Q博士的专栏
客户端三年，服务端1年，学习的路上

个人资料

DoctorQ

关注 发私信

访问： 1017138次
积分： 15090
等级：
排名： 第621名

原创： 479篇 转载： 1篇
译文： 9篇 评论： 512条

Android性能专项测试之耗电量统计API

2015-10-19 22:29 5853人阅读 评论(2) 收藏 举报

分类： 测试[Android性能]（14）

目录(?) [+]

参考文章:Android应用的耗电量统计
深入浅出Android App耗电量统计
Battery stats - CPU total vs CPU foreground
深入浅出 Android App 耗电量统计
浅析Wakelock机制与Android电源管理

耗电量API

Android 系统中很早就有耗电量的 API ,只不过一直都是隐藏的， Android 系统的 设置-电池功能 就是调用的这个 API ，该 API 的核心部分是调用了 com.android.internal.os.BatteryStatsHelper 类，利用 PowerProfile 类，读取 power_profile.xml 文件，我们一起来看看具体如何计算耗电量，首先从最新版本 6.0开始看

6.0的API

源码

BatteryStatsHelper

其中计算耗电量的方法为490行的 processAppUsage ,下来一步一步来解释该方法。

App耗电量的计算探究

```
1 private void processAppUsage(SparseArray<UserHandle> asUsers) {
```

方法的参数是一个 SparseArray 数组，存储的对象是 UserHandle ,官方文档给出的解释是，代表一个用户，可以理解为这个类里面存储了用户的相关信息.

```
1 final boolean forAllUsers = (asUsers.get(UserHandle.USER_ALL) != null);
```

然后判断该次计算是否针对所有用户，通过 UserHandle 的 USER_ALL 值来判断，该值为 -1 ,源码的地址在https://github.com/DoctorQ/platform_frameworks_base/blob/android-6.0.0_r1/core/java/android/os/UserHandle.java.

```
1 mStatsPeriod = mTypeBatteryRealtime;
```

然后给公共变量int类型的 mStatsPeriod 赋值,这个值 mTypeBatteryRealtime 的计算过程又在320行的 refreshStats 方法中:

```
1 mTypeBatteryRealtime = mStats.computeBatteryRealtime(rawRealtimeUs, mStatsType);
```

这里面用到了 BatteryStats(mStats) 类中的 computeBatteryRealtime 方法,该方法计算出此次统计电量的时间间隔。好，歪楼了，回到 BatteryStatsHelper 中。

```
1 BatterySipper osSipper = null;
2 final SparseArray<? extends Uid> uidStats = mStats.getUidStats();
3 final int NU = uidStats.size();
```

首先创建一个 BatterySipper 对象 osSipper ,该对象里面可以存储一些后续我们要计算的值，然后通过 BatteryStats 类对象 mStats 来得到一个包含 Uid 的对象的 SparseArray 组数,然后计算了一下这个数组的大小，保存在变量NU中。

```
for (int iu = 0; iu < NU; iu++) {
final Uid u = uidStats.valueAt(iu);
        final BatterySipper app = new BatterySipper(BatterySipper.DrainType.APP, u, 0);
```

然后 for 循环计算每个 Uid 代表的 App 的耗电量，因为 BatterySipper 可计算的类型有三种:应用, 系统服务, 硬件类型,所以这个地方传入的是 DrainType.APP ,

还有其他可选类型如下:

```
public enum DrainType {
    IDLE,
    CELL,
    PHONE,
    WIFI,
    BLUETOOTH,
    FLASHLIGHT,
    SCREEN,
    APP,
    USER,
    UNACCOUNTED,
    OVERCOUNTED,
    CAMERA
  }
```

列举了目前可计算耗电量的模块。

```
mCpuPowerCalculator.calculateApp(app, u, mRawRealtime, mRawUptime, mStatsType);
        mWakelockPowerCalculator.calculateApp(app, u, mRawRealtime, mRawUptime, mStatsType);
        mMobileRadioPowerCalculator.calculateApp(app, u, mRawRealtime, mRawUptime, mStatsType);
        mWifiPowerCalculator.calculateApp(app, u, mRawRealtime, mRawUptime, mStatsType);
        mBluetoothPowerCalculator.calculateApp(app, u, mRawRealtime, mRawUptime, mStatsType);
        mSensorPowerCalculator.calculateApp(app, u, mRawRealtime, mRawUptime, mStatsType);
        mCameraPowerCalculator.calculateApp(app, u, mRawRealtime, mRawUptime, mStatsType);
        mFlashlightPowerCalculator.calculateApp(app, u, mRawRealtime, mRawUptime, mStatsType);
```

其中 mStatsType 的值为 BatteryStats.STATS_SINCE_CHARGED ,代表了我们的计算规则是从上次充满电后数据，还有一种规则是 STATS_SINCE_UNPLUGGED 是拔掉USB线后的数据。而 mRawRealtime 是当前时间， mRawUptime 是运行时间。6.0的对各个模块的消耗都交给了单独的类去计算，这些类都继承于 PowerCalculator 抽象类：

```
蓝牙耗电:BluetoothPowerCalculator.java
摄像头耗电:CameraPowerCalculator.java
Cpu耗电:CpuPowerCalculator.java
手电筒耗电:FlashlightPowerCalculator.java
无线电耗电:MobileRadioPowerCalculator.java
传感器耗电:SensorPowerCalculator.java
Wakelock耗电:WakelockPowerCalculator.java
Wifi耗电:WifiPowerCalculator.java
```

这一部分我一会单独拿出来挨个解释，现在我们还是回到 BatteryStatsHelper 继续往下走

```
final double totalPower = app.sumPower();
```

BatterySipper#sumPower 方法是统计总耗电量,方法详情如下，其中 usagePowerMah 这个值有点特殊，其他的上面都讲过.

```
/**
    * Sum all the powers and store the value into `value`.
    * @return the sum of all the power in this BatterySipper.
    */
   public double sumPower() {
       return totalPowerMah = usagePowerMah + wifiPowerMah + gpsPowerMah + cpuPowerMah +
           sensorPowerMah + mobileRadioPowerMah + wakeLockPowerMah + cameraPowerMah +
           flashlightPowerMah;
   }
```

然后根据是否是DEBUG版本打印信息，这个没啥可说的，然后会把刚才计算的电量值添加到列表中:

```
// Add the app to the list if it is consuming power.
        if (totalPower != 0 || u.getUid() == 0) {
            //
            // Add the app to the app list, WiFi, Bluetooth, etc, or into "Other Users" list.
            //
            final int uid = app.getUid();
            final int userId = UserHandle.getUserId(uid);
            if (uid == Process.WIFI_UID) {
                mWifiSippers.add(app);
            } else if (uid == Process.BLUETOOTH_UID) {
                mBluetoothSippers.add(app);
            } else if (!forAllUsers && asUsers.get(userId) == null
                    && UserHandle.getAppId(uid) >= Process.FIRST_APPLICATION_UID) {
                // We are told to just report this user's apps as one large entry.
                List<BatterySipper> list = mUserSippers.get(userId);
                if (list == null) {
                    list = new ArrayList<>();
                    mUserSippers.put(userId, list);
                }
                list.add(app);
            } else {
```

关闭

```
            mUsageList.add(app);
        }

        if (uid == 0) {
            osSipper = app;
        }
    }
```

首先判断 totalPower 的值和当前 uid号 是否符合规则，规则为总耗电量不为0或者用户id为0.当 uid 表明为WIFI或者蓝牙时，添加到下面对应的列表中，一般情况下正常的应用我们直接保存到下面的 mUsageList 中就行就行，但是也有一些例外:

```
/**
 * List of apps using power.
 */
private final List<BatterySipper> mUsageList = new ArrayList<>();

/**
 * List of apps using wifi power.
 */
private final List<BatterySipper> mWifiSippers = new ArrayList<>();

/**
 * List of apps using bluetooth power.
 */
private final List<BatterySipper> mBluetoothSippers = new ArrayList<>();
```

如果我们的系统是单用户系统，且当前的 userId 号不在我们的统计范围内，且其进程 id 号是大于 Process.FIRST_APPLICATION_UID (10000,系统分配给普通应用的其实id号),我们就要将其存放到 mUserSippers 数组中，定义如下:

```
private final SparseArray<List<BatterySipper>> mUserSippers = new SparseArray<>();
```

最后判断 uid 为0的话，代表是 Android 操作系统的耗电量，赋值给 osSipper (494行定义)就可以了,这样一个 app 的计算就完成了，遍历部分就不说了,保存这个 osSipper 是为了最后一步计算:

```
if (osSipper != null) {
        // The device has probably been awake for longer than the screen on
        // time and application wake lock time would account for.  Assign
        // this remainder to the OS, if possible.
        mWakelockPowerCalculator.calculateRemaining(osSipper, mStats, mRawRealtime,
                                    mRawUptime, mStatsType);
        osSipper.sumPower();
    }
```

主流程我们已经介绍完了，下面来看各个子模块耗电量的计算

Cpu耗电量

CpuPowerCalculator.java

Cpu的计算要用到PowerProfile类,该类主要是解析power_profile.xml:

```
<device name="Android">
  <!-- Most values are the incremental current used by a feature,
       in mA (measured at nominal voltage).
       The default values are deliberately incorrect dummy values.
       OEM's must measure and provide actual values before
       shipping a device.
       Example real-world values are given in comments, but they
       are totally dependent on the platform and can vary
       significantly, so should be measured on the shipping platform
       with a power meter. -->
  <item name="none">0</item>
  <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="radio.active">0.1</item> <!-- ~200mA -->
  <item name="radio.scanning">0.1</item> <!-- cellular radio scanning for signal, ~10mA -->
  <item name="gps.on">0.1</item> <!-- ~50mA -->
  <!-- Current consumed by the radio at different signal strengths, when paging -->
  <array name="radio.on"> <!-- Strength 0 to BINS-1 -->
     <value>0.2</value> <!-- ~2mA -->
     <value>0.1</value> <!-- ~1mA -->
  </array>
  <!-- Different CPU speeds as reported in
       /sys/devices/system/cpu/cpu0/cpufreq/stats/time_in_state -->
  <array name="cpu.speeds">
```

关闭

```
    <value>400000</value> <!-- 400 MHz CPU speed -->
  </array>
  <!-- Current when CPU is idle -->
  <item name="cpu.idle">0.1</item>
  <!-- Current at each CPU speed, as per 'cpu.speeds' -->
  <array name="cpu.active">
    <value>0.1</value>  <!-- ~100mA -->
  </array>
  <!-- This is the battery capacity in mAh (measured at nominal voltage) -->
  <item name="battery.capacity">1000</item>

  <array name="wifi.batchedscan"> <!-- mA -->
    <value>.0002</value> <!-- 1-8/hr -->
    <value>.002</value>  <!-- 9-64/hr -->
    <value>.02</value>   <!-- 65-512/hr -->
    <value>.2</value>    <!-- 513-4,096/hr -->
    <value>2</value>    <!-- 4097-/hr -->
  </array>
</device>
```

这个里面存储了Cpu(cpu.speeds)的主频等级，以及每个主频每秒消耗的毫安(cpu.active)，好，现在回到 CpuPowerCalculator 中，先来看构造方法

```
public CpuPowerCalculator(PowerProfile profile) {
    final int speedSteps = profile.getNumSpeedSteps();
    mPowerCpuNormal = new double[speedSteps];
    mSpeedStepTimes = new long[speedSteps];
    for (int p = 0; p < speedSteps; p++) {
        mPowerCpuNormal[p] = profile.getAveragePower(PowerProfile.POWER_CPU_ACTIVE, p);
    }
}
```

第一步获得 Cpu 有几个主频等级，因为不同等级消耗的电量不一样，所以要区别对待，根据主频的个数，然后初始化 mPowerCpuNormal 和 mSpeedStepTimes ,前者用来保存不同等级的耗电速度，后者用来保存在不同等级上耗时，然后给 mPowerCpuNormal 的每个元素附上值。构造方法就完成了其所有的工作，现在来计算方法 calculateApp，

```
final int speedSteps = mSpeedStepTimes.length;

    long totalTimeAtSpeeds = 0;
    for (int step = 0; step < speedSteps; step++) {
        mSpeedStepTimes[step] = u.getTimeAtCpuSpeed(step, statsType);
        totalTimeAtSpeeds += mSpeedStepTimes[step];
    }
    totalTimeAtSpeeds = Math.max(totalTimeAtSpeeds, 1);
```

首先得到 Cpu 主频等级个数，然后 BatteryStats.Uid 得到不同主频上执行时间，计算 Cpu 总耗时保存在 totalTimeAtSpeeds 中，

```
app.cpuTimeMs = (u.getUserCpuTimeUs(statsType) + u.getSystemCpuTimeUs(statsType)) / 1000;
```

Cpu 的执行时间分很多部分，但是我们关注 User 和 Kernal 部分,也就是上面的 UserCpuTime 和 SystemCpuTime 。

```
double cpuPowerMaMs = 0;
    for (int step = 0; step < speedSteps; step++) {
        final double ratio = (double) mSpeedStepTimes[step] / totalTimeAtSpeeds;
        final double cpuSpeedStepPower = ratio * app.cpuTimeMs * mPowerCpuNormal[step];
        if (DEBUG && ratio != 0) {
            Log.d(TAG, "UID " + u.getUid() + ": CPU step #"
                    + step + " ratio=" + BatteryStatsHelper.makemAh(ratio) + " power="
                    + BatteryStatsHelper.makemAh(cpuSpeedStepPower / (60 * 60 * 1000)));
        }
        cpuPowerMaMs += cpuSpeedStepPower;
    }
```

上面的代码就是将不同主频的消耗累加到一起，但是其中值得注意的是，他并不是用各个主频的消耗时间*主频单位时间内消耗的电量，而是用一个radio变量来计算得到各个主频段执行时间占总时间的百分比，然后用 cpuTimeMs 来换算成各个主频的Cpu实际消耗时间，这比5.0的API多了这么一步，我估计是发现了计算的不严谨性，这也是 Android 迟迟不放出统计电量方式的原因，其实google自己对这块也没有把握，所以才会造成不同 API 计算方式的差异。好，计算完我们的总消耗后，是不是就算完事了？如果你只需要得到一个App的耗电总量，上面的讲解已经足够了，但是6.0的API计算了每个App的不同进程的耗电量，这个我们就只当看看就行，暂时没什么实际意义。

```
// Keep track of the package with highest drain.
    double highestDrain = 0;

    app.cpuFgTimeMs = 0;
    final ArrayMap<String, ? extends BatteryStats.Uid.Proc> processStats = u.getProcessStats();
    final int processStatsCount = processStats.size();
    for (int i = 0; i < processStatsCount; i++) {
        final BatteryStats.Uid.Proc ps = processStats.valueAt(i);
        final String processName = processStats.keyAt(i);
        app.cpuFgTimeMs += ps.getForegroundTime(statsType);

        final long costValue = ps.getUserTime(statsType) + ps.getSystemTime(statsType)
                + ps.getForegroundTime(statsType);

        // Each App can have multiple packages and with multiple running processes.
        // Keep track of the package who's process has the highest drain.
```

关闭

```
            if (app.packageWithHighestDrain == null ||
                    app.packageWithHighestDrain.startsWith("*")) {
                highestDrain = costValue;
                app.packageWithHighestDrain = processName;
            } else if (highestDrain < costValue && !processName.startsWith("*")) {
                highestDrain = costValue;
                app.packageWithHighestDrain = processName;
            }
        }

        // Ensure that the CPU times make sense.
        if (app.cpuFgTimeMs > app.cpuTimeMs) {
            if (DEBUG && app.cpuFgTimeMs > app.cpuTimeMs + 10000) {
                Log.d(TAG, "WARNING! Cputime is more than 10 seconds behind Foreground time");
            }

            // Statistics may not have been gathered yet.
            app.cpuTimeMs = app.cpuFgTimeMs;
        }
```

上面统计同一 App 下不同的进程的耗电量，得到消耗最大的进程名，保存到 BatterySipper 对象中,然后得出 App 的 Cpu 的 foreground 消耗时间,将 foreground 时间与之前计算得到的 cpuTimeMs 进行比较，如果 foreground 时间比 cpuTimeMs 还要大，那么就将 cpuTimeMs 的时间改变为值，但是这个值的变化对之前耗电总量的计算没有丝毫影响。

```
// Convert the CPU power to mAh
        app.cpuPowerMah = cpuPowerMaMs / (60 * 60 * 1000);
```

最后的最后，将耗电量用mAh单位来表示，所以在毫秒的基础上除以 60*60*1000 。

总结: Cpu 耗电量的计算是要区分不同主频的，频率不同，单位时间内消耗的电量是有区分的，这一点要明白。还有一点就是不同主频上的执行时间不是通过 BatteryStats.Uid#getTimeAtCpuSpeed 方法得到的，二十是通过百分比和 BatteryStats.Uid#getUserCpuTimeUs 和 getSystemCpuTimeUs 计算得到 cpuTimeMs 乘积得到的。最后一点就是， cpuTimeMs 时间是会在计算完毕后进行比较，比较的对象是 CPU 的 foreground 时间。

WakeLock耗电量的计算

WakelockPowerCalculator.java

从构造方法开始，

```
public WakelockPowerCalculator(PowerProfile profile) {
        mPowerWakelock = profile.getAveragePower(PowerProfile.POWER_CPU_AWAKE);
    }
```

首先得到 power_profile.xml 中 cpu.awake 表示的值，保存在 mPowerWakelock 变量中。构造方法只做了这么点事，下面进入 calculateApp 方法。

```
@Override
    public void calculateApp(BatterySipper app, BatteryStats.Uid u, long rawRealtimeUs,
                             long rawUptimeUs, int statsType) {
        long wakeLockTimeUs = 0;
        final ArrayMap<String, ? extends BatteryStats.Uid.Wakelock> wakelockStats =
                u.getWakelockStats();
        final int wakelockStatsCount = wakelockStats.size();
        for (int i = 0; i < wakelockStatsCount; i++) {
            final BatteryStats.Uid.Wakelock wakelock = wakelockStats.valueAt(i);

            // Only care about partial wake locks since full wake locks
            // are canceled when the user turns the screen off.
            BatteryStats.Timer timer = wakelock.getWakeTime(BatteryStats.WAKE_TYPE_PARTIAL);
            if (timer != null) {
                wakeLockTimeUs += timer.getTotalTimeLocked(rawRealtimeUs, statsType);
            }
        }
        app.wakeLockTimeMs = wakeLockTimeUs / 1000; // convert to millis
        mTotalAppWakelockTimeMs += app.wakeLockTimeMs;

        // Add cost of holding a wake lock.
        app.wakeLockPowerMah = (app.wakeLockTimeMs * mPowerWakelock) / (1000*60*60);
        if (DEBUG && app.wakeLockPowerMah != 0) {
            Log.d(TAG, "UID " + u.getUid() + ": wake " + app.wakeLockTimeMs
                    + " power=" + BatteryStatsHelper.makemAh(app.wakeLockPowerMah));
        }
    }
```

关闭

首先获得 Wakelock 的数量，然后逐个遍历得到每个 Wakelock 对象，得到该对象后，得到 BatteryStats.WAKE_TYPE_PARTIAL 的唤醒时间，然后累加，其实 wakelock 有4种，为什么只取 partial 的时间,具体代码 google 也没解释的很清楚,只是用一句注释打发了我们。得到总时间后，就可以与构造方法中的单位时间 waklock 消耗电量相乘得到 Wakelock 消耗的总电量。

Wifi耗电量的计算

首先来看构造方法，来了解一下WIFI的耗电量计算用到了 power_profile.xml 中的哪些属性:

```
public WifiPowerCalculator(PowerProfile profile) {
```

```
    mIdleCurrentMa = profile.getAveragePower(PowerProfile.POWER_WIFI_CONTROLLER_IDLE);
    mTxCurrentMa = profile.getAveragePower(PowerProfile.POWER_WIFI_CONTROLLER_TX);
    mRxCurrentMa = profile.getAveragePower(PowerProfile.POWER_WIFI_CONTROLLER_RX);
  }
```

我们去 PowerProfile.java 找到上面三个常量代表的属性:

```
public static final String POWER_WIFI_CONTROLLER_IDLE = "wifi.controller.idle";
  public static final String POWER_WIFI_CONTROLLER_RX = "wifi.controller.rx";
  public static final String POWER_WIFI_CONTROLLER_TX = "wifi.controller.tx";
```

知道对应的xml的属性后我们直接看 calculateApp 方法:

```
@Override
  public void calculateApp(BatterySipper app, BatteryStats.Uid u, long rawRealtimeUs,
                  long rawUptimeUs, int statsType) {
    final long idleTime = u.getWifiControllerActivity(BatteryStats.CONTROLLER_IDLE_TIME,
        statsType);
    final long txTime = u.getWifiControllerActivity(BatteryStats.CONTROLLER_TX_TIME, statsType);
    final long rxTime = u.getWifiControllerActivity(BatteryStats.CONTROLLER_RX_TIME, statsType);
    app.wifiRunningTimeMs = idleTime + rxTime + txTime;
    app.wifiPowerMah =
        ((idleTime * mIdleCurrentMa) + (txTime * mTxCurrentMa) + (rxTime * mRxCurrentMa))
        / (1000*60*60);
    mTotalAppPowerDrain += app.wifiPowerMah;

    app.wifiRxPackets = u.getNetworkActivityPackets(BatteryStats.NETWORK_WIFI_RX_DATA,
        statsType);
    app.wifiTxPackets = u.getNetworkActivityPackets(BatteryStats.NETWORK_WIFI_TX_DATA,
        statsType);
    app.wifiRxBytes = u.getNetworkActivityBytes(BatteryStats.NETWORK_WIFI_RX_DATA,
        statsType);
    app.wifiTxBytes = u.getNetworkActivityBytes(BatteryStats.NETWORK_WIFI_TX_DATA,
        statsType);

    if (DEBUG && app.wifiPowerMah != 0) {
      Log.d(TAG, "UID " + u.getUid() + ": idle=" + idleTime + "ms rx=" + rxTime + "ms tx=" +
          txTime + "ms power=" + BatteryStatsHelper.makemAh(app.wifiPowerMah));
    }
  }
```

这里的计算方式也是差不多，先根据Uid得到时间，然后乘以构造方法里对应的wifi类型单位时间内消耗电量值，没什么难点，就不一一分析，需要注意的是，这里面还计算了 wifi 传输的数据包的数量和字节数。

蓝牙耗电量的计算

蓝牙关注的 power_profile.xml 中的属性如下:

```
public static final String POWER_BLUETOOTH_CONTROLLER_IDLE = "bluetooth.controller.idle";
  public static final String POWER_BLUETOOTH_CONTROLLER_RX = "bluetooth.controller.rx";
  public static final String POWER_BLUETOOTH_CONTROLLER_TX = "bluetooth.controller.tx";
```

但是还没有单独为App计算耗电量的，所以这个地方是空的。

```
@Override
  public void calculateApp(BatterySipper app, BatteryStats.Uid u, long rawRealtimeUs,
                  long rawUptimeUs, int statsType) {
    // No per-app distribution yet.
  }
```

摄像头耗电量的计算

CameraPowerCalculator.java

摄像头的耗电量关注的是 power_profile.xml 中 camera.avg 属性代表的值，保存到 mCameraPowerOnAvg，

```
public static final String POWER_CAMERA = "camera.avg";
```

计算方式如下:

```
@Override
  public void calculateApp(BatterySipper app, BatteryStats.Uid u, long rawRealtimeUs,
                  long rawUptimeUs, int statsType) {

    // Calculate camera power usage.  Right now, this is a (very) rough estimate based on the
    // average power usage for a typical camera application.
    final BatteryStats.Timer timer = u.getCameraTurnedOnTimer();
    if (timer != null) {
      final long totalTime = timer.getTotalTimeLocked(rawRealtimeUs, statsType) / 1000;
      app.cameraTimeMs = totalTime;
      app.cameraPowerMah = (totalTime * mCameraPowerOnAvg) / (1000*60*60);
    } else {
      app.cameraTimeMs = 0;
      app.cameraPowerMah = 0;
    }
```

关闭

```
16    }
```

先计算摄像头打开的时间 totalTime ，然后根据这个值乘以 mCameraPowerOnAvg 得到摄像头的耗电量。

手电筒耗电量的计算

FlashlightPowerCalculator.java

```
1  public static final String POWER_FLASHLIGHT = "camera.flashlight";
```

跟摄像头类似，也是先得到时间，然后乘积，不想说了，没意思。

无线电耗电量的计算

MobileRadioPowerCalculator.java

关注的是 power_profile.xml 中如下三个属性:

```
1  /**
2      * Power consumption when screen is on, not including the backlight power.
3      */
4     public static final String POWER_SCREEN_ON = "screen.on";
5
6     /**
7      * Power consumption when cell radio is on but not on a call.
8      */
9     public static final String POWER_RADIO_ON = "radio.on";
10
11    /**
12     * Power consumption when cell radio is hunting for a signal.
13     */
14    public static final String POWER_RADIO_SCANNING = "radio.scanning";
```

当无限量连接上时，根据信号强度不同，耗电量的计算是有区别的，所以在构造方法，当无线电的状态为on时，是要特殊处理的，其他两个状态(active和scan)就正常取值就可以了。

```
1  /**
2      * Power consumption when screen is on, not including the backlight power.
3      */
4     public static final String POWER_SCREEN_ON = "screen.on";
5
6     /**
7      * Power consumption when cell radio is on but not on a call.
8      */
9     public static final String POWER_RADIO_ON = "radio.on";
10
11    /**
12     * Power consumption when cell radio is hunting for a signal.
13     */
14    public static final String POWER_RADIO_SCANNING = "radio.scanning";
```

计算的方式分两种，以无线电处于 active 状态的次数为区分，当 active 大于0，我们用处于 active 状态的时间来乘以它的单位耗时。另一种情况就要根据网络转化的数据包来计算耗电量了。

传感器耗电量的计算

SensorPowerCalculator.java

只关注一个属性:

```
1  public static final String POWER_GPS_ON = "gps.on";
```

计算方式如下:

```
1  @Override
2     public void calculateApp(BatterySipper app, BatteryStats.Uid u, long rawRealtimeUs,
3                              long rawUptimeUs, int statsType) {
4        // Process Sensor usage
5        final SparseArray<? extends BatteryStats.Uid.Sensor> sensorStats = u.getSensorStats();
6        final int NSE = sensorStats.size();
7        for (int ise = 0; ise < NSE; ise++) {
8           final BatteryStats.Uid.Sensor sensor = sensorStats.valueAt(ise);
9           final int sensorHandle = sensorStats.keyAt(ise);
10          final BatteryStats.Timer timer = sensor.getSensorTime();
11          final long sensorTime = timer.getTotalTimeLocked(rawRealtimeUs, statsType) / 1000;
12          switch (sensorHandle) {
13             case BatteryStats.Uid.Sensor.GPS:
14                app.gpsTimeMs = sensorTime;
15                app.gpsPowerMah = (app.gpsTimeMs * mGpsPowerOn) / (1000*60*60);
16                break;
17             default:
18                final int sensorsCount = mSensors.size();
19                for (int i = 0; i < sensorsCount; i++) {
```

关闭

```
                final Sensor s = mSensors.get(i);
                if (s.getHandle() == sensorHandle) {
                    app.sensorPowerMah += (sensorTime * s.getPower()) / (1000*60*60);
                    break;
                }
            }
            break;
        }
    }
}
```

当传感器的类型为GPS时，我们计算每个传感器的时间然后乘以耗电量，和所有的耗电量计算都是一样，不同的是，当传感器不是GPS时，这个时候计算就根据 SensorManager 得到所有传感器类型，这个里面保存有不同传感器的单位耗电量，这样就能计算不同传感器的耗电量。

总结

至此我已经把App耗电量的计算讲完了(还有硬件)，前后花费3天时间，好痛苦(此处一万只草泥马)，不过好在自己也算对这个耗电量的理解有了一定的认识。google官方对耗电量的统计给出的解释都是不能代表真实数据，只能作为参考值，因为受power_profile.xml的干扰太大，如果手机厂商没有严格设置这个文件，那可想而知出来的值可能是不合理的。

提示

腾讯的GT团队前几天推出了耗电量的计算APK，原理是一样的，大家可以试用下GT

- 上一篇 Android性能专项测试之Batterystats
- 下一篇 Android静态代码检查-Lint

我的同类文章

测试[Android性能]（14）

- 心向百度 2016-04-12 阅读 3909
- Android性能专项测试之Batterystats 2015-10-14 阅读 7312
- Android性能专项测试之GPU Monitor 2015-10-09 阅读 5320
- Android性能专项测试之MAT 2015-10-05 阅读 2289
- Android性能专项测试之Heap Snapshot... 2015-10-09 阅读 4432
- Android内存泄漏检测-LeakCanary 2015-10-23 阅读 2915
- Android性能专项测试之Network monitor 2015-10-09 阅读 3192
- Android性能专项测试之Systrace工具 2015-10-08 阅读 4598
- Android性能专项测试之TraceView工具(... 2015-09-30 阅读 2810
- Android性能专项测试之Allocation Trac... 2015-09-26 阅读 5334

更多文章

参考知识库

猜你在找

- Android开发高级组件与框架—...
- Android之数据存储
- 移动手机APP测试从零开始（初...
- Android中的数据存储
- Android核心技术——Android...
- MIG专项测试组实战分析内存突...
- 升级Xcode到61之后使用iPhon...
- 专项测试三-如何分析内存溢出问...
- 专项测试一兼容测试1-app兼容...
- Android-NDK错误 undefined ...

查看评论


yuger 1楼 2015-10-17 11:01发表

昨天晚上我Nexus5推送的Marshmallow。

Re: 2015-10-19 12:01发表

回复WXY9206：那你可以试试adb shell dumpsys batterystats来获取耗电量了。原理一样

发表评论

用 户 名： haijunz

评论内容：

提交

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack VPN Spark ERP
IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery BI HTML5 Spring Apache .NET
HTML SDK IIS Fedora XML LBS Unity Splashtop UML components Windows Mobile Rails QEMU KDE
Cassandra CloudStack FTC coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo
Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase Pure Solr Angular Cloud Foundry
Redis Scala Django Bootstrap

关闭