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：

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

mWorker = new BatteryExternalStatsWorker(context, mStats);//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

mStats.setExternalStatsSyncLocked(mWorker);

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

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

@GuardedBy("this")

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

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

private String mCurrentReason = null;//wakeup reason

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

private final ScheduledExecutorService mExecutorService =

Executors.newSingleThreadScheduledExecutor(

(ThreadFactory) r -> {

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

t.setPriority(Thread.NORM\_PRIORITY);

return t;

});

@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;

}

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();

}

// Clean up any UIDs if necessary.

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();

}

}

};

@Override

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

return scheduleSyncLocked(reason, flags);

}

@Override

public Future<?> scheduleSyncDueToBatteryLevelChange(long delayMillis) {

synchronized (BatteryExternalStatsWorker.this) {

mBatteryLevelSync = scheduleDelayedSyncLocked(mBatteryLevelSync,

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

delayMillis);

return mBatteryLevelSync;

}

}

@GuardedBy("this")

private void cancelSyncDueToBatteryLevelChangeLocked() {//取消battery-level任务

if (mBatteryLevelSync != null) {

mBatteryLevelSync.cancel(false);

mBatteryLevelSync = null;

}

}