# Android功耗优化调研

1、Android耗电统计算法

process\_Power = CPU功耗 + Wakelock功耗 + 无线电功耗 + WIFI功耗 + 蓝牙功耗 + Sensor功耗 + 相机功耗 + 闪光灯功耗

2、Android耗电量测试工具

[**battery historian**](http://blog.csdn.net/oujunli/article/details/37563067)

1、adb shell dumpsys batterystats --enable full-wake-history

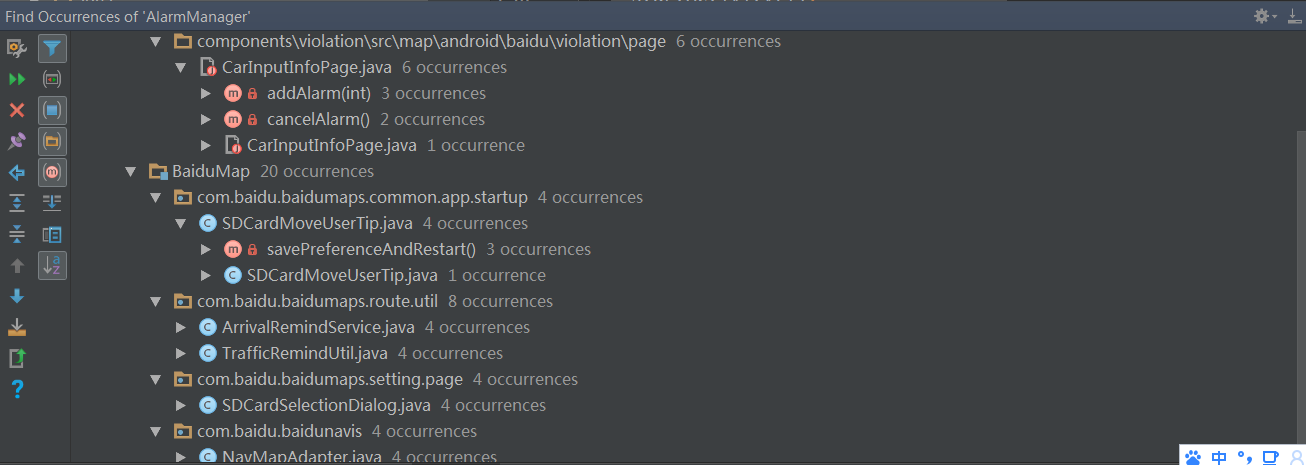
2、adb shell dumpsys batterystats –reset

3、adb bugreport |tee bugreport.txt

4、**[Python](http://lib.csdn.net/base/11" \o "undefined" \t "_blank)** historian.py -a bugreport.txt | tee battery.html

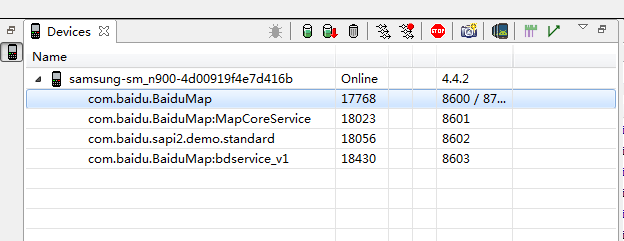
**Wakelock Detector**



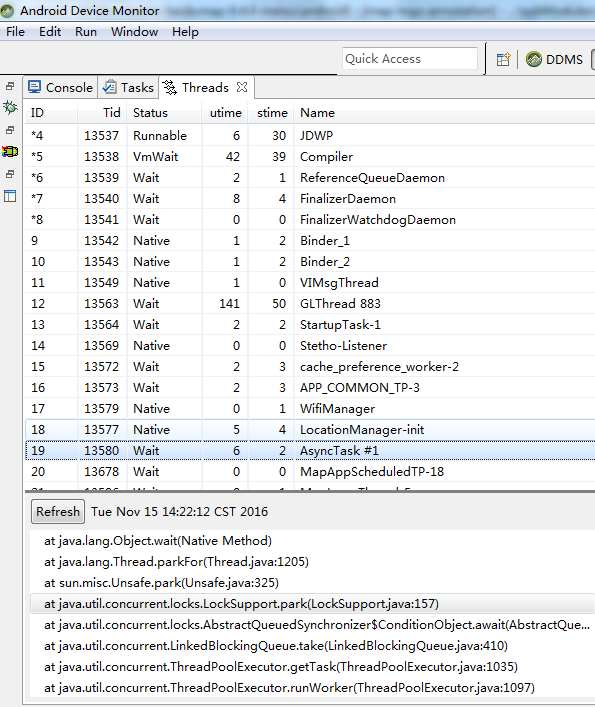


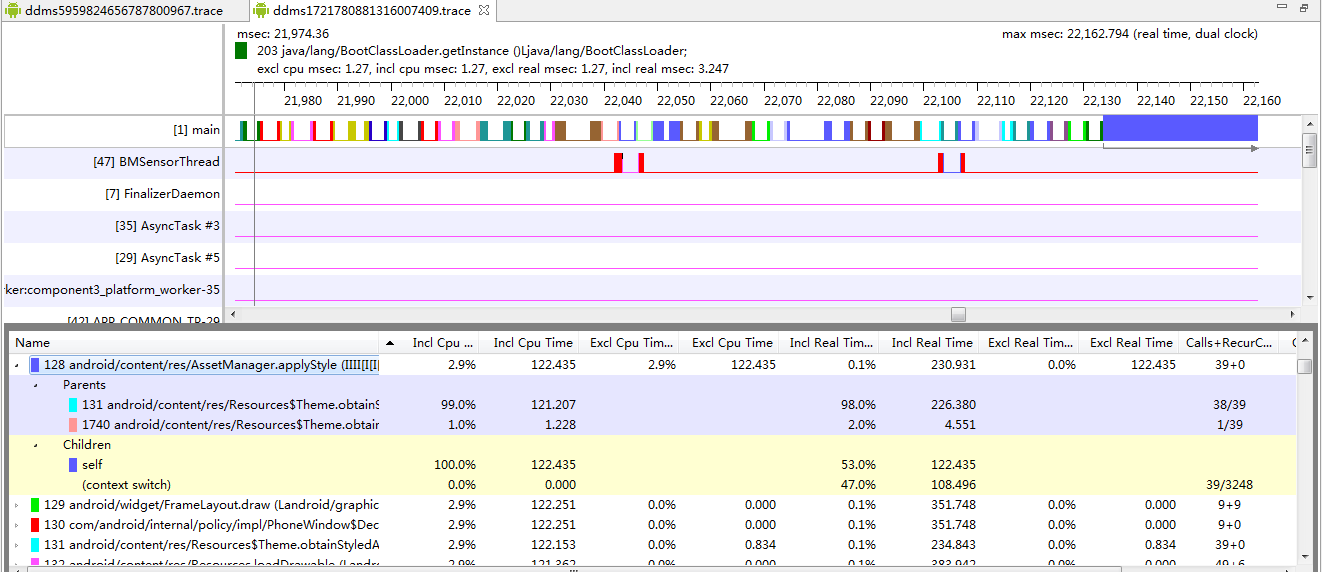
3、Android进程与线程分析工具

Android DDMS查看进程



Android DDMS查看线程





4、Android功耗过高的异常Case

PowerManager.WakeLock的API来保持CPU工作并防止屏幕关闭

蜂窝移动网络下频繁的间隔网络请求

定时任务频繁唤醒CPU

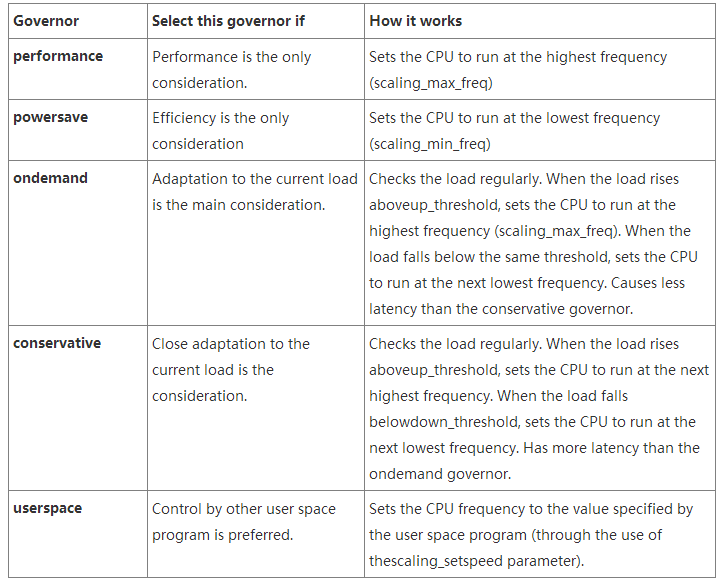
后台service使用不当

传感器使用不当

5、Android电量性能优化

* Android省电开发之CPU降频

在android系统的耗电量排行里，CPU的耗电占 了比较大的一部分比例， CPU的使用率和使用频率将直接或间接的影响电量的分配和使用



1.performance cpu的工作频率调整到最大模式，让cpu充分的工作。

2.powersave 将cpu的工作频率调整到节能模式，也就是这个模式下的cpu平率是最低的。

3. ondemand cpu在工作的时候频率会在一个最大值和最小值之间波动，当负载提高时，该调控期会自动提高cpu的频率，反之亦然。 该模式跟conservative相比，会导致更少的延迟。

4. conservative，该模式与ondemand的最大区别在于，conservative模式不会立刻在负载增加的情况下将cpu频率调整到最大，他会调整到比目前频率稍微大的频段去工作，在某种极端情况下，该模式的延迟会大于ondemand。

5. usersapce，应用程序可以通过配置文件的方式修改cpu的频率信息

public class CPUFreqSetting {

/\*\*

\* cpu cat命令大全

\* cat [%cpuFreqPath%]/cpuinfo\_cur\_freq (当前cpu频率)

\* cat [%cpuFreqPath%]/cpuinfo\_max\_freq (最大cpu频率)

\* cat [%cpuFreqPath%]/cpuinfo\_min\_freq (最小cpu频率)

\* cat [%cpuFreqPath%]/related\_cpus (cpu数量标号,从0开始,如果是双核,结果为0,1)

\* cat [%cpuFreqPath%]/scaling\_available\_frequencies (cpu所有可用频率)

\* cat [%cpuFreqPath%]/scaling\_available\_governors (cpu所有可用调控模式)

\* cat [%cpuFreqPath%]/scaling\_available\_governors (cpu所有可用调控模式)

\* cat [%cpuFreqPath%]/scaling\_cur\_freq

\* cat [%cpuFreqPath%]/scaling\_driver

\* cat [%cpuFreqPath%]/scaling\_governor

\* cat [%cpuFreqPath%]/scaling\_max\_freq

\* cat [%cpuFreqPath%]/scaling\_min\_freq

\* cat [%cpuFreqPath%]/scaling\_setspeed

\* cat [%cpuFreqPath%]/cpuinfo\_transition\_latency

\*/

private final String TAG = "SetCPU";

private final String cpuFreqPath = "/sys/devices/system/cpu/cpu0/cpufreq";

private final static String PERFORMANCE\_GOVERNOR = "performance";

private final static String POWER\_SAVE\_GOVERNOR = "performance";

private final static String ONDEMAND\_GOVERNOR = "performance";

private final static String CONSERVATIVE\_GOVERNOR = "performance";

private final static String USERSAPCE\_GOVERNOR = "performance";

// public void powerSaveGovernor() {

// List<String> governors = readCpuGovernors();

// if (governors.contains(object)) {

//

// }

// }

/\*\*

\* 获得当前CPU调控模式

\*/

public void getCpuCurGovernor() {

try {

DataInputStream is = null;

Process process = Runtime.getRuntime().exec("cat " + cpuFreqPath + "/scaling\_governor");

is = new DataInputStream(process.getInputStream());

String line = is.readLine();

} catch (IOException e) {

e.printStackTrace();

}

}

/\*\*

\* 重写CPU调控模式

\* @param governor

\* @return

\*/

private boolean writeCpuGovernor(String governor) {

DataOutputStream os = null;

byte[] buffer = new byte[256];

String command = "echo " + governor + " > " + cpuFreqPath + "/scaling\_governor";

Log.i(TAG, "command: " + command);

try {

Process process = Runtime.getRuntime().exec("su");

os = new DataOutputStream(process.getOutputStream());

os.writeBytes(command + "\n");

os.writeBytes("exit\n");

os.flush();

process.waitFor();

Log.i(TAG, "exit value = " + process.exitValue());

} catch (IOException e) {

Log.i(TAG, "writeCpuGovernor: write CPU Governor(" + governor + ") failed!");

return false;

} catch (InterruptedException e) {

e.printStackTrace();

}

return true;

}

/\*\*

\* 获得CPU所有调控模式

\* @return

\*/

private List<String> readCpuGovernors() {

List<String> governors = new ArrayList<String>();

DataInputStream is = null;

try {

Process process = Runtime.getRuntime().exec("cat " + cpuFreqPath + "/scaling\_available\_governors");

is = new DataInputStream(process.getInputStream());

String line = is.readLine();

String[] strs = line.split(" ");

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

governors.add(strs[i]);

} catch (IOException e) {

Log.i(TAG, "readCpuGovernors: read CPU Governors failed!");

}

return governors;

}

}

* Android省电开发之service

### service是否存活以及降低优先级：

假如一个service工作完成，但是来不及关掉或者kill掉，用户又看不见，所以这个service将会一直在后台运行，势必耗电。所以我们可以降低某些不常用service进程的优先级，在系统内存吃紧的情况下， 进程优先级低的service容易被系统kill掉。除此之外，可以利用监听系统广播来判断service状态是否存活，死亡即可手动kill掉。

### 用IntentService代替Service开发：

普通服务一旦启动之后，就会一直处于运行状态，必须调用stopService()或者stopSelf()方法才能让服务停止下来。为了简单的创建一个异步的、会自动停止的服务，Android专门提供了一个IntentService类。IntentService在运行完毕后自动停止，减少耗电量。

### 3. 后台执行的定时任务Alarm机制:

Service没必要一直在后台运行，这时候的定时任务显得很重要。Android的定时任务有两种实现方式，Timer类和Alarm机制。Timer有一个明显的短板，不适合长期后台运行的定时任务。为了能让电池更加耐用，每种手机都会有自己的休眠策略，Android手机就会在长时间不操作的情况下自动让CPU进入到睡眠状态，这就有可能导致Timer中的定时任务无法正常运行。而Alarm机制则不存在这种情况，它具有唤醒CPU的功能，即可以保证每次需要执行定时任务的时候CPU能正常工作。从Android 4.4之后，Alarm任务的触发时间将会变得不准确，有可能会延迟一段时间后任务才能得到执行。系统会自动检测目前有多少Alarm任务存在，然后将触发时间将近的几个任务放在一起执行，这就可以大幅度的减少CPU被唤醒的次数，从而有效延长电池的使用时间。

* Android省电开发之网络

wifi比蜂窝数据，包括2G(GPRS)、3G更省电。

a) 尽量在Wi-Fi下传输数据，可以考虑在有Wi-Fi的时候做预加载

b) 非Wi-Fi下，尽量减少网络访问，每一次后台交互都要考虑是否必须。虽然WiFi接入方式已经占到移动互联网用户的50%，但是是有些手机设置为待机关闭WiFi连接，即便有Wi-Fi信号也只能切换到蜂窝数据;

灭屏情况：灭屏传输，高负载download的时候WiFi最省电(70mA)，3G(270mA)和2G(280mA)相当，是WiFi的4倍左右;亮屏情况：亮屏传输，高负载download的时候WiFi最省电(280mA)，3G(360mA)和2G(370mA)相当，是WiFi的1.3倍左右;所以在Android应用省电开发中，我们可以在获取网络方式的方面加以考虑。

* Android省电开发之Android L5.0（ART）

安卓4.4系统中引入了全新的ART模式，相比之前流行已久的Dalvik模式有了很大的改变。通过对比，可以一眼看出 ART模式和Dalvik模式之间的区别。



在Dalvik中，应用每次运行，字节码都需要通过即时编译器转换为机器码，这回拖慢应用的运行效率，而在ART环境中，应用在第一次安装时，字节码就会预先编译成机器码，使其成为了真正的本地应用。这个过程叫做预编译。这样的话，应用的启动和执行都会变得更加快速。但是ART的缺点是预编译完的机器人占用了更大的存储空间，应用的安装需要更长的时间。但是牺牲空间时间换取省电速度，在Android应用中还是可以接受的，毕竟性能的提升，运行速度的变快，体验更流畅，电池续航更久显得更重要。

经测试，谷歌最新的Android L系统与上一版本相比续航能力可以提升36%。

### 利用JobScheduler API

过去，如果开发人员想通过后台调取服务器数据，或完成某些处理工作，应用程序必须先监听是否有事件正在发生，并为自己设定一个唤醒时间，一旦应用程序开始运行，他需要检查各种环境条件，以确定是否具备条件让它完成工作，还是需要稍后再试，这种方式不仅复杂，而且容易出错，它会不断的浪费资源，比如当 一个应用程序被唤醒后，发现条件不符合就只能去睡觉并为下次唤醒再次设定时间，这是一个反复的过程。

这个问题，将引用JobScheduler来修复，它作为一个调度应用程序，负责当应用程序被唤醒时，提供适当的运行环境，所以开发者不用再让程序检测环境是否符合需求，开发人员只需要按照标准的流程来，调度程序会自动为唤醒的程序，准备好运行环境。应用程序可以使用这个调度程序，来唤醒他们，比如当设备连接到充电器后，调度程序将唤醒那些需要处理器工作的程序，让他们进行工作，或者在设备连接至WiFi网络的时候上传下载照片，更新内容等。该调度程序还支持一个时间窗口，以便它可以唤醒一组应用程序，这将使那些不需要精确唤醒时间，但每隔一两小时需要运行一次的程序能在同一时间点运行，这样就能让处理器保持更长时间的休眠。

JobScheduler的优势相当巨大，它不仅可以帮助手机节省电量， 实际由于不在需要监听，更改和设置报警，还可以帮助开发人员减少代码书写量。目前该JobScheduler类，已经加入Android L开发者预览版。

Android 5.0的省电模式是通过JobSchedulerAPI以及自动调节屏幕亮度、刷新率来达到省电的效果，而且还使用了Project Volta量化每个应用的耗电量，在手机快没电的情况下主动降低CPU的主频或者关闭通讯模块，以获得更长的待机时间。还配备了一个Battery Bistorian电池历史记录功能，可让详细显示设备的耗电情况。

参考资料

<http://gityuan.com/2016/01/10/power_rank/>

<http://hukai.me/android-performance-battery/>

<http://blog.csdn.net/matrix_xu/article/details/8509319>

<http://www.androidchina.net/2011.html>

<http://www.cnblogs.com/pasco/p/4106209.html>

<http://www.51testing.com/html/36/489136-817075.html>