内容目录

[1. BatteryStatsService构造函数 1](#__RefHeading___Toc1319_181201217)

[1.1 UserInfoProvider 1](#__RefHeading___Toc1321_181201217)

[1.2 BatteryExternalStatsWorker implements BatteryStatsImpl.ExternalStatsSync 1](#__RefHeading___Toc1323_181201217)

[1.2.1 BatteryExternalStatsWorker 2](#__RefHeading___Toc1325_181201217)

[1.2.2 Worker线程 2](#__RefHeading___Toc1327_181201217)

[1.2.2.1 mExecutorService执行任务相关接口 2](#__RefHeading___Toc1435_181201217)

[1.2.2.2调用mExecutorService执行任务相关接口 3](#__RefHeading___Toc1437_181201217)

[1.2.2.3 mSyncTask任务 3](#__RefHeading___Toc1439_181201217)

[1.2.3其他接口 5](#__RefHeading___Toc1329_181201217)

BatteryStasService的主要功能是收集系统中各模块和应用进程的用电情况。因此，我们可以认为BatteryStatsService是Android中的“电表”。

只不过这个电表比较智能，不是单纯地统计整体的耗电，而是分门别类的统计每个部分的耗电情况。接下来我们就分析一下BatteryStatsService的主要流程。为了方便叙述，后文中我们将BatteryStatsService简称为BSS。我们这次的参考代码为Android P(2018-9-14)，相关文件代码位置如下，后续本文不再列出文件路径

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

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

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

~/frameworks/base/services/core/java/com/android/server/am/BatteryExternalStatsWorker.java

# 1. BatteryStatsService构造函数

与一般的系统服务不太一样，BSS的创建和发布是在ActivityManagerService中进行的，相关代码如下：

public ActivityManagerService(Context systemContext) {

...

File dataDir = Environment.getDataDirectory();

File systemDir = new File(dataDir, "system");

systemDir.mkdirs();

…

//创建BSS对象，传入/data/system目录，同时传入ActivityManagerService的handler

mBatteryStatsService = new BatteryStatsService(systemContext, systemDir, mHandler);

//调用BSS中BatteryStatsImpl对象的readLocked方法

mBatteryStatsService.getActiveStatistics().readLocked();

//将初始化得到的信息写入disk

mBatteryStatsService.scheduleWriteToDisk();

...

}

接下来我们先看看BatteryStatsService的构造函数：

private final BatteryExternalStatsWorker mWorker;//由 mWorker 来串行执行相关电量统计任务(单线程)

private final Context mContext;

private final BatteryStatsImpl mStats;

BatteryStatsService(Context context, File systemDir, Handler handler) {//传递的参数：文件路径/data/system，AMS MainHandler

//BatteryStatsImpl expects the ActivityManagerService handler, so pass that one through.

mContext = context;

mUserManagerUserInfoProvider = new BatteryStatsImpl.UserInfoProvider() {//见1.1分析

private UserManagerInternal umi;

@Override

public int[] getUserIds() {

if (umi == null) {

umi = LocalServices.getService(UserManagerInternal.class);

}

return (umi != null) ? umi.getUserIds() : null;

}

};

mStats = new BatteryStatsImpl(systemDir, handler, this, mUserManagerUserInfoProvider);//创建BatteryStatsImpl实例

//8.1之前的是创建一个新的线程batterystats-sync用来记录电池电量信息

//9.0中是利用Executors来顺序执行任务(线程)，见1.2分析

mWorker = new BatteryExternalStatsWorker(context, mStats);

mStats.setExternalStatsSyncLocked(mWorker);//设置到 mStats里

mStats.setRadioScanningTimeoutLocked(mContext.getResources().getInteger(com.android.internal.R.integer.config\_radioScanningTimeout) \* 1000L);//见1.3分析StopwatchTimer

mStats.setPowerProfileLocked(new PowerProfile(context));

}

## 1.1 UserInfoProvider

BatteryStatsImpl的内部类UserInfoProvider，从 Android 5.0 开始，多用户功能默认处于停用状态

public static abstract class UserInfoProvider {

private int[] userIds;

protected abstract @Nullable int[] getUserIds();

@VisibleForTesting

public final void refreshUserIds() {

userIds = getUserIds();

}

@VisibleForTesting

public boolean exists(int userId) {//判断userId在userIds 中是否存在

return userIds != null ? ArrayUtils.contains(userIds, userId) : true;

}

}

## **1.2** **BatteryExternalStatsWorker implements BatteryStatsImpl.ExternalStatsSync**

BatteryExternalStatsWorker实现了BatteryStatsImpl.ExternalStatsSync接口，一共604行，一个worker在专用的线程上从外部源(WiFi controller、BT chipset等)获得数据并且调用通过BatteryStatsImpl方法来更新BatteryStatsImpl。

尽可能的不通过持有BatteryStatsImpl lock来进行工作，并且在持有BatteryStatsImpl lock情况下，将有用的数据推入BatteryStatsImpl中。

class BatteryExternalStatsWorker implements BatteryStatsImpl.ExternalStatsSync

我们先看下ExternalStatsSync

public interface ExternalStatsSync {

int UPDATE\_CPU = 0x01;

int UPDATE\_WIFI = 0x02;

int UPDATE\_RADIO = 0x04;

int UPDATE\_BT = 0x08;

int UPDATE\_RPM = 0x10; // 16，Resource Power Manager对应subsystem，比如高通的各个子系统，不过目前看来这块底层实现，高通至今未对接上

int UPDATE\_ALL = UPDATE\_CPU | UPDATE\_WIFI | UPDATE\_RADIO | UPDATE\_BT | UPDATE\_RPM;

Future<?> scheduleSync(String reason, int flags);

Future<?> scheduleCpuSyncDueToRemovedUid(int uid);

Future<?> scheduleReadProcStateCpuTimes(boolean onBattery, boolean onBatteryScreenOff, long delayMillis);

Future<?> scheduleCopyFromAllUidsCpuTimes(boolean onBattery, boolean onBatteryScreenOff);

Future<?> scheduleCpuSyncDueToSettingChange();

Future<?> scheduleCpuSyncDueToScreenStateChange(boolean onBattery, boolean onBatteryScreenOff);

Future<?> scheduleCpuSyncDueToWakelockChange(long delayMillis);

void cancelCpuSyncDueToWakelockChange();

Future<?> scheduleSyncDueToBatteryLevelChange(long delayMillis);

}

上面的方法都需要BatteryExternalStatsWorker进行实现

### 1.2.1 BatteryExternalStatsWorker

把 BatteryStatsImpl作为参数传递了进来

BatteryExternalStatsWorker(Context context, BatteryStatsImpl stats) {

mContext = context;

mStats = stats;

}

### 1.2.2 Worker线程

使用了 Executors的 newSingleThreadScheduledExecutor线程池，然后有各种任务往这个线程池里面丢，其实这些任务主要需要mStats接口来完成

@GuardedBy("this")

private int mUpdateFlags = 0;//对应 ExternalStatsSync UPDATE\_XXX

private Future<?> mCurrentFuture = null;//对应mSyncTask

private String mCurrentReason = null;//执行该任务的reason XXX，对应方法的DueToXXX

private boolean mOnBattery;//是否充电

private boolean mOnBatteryScreenOff;//充电是否亮屏

private boolean mUseLatestStates = true;//最近使用状态？

private final IntArray mUidsToRemove = new IntArray();//UIDs

private Future<?> mWakelockChangesUpdate;//对应wakelock-change任务

private Future<?> mBatteryLevelSync;//对应battery-level任务

private final Object mWorkerLock = new Object();

@GuardedBy("mWorkerLock")

private IWifiManager mWifiManager = null;

@GuardedBy("mWorkerLock")

private TelephonyManager mTelephony = null;

@GuardedBy("mWorkerLock")

//不像蓝牙 ，WiFi要保持统计的累计总量，保留最后的WiFi统计数据，这样我们就可以计算一个增量

private WifiActivityEnergyInfo mLastInfo = new WifiActivityEnergyInfo(0, 0, 0, new long[]{0}, 0, 0, 0, 0);

private long mLastCollectionTimeStamp;//执行mSyncTask记录一次时间戳

private final ScheduledExecutorService mExecutorService =

Executors.newSingleThreadScheduledExecutor(

(ThreadFactory) r -> {

Thread t = new Thread(r, "batterystats-worker");

t.setPriority(Thread.NORM\_PRIORITY);

return t;

});

#### **1.2.2.1 mExecutorService执行任务相关接口**

(1) 直接执行 mSyncTask任务

@GuardedBy("this")

private Future<?> scheduleSyncLocked(String reason, int flags) {

if (mExecutorService.isShutdown()) {

return CompletableFuture.failedFuture(new IllegalStateException("worker shutdown"));//意外executor shutdown

}

if (mCurrentFuture == null) {

mUpdateFlags = flags;

mCurrentReason = reason;

mCurrentFuture = mExecutorService.submit(mSyncTask);//执行 mSyncTask任务

}

mUpdateFlags |= flags;

return mCurrentFuture;

}

(2) 延迟 delayMillis毫秒，执行传入的 syncRunnable任务

@GuardedBy("this")

private Future<?> scheduleDelayedSyncLocked(Future<?> lastScheduledSync, Runnable syncRunnable, long delayMillis) {

if (mExecutorService.isShutdown()) {

return CompletableFuture.failedFuture(new IllegalStateException("worker shutdown"));

}

if (lastScheduledSync != null) {

if (delayMillis == 0) {

lastScheduledSync.cancel(false);//false: 如果该任务已经执行中，则不取消

} else {

return lastScheduledSync;

}

}

return mExecutorService.schedule(syncRunnable, delayMillis, TimeUnit.MILLISECONDS);

}

#### **1.2.2.2调用mExecutorService执行任务相关接口**

(1) scheduleSync(String reason, int flags)

@Override

public synchronized Future<?> scheduleSync(String reason, int flags) {

return scheduleSyncLocked(reason, flags);

}

(2) scheduleCpuSyncDueToRemovedUid(int uid)

@Override

public synchronized Future<?> scheduleCpuSyncDueToRemovedUid(int uid) {

mUidsToRemove.add(uid);

return scheduleSyncLocked("remove-uid", UPDATE\_CPU);

}

(3) scheduleCpuSyncDueToSettingChange

@Override

public synchronized Future<?> scheduleCpuSyncDueToSettingChange() {

return scheduleSyncLocked("setting-change", UPDATE\_CPU);

}

(4) scheduleCpuSyncDueToScreenStateChange

@Override

public Future<?> scheduleCpuSyncDueToScreenStateChange(boolean onBattery, boolean onBatteryScreenOff) {

synchronized (BatteryExternalStatsWorker.this) {

if (mCurrentFuture == null || (mUpdateFlags & UPDATE\_CPU) == 0) {

mOnBattery = onBattery;

mOnBatteryScreenOff = onBatteryScreenOff;

mUseLatestStates = false;

}

return scheduleSyncLocked("screen-state", UPDATE\_CPU);

}

}

(5) scheduleWrite

public synchronized Future<?> scheduleWrite() {

if (mExecutorService.isShutdown()) {

return CompletableFuture.failedFuture(new IllegalStateException("worker shutdown"));

}

scheduleSyncLocked("write", UPDATE\_ALL);

return mExecutorService.submit(mWriteTask);

}

private final Runnable mWriteTask = new Runnable() {

@Override

public void run() {

synchronized (mStats) {

mStats.writeAsyncLocked();//更新到Disk

}

}

};

(6) scheduleCpuSyncDueToWakelockChange

public Future<?> scheduleCpuSyncDueToWakelockChange(long delayMillis) {

synchronized (BatteryExternalStatsWorker.this) {

mWakelockChangesUpdate = scheduleDelayedSyncLocked(mWakelockChangesUpdate,

() -> {//顺序执行了两个任务

scheduleSync("wakelock-change", UPDATE\_CPU);

scheduleRunnable(() -> mStats.postBatteryNeedsCpuUpdateMsg());//mStats中发送MSG\_REPORT\_CPU\_UPDATE\_NEEDED

},

delayMillis);

return mWakelockChangesUpdate;

}

}

(7) scheduleSyncDueToBatteryLevelChange

@Override

public Future<?> scheduleSyncDueToBatteryLevelChange(long delayMillis) {//执行battery-level change任务

synchronized (BatteryExternalStatsWorker.this) {

mBatteryLevelSync = scheduleDelayedSyncLocked(mBatteryLevelSync,

() -> scheduleSync("battery-level", UPDATE\_ALL),//用 Lambda 表达式表示Runnable()参数

delayMillis);

return mBatteryLevelSync;

}

}

取消相关任务的接口如下，如果任务未在执行中则取消，具体代码原文略

public void cancelCpuSyncDueToWakelockChange()

private void cancelSyncDueToBatteryLevelChangeLocked()

#### **1.2.2.3 mSyncTask任务**

这是主任务，由scheduleSyncLocked 接口调用进入

private final Runnable mSyncTask = new Runnable() {

@Override

public void run() {

// Capture a snapshot of the state we are meant to process.

final int updateFlags;

final String reason;

final int[] uidsToRemove;

final boolean onBattery;

final boolean onBatteryScreenOff;

final boolean useLatestStates;

synchronized (BatteryExternalStatsWorker.this) {

updateFlags = mUpdateFlags;

reason = mCurrentReason;

uidsToRemove = mUidsToRemove.size() > 0 ? mUidsToRemove.toArray() : EmptyArray.INT;

onBattery = mOnBattery;

onBatteryScreenOff = mOnBatteryScreenOff;

useLatestStates = mUseLatestStates;

mUpdateFlags = 0;

mCurrentReason = null;

mUidsToRemove.clear();

mCurrentFuture = null;

mUseLatestStates = true;

if ((updateFlags & UPDATE\_ALL) != 0) {

cancelSyncDueToBatteryLevelChangeLocked();

}

if ((updateFlags & UPDATE\_CPU) != 0) {

cancelCpuSyncDueToWakelockChange();

}

}

try {

synchronized (mWorkerLock) {

if (DEBUG) {

Slog.d(TAG, "begin updateExternalStatsSync reason=" + reason);

}

try {

updateExternalStatsLocked(reason, updateFlags, onBattery, onBatteryScreenOff, useLatestStates);

} finally {

if (DEBUG) {

Slog.d(TAG, "end updateExternalStatsSync");

}

}

}

if ((updateFlags & UPDATE\_CPU) != 0) {

mStats.copyFromAllUidsCpuTimes();//copy所有uids的cpu time快照

}

// Clean up any UIDs if necessary. 电量统计中，移除相应uid

synchronized (mStats) {

for (int uid : uidsToRemove) {

mStats.removeIsolatedUidLocked(uid);

}

mStats.clearPendingRemovedUids();

}

} catch (Exception e) {

Slog.wtf(TAG, "Error updating external stats: ", e);

}

synchronized (BatteryExternalStatsWorker.this) {

mLastCollectionTimeStamp = SystemClock.elapsedRealtime();//主任务执行完毕，更新本次时间戳

}

}

};

@GuardedBy("mWorkerLock")

private void updateExternalStatsLocked(final String reason, int updateFlags, boolean onBattery, boolean onBatteryScreenOff, boolean useLatestStates) {

// We will request data from external processes asynchronously, and wait on a timeout.

SynchronousResultReceiver wifiReceiver = null;

SynchronousResultReceiver bluetoothReceiver = null;

SynchronousResultReceiver modemReceiver = null;

if ((updateFlags & BatteryStatsImpl.ExternalStatsSync.UPDATE\_WIFI) != 0) {

// We were asked to fetch WiFi data.

if (mWifiManager == null) {

mWifiManager = IWifiManager.Stub.asInterface(ServiceManager.getService(Context.WIFI\_SERVICE));

}

if (mWifiManager != null) {

try {

wifiReceiver = new SynchronousResultReceiver("wifi");

mWifiManager.requestActivityInfo(wifiReceiver);//获取WiFi的信息，传入SynchronousResultReceiver对象

} catch (RemoteException e) {

// Oh well.

}

}

}

if ((updateFlags & BatteryStatsImpl.ExternalStatsSync.UPDATE\_BT) != 0) {

// We were asked to fetch Bluetooth data.

final BluetoothAdapter adapter = BluetoothAdapter.getDefaultAdapter();

if (adapter != null) {

bluetoothReceiver = new SynchronousResultReceiver("bluetooth");//获取蓝牙的信息，传入SynchronousResultReceiver对象

adapter.requestControllerActivityEnergyInfo(bluetoothReceiver);

}

}

if ((updateFlags & BatteryStatsImpl.ExternalStatsSync.UPDATE\_RADIO) != 0) {

// We were asked to fetch Telephony data.

if (mTelephony == null) {

mTelephony = TelephonyManager.from(mContext);

}

//获取modem的信息，传入SynchronousResultReceiver对象，对于Telephony而言，就是通过RIL发送消息给modem，获取返回结果

if (mTelephony != null) {

modemReceiver = new SynchronousResultReceiver("telephony");

mTelephony.requestModemActivityInfo(modemReceiver);

}

}

//awaitControllerInfo内部调用SynchronousResultReceiver对象的awaitResult函数，在获得消息前，会等待一段时间直到超时，获取到结果时，退出等待

//Wifi / BT / Modem能耗模型，统计Tx、Rx等等使用时间来计算电量

final WifiActivityEnergyInfo wifiInfo = awaitControllerInfo(wifiReceiver);

final BluetoothActivityEnergyInfo bluetoothInfo = awaitControllerInfo(bluetoothReceiver);

final ModemActivityInfo modemInfo = awaitControllerInfo(modemReceiver);

synchronized (mStats) {

mStats.addHistoryEventLocked(//电池历史时间，用于电量统计

SystemClock.elapsedRealtime(),

SystemClock.uptimeMillis(),

BatteryStats.HistoryItem.EVENT\_COLLECT\_EXTERNAL\_STATS,

reason, 0);

if ((updateFlags & UPDATE\_CPU) != 0) {

if (useLatestStates) {

onBattery = mStats.isOnBatteryLocked();//充电否

onBatteryScreenOff = mStats.isOnBatteryScreenOffLocked();//充电灭屏否

}

mStats.updateCpuTimeLocked(onBattery, onBatteryScreenOff);//在应用程序中读取和分发CPU使用情况，后文中会详细分析updateXxxXxxXxx

}

if ((updateFlags & UPDATE\_ALL) != 0) {

mStats.updateKernelWakelocksLocked();//读取和分发跨应用程序的内核wakelock

mStats.updateKernelMemoryBandwidthLocked();//从内核读取最新的内存统计信息

}

if ((updateFlags & UPDATE\_RPM) != 0) {

mStats.updateRpmStatsLocked();//读取和记录RPM状态

}

if (bluetoothInfo != null) {

if (bluetoothInfo.isValid()) {

mStats.updateBluetoothStateLocked(bluetoothInfo);//更新BT耗电情况

} else {

Slog.w(TAG, "bluetooth info is invalid: " + bluetoothInfo);

}

}

}

// WiFi and Modem state are updated without the mStats lock held, because they do some network stats retrieval before internally grabbing the mStats lock.

if (wifiInfo != null) {

if (wifiInfo.isValid()) {

mStats.updateWifiState(extractDeltaLocked(wifiInfo));//更新wifi耗电情况

} else {

Slog.w(TAG, "wifi info is invalid: " + wifiInfo);

}

}

if (modemInfo != null) {

if (modemInfo.isValid()) {

mStats.updateMobileRadioState(modemInfo);//更新modem耗电情况

} else {

Slog.w(TAG, "modem info is invalid: " + modemInfo);

}

}

}

### 1.2.3其他接口

@Override

public Future<?> scheduleReadProcStateCpuTimes(boolean onBattery, boolean onBatteryScreenOff, long delayMillis) {

synchronized (mStats) {

if (!mStats.trackPerProcStateCpuTimes()) {

return null;

}

}

synchronized (BatteryExternalStatsWorker.this) {

if (!mExecutorService.isShutdown()) {

return mExecutorService.schedule(PooledLambda.obtainRunnable(

BatteryStatsImpl::updateProcStateCpuTimes, //Update per-freq cpu times for all the uids in

mStats, onBattery, onBatteryScreenOff).recycleOnUse(),

delayMillis, TimeUnit.MILLISECONDS);

}

}

return null;

}

@Override

public Future<?> scheduleCopyFromAllUidsCpuTimes(boolean onBattery, boolean onBatteryScreenOff) {

synchronized (mStats) {

if (!mStats.trackPerProcStateCpuTimes()) {

return null;

}

}

synchronized (BatteryExternalStatsWorker.this) {

if (!mExecutorService.isShutdown()) {

return mExecutorService.submit(PooledLambda.obtainRunnable(

BatteryStatsImpl::copyFromAllUidsCpuTimes,

mStats, onBattery, onBatteryScreenOff).recycleOnUse());

}

}

return null;

}

## 1.3 StopwatchTimer