power\_supply结构体。

**[cpp]** [view plain](http://blog.csdn.net/bhj1119/article/details/52947344" \t "_blank" \o "view plain) [copy](http://blog.csdn.net/bhj1119/article/details/52947344" \t "_blank" \o "copy)

1. **struct** power\_supply {
2. **const** **char** \*name;
3. **enum** power\_supply\_type type;
4. **enum** power\_supply\_property \*properties;
5. **size\_t** num\_properties;
7. **char** \*\*supplied\_to;
8. **size\_t** num\_supplicants;
10. **int** (\*get\_property)(**struct** power\_supply \*psy,
11. **enum** power\_supply\_property psp,
12. **union** power\_supply\_propval \*val);
13. **int** (\*set\_property)(**struct** power\_supply \*psy,
14. **enum** power\_supply\_property psp,
15. **const** **union** power\_supply\_propval \*val);
16. **int** (\*property\_is\_writeable)(**struct** power\_supply \*psy,
17. **enum** power\_supply\_property psp);
18. **void** (\*external\_power\_changed)(**struct** power\_supply \*psy);
19. **void** (\*set\_charged)(**struct** power\_supply \*psy);
21. /\* For APM emulation, think legacy userspace. \*/
22. **int** use\_for\_apm;
24. /\* Driver private data \*/
25. **void** \*drv\_data;//add by bhj
27. /\* private \*/
28. **struct** device \*dev;
29. **struct** work\_struct changed\_work;
30. spinlock\_t changed\_lock;
31. **bool** changed;
32. **struct** wake\_lock work\_wake\_lock;
33. **struct** delayed\_work deferred\_register\_work;
35. #ifdef CONFIG\_LEDS\_TRIGGERS
36. **struct** led\_trigger \*charging\_full\_trig;
37. **char** \*charging\_full\_trig\_name;
38. **struct** led\_trigger \*charging\_trig;
39. **char** \*charging\_trig\_name;
40. **struct** led\_trigger \*full\_trig;
41. **char** \*full\_trig\_name;
42. **struct** led\_trigger \*online\_trig;
43. **char** \*online\_trig\_name;
44. **struct** led\_trigger \*charging\_blink\_full\_solid\_trig;
45. **char** \*charging\_blink\_full\_solid\_trig\_name;
46. #endif
47. };</span>

 内核主要通过get\_property这个函数指针来获得驱动中的有关电池的信息，而这个函数在内核中只给出了声明，我们在写驱动的

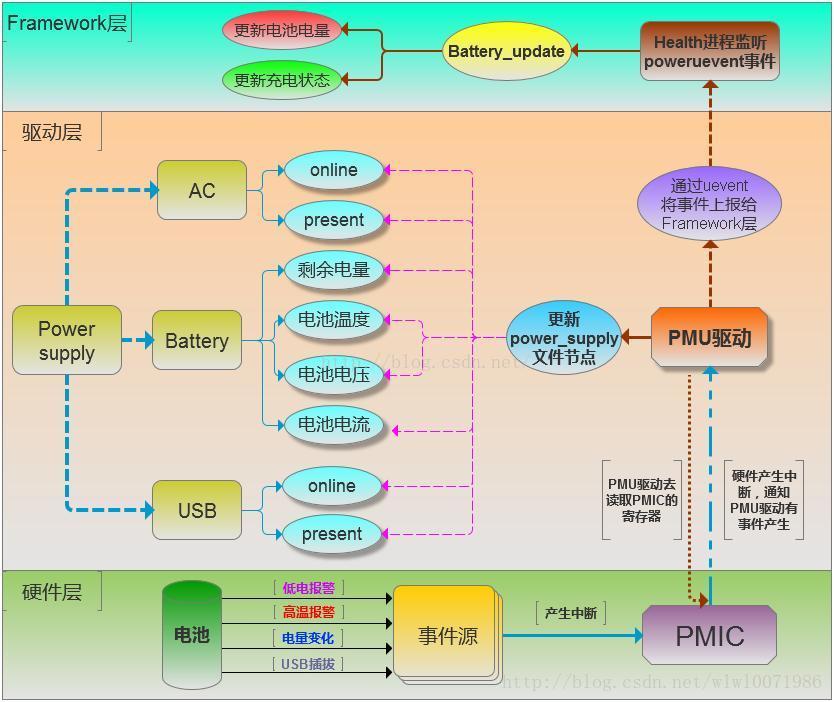
时候要自己实现这个函数，即将自己写的函数赋值给这个函数指针，当内核需要驱动中电源信息的时候就回调这个get\_property函数

。另外，我们写驱动程序的时候又要给用户提供接口，内核中提供给用户的接口就是sysfs，通过读取sysfs文件系统中文件内容，就

可以得到电源的信息。内核主要通过两个文件power\_supply\_class.c和power\_supply\_core.c，我们调用其中的函数就可以把电源（电

池，USB或AC）的信息展现给用户，有关电源的属性写在/sys/class/powersupply文件夹下（此文件夹为程序运行后所生成的）。

电池系统从底层向Framework层上报数据的流程：



## REF

[Android 电池管理系统](http://blog.csdn.net/lqxandroid2012/article/details/70050071)

[Android BatteryStatsHelper深入理解(and5.1)](http://blog.csdn.net/kc58236582/article/details/47273507)

[Android的文件系统结构](http://blog.csdn.net/conowen/article/details/7251057)

# AlarmManager

### [认领AlarmManager源码解析](https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=4&cad=rja&uact=8&ved=0ahUKEwjtpdipvOLVAhUBi5QKHcTUBPcQFgg3MAM&url=https%3A%2F%2Fgithub.com%2FLittleFriendsGroup%2FAndroidSdkSourceAnalysis%2Fissues%2F68&usg=AFQjCNEuaO5NC5s40NH1uMuWKzH_74Eg_w)

<https://github.com/LittleFriendsGroup/AndroidSdkSourceAnalysis>

setings时间设置原理

## 什么是Alarm

Alarm是**[android](http://lib.csdn.net/base/android" \t "_blank" \o "Android知识库)**提供的用于完成闹钟式定时任务的类，系统通过AlarmManager来管理所有的Alarm，Alarm支持一次性定时任务和循环定时任务，使用方式：

1. AlarmManager alarmMgr = (AlarmManager) getSystemService(Context.ALARM\_SERVICE);
2. Intent intent = new
3. Intent("android.intent.action.POWER\_USAGE\_SUMMARY");
4. alarmMgr.set(AlarmManager.ELAPSED\_REALTIME
5. , SystemClock.elapsedRealtime() + 5 \* 1000//5S后打开电池界面
6. , PendingIntent.getActivity(getApplicationContext(),0, intent, PendingIntent.FLAG\_UPDATE\_CURRENT));

Alarm和Timer以及Handler在定时任务上的区别

**相同点**：三者都可以完成定时任务，都支持一次性定时和循环定时（注：Handler可以间接支持循环定时任务）

**不同点**：Handler和Timer在定时上是类似的，二者在系统休眠的情况下无法正常工作，定时任务不会按时触发。Alarm在系统休眠的情况下可以正常工作，并且还可以决定是否唤醒系统，同时Alarm在自身不启动的情况下仍能正常收到定时任务提醒，但是当系统重启或者应用被杀死的情况下，Alarm定时任务会被取消。另外，从Android4.4开始，Alarm事件默认采用非精准方式，即定时任务可能会有小范围的提前或延后，当然我们可以强制采用精准方式，而在此之前，Alarm事件都是精准方式。

## Alarm与Binder的交互

code：AlarmManager#set

**[java]** [view plain](http://blog.csdn.net/singwhatiwanna/article/details/18448997" \t "_blank" \o "view plain) [copy](http://blog.csdn.net/singwhatiwanna/article/details/18448997" \t "_blank" \o "copy)

1. **public** **void** set(**int** type, **long** triggerAtMillis, PendingIntent operation) {
2. setImpl(type, triggerAtMillis, legacyExactLength(), 0, operation, **null**);
3. }
5. **public** **void** set(**int** type, **long** triggerAtMillis, **long** windowMillis, **long** intervalMillis,
6. PendingIntent operation, WorkSource workSource) {
7. setImpl(type, triggerAtMillis, windowMillis, intervalMillis, operation, workSource);
8. }
10. **private** **void** setImpl(**int** type, **long** triggerAtMillis, **long** windowMillis, **long** intervalMillis,
11. PendingIntent operation, WorkSource workSource) {
12. **if** (triggerAtMillis < 0) {
13. /\* NOTYET
14. if (mAlwaysExact) {
15. // Fatal error for KLP+ apps to use negative trigger times
16. throw new IllegalArgumentException("Invalid alarm trigger time "
17. + triggerAtMillis);
18. }
19. \*/
20. triggerAtMillis = 0;
21. }
23. **try** {
24. //定时任务实际上都有mService来完成，也就是说AlarmManager只是一个空壳
25. //从下面的构造方法可以看出，这个mService是IAlarmManager类型的，而IAlarmManager是一个接口
26. //如果大家了解AIDL就应该知道IAlarmManager应该是一个AIDL接口
27. mService.set(type, triggerAtMillis, windowMillis, intervalMillis, operation,
28. workSource);
29. } **catch** (RemoteException ex) {
30. }
31. }
33. AlarmManager(IAlarmManager service, Context ctx) {
34. mService = service;
36. **final** **int** sdkVersion = ctx.getApplicationInfo().targetSdkVersion;
37. mAlwaysExact = (sdkVersion < Build.VERSION\_CODES.KITKAT);
38. }

说明：我对代码进行了注释，从注释可以看出，现在我们需要去找到这个mService，其实我已经帮大家找到了，它就是AlarmManagerService，看下它的类的声明：

class AlarmManagerService extends IAlarmManager.Stub

很显然，AlarmManagerService的确实现了IAlarmManager接口，为什么是显然呢？因为按照AIDL的规范，IAlarmManager.Stub是按照如下这种方式声明的：

**[java]** [view plain](http://blog.csdn.net/singwhatiwanna/article/details/18448997" \t "_blank" \o "view plain) [copy](http://blog.csdn.net/singwhatiwanna/article/details/18448997" \t "_blank" \o "copy)

1. **public** **static** **abstract** **class** Stub **extends** Binder **implements** IAlarmManager {
3. **public** **static** IAlarmManager asInterface(IBinder obj)
4. ...
5. }

可见这个Stub类就是一个普通的Binder，只不过它实现了IAlarmManager接口。它还有一个静态方法asInterface，这个方法很有用，通过它，我们就可以将IBinder对象转换成IAlarmManager的实例，进而通过实例来调用其方法。

code：SystemServer

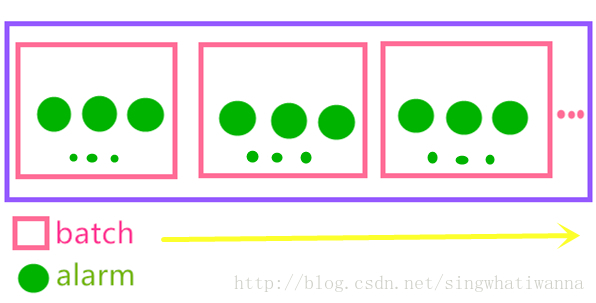
1. //main方法，由底层调用
2. **public** **static** **void** main(String[] args) {
3. //这里是关键，ServerThread被创建，同时其initAndLoop被调用
4. ServerThread thr = **new** ServerThread();
5. thr.initAndLoop();
6. }

code：ServerThread#initAndLoop

1. **public** **void** initAndLoop() {
2. //主线程Looper被创建
3. Looper.prepareMainLooper();      android.os.Process.setThreadPriority(
4. android.os.Process.THREAD\_PRIORITY\_FOREGROUND);
5. BinderInternal.disableBackgroundScheduling(**true**);
6. android.os.Process.setCanSelfBackground(**false**);
7. ...此处省略
8. //下面是各种Binder服务，从名字我们应该能够大致看出它们所对应的Manager
9. AlarmManagerService alarm = **null**;
10. Slog.i(TAG, "Alarm Manager");
11. //这里AlarmManager对应的Binder服务被创建
12. alarm = **new** AlarmManagerService(context);
13. //将AlarmManagerService加入ServiceManager中统一管理
14. ServiceManager.addService(Context.ALARM\_SERVICE, alarm);
15. ActivityManagerService.self().setWindowManager(wm);
16. }

## Alarm机制分析

在分析源码之前，我先来描述下Alarm的工作原理：从Android4.4开始，Alarm默认为非精准模式，除非显示指定采用精准模式。在非精准模式下，Alarm是批量提醒的，每个alarm根据其触发时间和最大触发时间的不同会被加入到不同的batch中，同一个batch的不同alarm是同时发生的，这样就无法实现精准闹钟，官方的解释是批量处理可以减少设备被唤醒次数以及节约电量，不过针对精准闹钟，官方预留的方法是setExact和setWindow，二者都是通过将时间窗口定义为0来实现精准闹钟的，因为时间窗口为0，意味着触发时间和最大触发时间是一样的，因为典型的情况下：最大触发时间= 触发时间 + 时间窗口。同时所有的batch是按开始时间升序排列的，在一个batch内部，不同的闹钟也是按触发时间升序排列的，所以闹钟的唤醒顺序是按照batch的排序依次触发的，而同一个batch中的alarm是同时触发的，可以用下面这个示意图来描述：

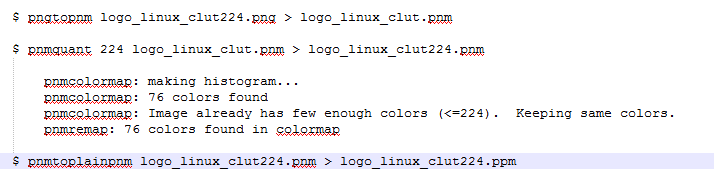


上图是示意图，系统中可以有多个batch，每个batch中可以有多个alarm。下面我们分析一下AlarmManagerService中的代码。其入口方法为set，set又调用了setImplLocked，所以我们直接看setImplLocked。

code：AlarmManagerService#setImplLocked

<http://blog.csdn.net/singwhatiwanna/article/details/18448997>

# 修改内核logo



<http://www.52rd.com/blog/Detail_RD.Blog_wuyaya_67898.html>

# TASK

[Linux内核（Kernel-3.18） - Linux Input 子系统分析](http://zhoujinjian.cc/2018/04/01/Linux%E5%86%85%E6%A0%B8%EF%BC%88Kernel-3-18%EF%BC%89-Input-%E5%AD%90%E7%B3%BB%E7%BB%9F%E5%88%86%E6%9E%90-i-wonder/)