# TODO

AlarmManagerService

SystemClock.setCurrentTimeMillis原理

[Android5.1AlarmManagerService深入分析（Android4.4补充）](http://blog.csdn.net/kc58236582/article/details/45666709)

<http://blog.csdn.net/kc58236582/article/details/45666709>

<https://github.com/clarkehe/Android/wiki/Android%E4%BC%91%E7%9C%A0%E6%9C%BA%E5%88%B6>

休眠机制

android-7.1.1\_r1/frameworks/base/services/core/java/com/android/server/AlarmManagerService.java

[AlarmManager 实现闹钟的基本功能](http://www.cnblogs.com/rainly/p/3437627.html)

LC震荡电路原理

# RTC

RTC(Real\_Time Clock)。RTC 是集成电路，通常称为[时钟芯片](https://baike.baidu.com/item/%E6%97%B6%E9%92%9F%E8%8A%AF%E7%89%87/8832713)。

实时时钟芯片是日常生活中应用最为广泛的消费类电子产品之一。它为人们提供精确的实时时间,或者为电子系统提供精确的时间基准,目前实时时钟芯片大多采用精度较高的晶体振荡器作为时钟源。有些时钟芯片为了在主电源掉电时，还可以工作，需要外加电池供电。

## 硬件结构

1） 晶振

晶振一般叫做晶体谐振器，是一种机电器件，晶振是石英振荡器的简称，英文名为Crystal是用电损耗很小的石英晶体经精密切割磨削并镀上电极焊上引线做成。

晶振的作用：提供基准频率。

RTC的晶振：

任何实时时钟的核心都是晶振，晶振频率为32768 Hz 。它为分频计数器提供精确的与低功耗的实基信号。它可以用于产生秒、分、时、日等信息。为了确保时钟长期的准确性，晶振必须正常工作，不能够收到干扰。RTC的晶振又分为：外部晶振和内置晶振。

RTC的晶振频率为什么是32768Hz？

① RTC时间是以振荡频率来计算的。故它不是一个时间器而是一个计数器。而一般的计数器都是16位的。又因为时间的准确性很重要，故震荡次数越低，时间的准确性越低。所以必定是个高次数。215 = 32768 。

② 32768 Hz = 215 即分频15次后为1Hz，周期 = 1s。

③ 经过工程师的经验总结32768 Hz，时钟最准确。

④ 规范和统一。

# 源码分析

在开启AlarmManagerService服务时，会创建服务的实例

final AlarmHandler mHandler = new AlarmHandler();

final Constants mConstants;

public AlarmManagerService(Context context) {

super(context);

// 初始化Handler和常量Constants类

mConstants = new Constants(mHandler);

## 常量类的实现

/\*\*

\* 该类中所有的时间单位都是毫秒。

\* 这些常量保持与系统全局设置一致。

\* 任何访问该类或该类中的字段都要持有AlarmManagerService.mLock锁

\*/

private final class Constants extends ContentObserver {

// 在设置中保存的键值

private static final String KEY\_MIN\_FUTURITY = "min\_futurity";

private static final String KEY\_MIN\_INTERVAL = "min\_interval";

private static final String KEY\_ALLOW\_WHILE\_IDLE\_SHORT\_TIME = "allow\_while\_idle\_short\_time";

private static final String KEY\_ALLOW\_WHILE\_IDLE\_LONG\_TIME = "allow\_while\_idle\_long\_time";

private static final String KEY\_ALLOW\_WHILE\_IDLE\_WHITELIST\_DURATION

= "allow\_while\_idle\_whitelist\_duration";

private static final long DEFAULT\_MIN\_FUTURITY = 5 \* 1000;

private static final long DEFAULT\_MIN\_INTERVAL = 60 \* 1000;

private static final long DEFAULT\_ALLOW\_WHILE\_IDLE\_SHORT\_TIME = DEFAULT\_MIN\_FUTURITY;

private static final long DEFAULT\_ALLOW\_WHILE\_IDLE\_LONG\_TIME = 9\*60\*1000;

private static final long DEFAULT\_ALLOW\_WHILE\_IDLE\_WHITELIST\_DURATION = 10\*1000;

// Minimum futurity of a new alarm

public long MIN\_FUTURITY = DEFAULT\_MIN\_FUTURITY;

// Minimum alarm recurrence interval

public long MIN\_INTERVAL = DEFAULT\_MIN\_INTERVAL;

// 从系统非空闲状态到可以执行flag为ALLOW\_WHILE\_IDLE的alarm的最小时间间隔:5秒钟

public long ALLOW\_WHILE\_IDLE\_SHORT\_TIME = DEFAULT\_ALLOW\_WHILE\_IDLE\_SHORT\_TIME;

// 从系统空闲状态到可以执行flag为ALLOW\_WHILE\_IDLE的alarm的最小时间间隔:9分钟

public long ALLOW\_WHILE\_IDLE\_LONG\_TIME = DEFAULT\_ALLOW\_WHILE\_IDLE\_LONG\_TIME;

// BroadcastOptions.setTemporaryAppWhitelistDuration() to use for FLAG\_ALLOW\_WHILE\_IDLE.

public long ALLOW\_WHILE\_IDLE\_WHITELIST\_DURATION

= DEFAULT\_ALLOW\_WHILE\_IDLE\_WHITELIST\_DURATION;

private ContentResolver mResolver;

private final KeyValueListParser mParser = new KeyValueListParser(',');

private long mLastAllowWhileIdleWhitelistDuration = -1;

public Constants(Handler handler) {

super(handler);

// 更新可以开始执行flag为ALLOW\_WHILE\_IDLE的alarm的最小时间间隔

updateAllowWhileIdleMinTimeLocked();

updateAllowWhileIdleWhitelistDurationLocked();

}

// 系统启动后会调用该方法，注册数据库监听

public void start(ContentResolver resolver) {

mResolver = resolver;

// 监听数据库变化

mResolver.registerContentObserver(Settings.Global.getUriFor(

Settings.Global.ALARM\_MANAGER\_CONSTANTS), false, this);

updateConstants();

}

// 更新可以开始执行flag为ALLOW\_WHILE\_IDLE的alarm的最小时间间隔

public void updateAllowWhileIdleMinTimeLocked() {

mAllowWhileIdleMinTime = mPendingIdleUntil != null

? ALLOW\_WHILE\_IDLE\_LONG\_TIME : ALLOW\_WHILE\_IDLE\_SHORT\_TIME;

}

public void updateAllowWhileIdleWhitelistDurationLocked() {

if (mLastAllowWhileIdleWhitelistDuration != ALLOW\_WHILE\_IDLE\_WHITELIST\_DURATION) {

mLastAllowWhileIdleWhitelistDuration = ALLOW\_WHILE\_IDLE\_WHITELIST\_DURATION;

BroadcastOptions opts = BroadcastOptions.makeBasic();

opts.setTemporaryAppWhitelistDuration(ALLOW\_WHILE\_IDLE\_WHITELIST\_DURATION);

mIdleOptions = opts.toBundle();

}

}

@Override

public void onChange(boolean selfChange, Uri uri) {

// 数据库内容变化时，更新一些本地变量

updateConstants();

}

. . .

}

## onStart

long mNativeData;

// 下一个包含wakeup的batch的start时间

private long mNextWakeup;

// 下一个包含Rtc wakeup的batch的start时间

private long mNextRtcWakeup;

// 下一个非wakeup的batch的start时间

private long mNextNonWakeup;

static final String TIMEZONE\_PROPERTY = "persist.sys.timezone";

PowerManager.WakeLock mWakeLock;

// 时间变化发送者

PendingIntent mTimeTickSender;

// 日期变化发送者

PendingIntent mDateChangeSender;

// 时间变化的广播接收者

ClockReceiver mClockReceiver;

// 监听息屏/亮屏的广播接收者

InteractiveStateReceiver mInteractiveStateReceiver;

// 监听卸载的广播接收者

private UninstallReceiver mUninstallReceiver;

public void onStart() {

// 通过JNI对mNativeData进行初始化操作:

// 打开设备驱动"/dev/alarm"返回一个long型的与fd文件描述符有关的值

mNativeData = init();

mNextWakeup = mNextRtcWakeup = mNextNonWakeup = 0;

// 把当前时区信息保存到内核

// ***TIMEZONE\_PROPERTY*** = **"persist.sys.timezone"**;

setTimeZoneImpl(SystemProperties.get(TIMEZONE\_PROPERTY));

// 初始化wakelock

PowerManager pm = (PowerManager) getContext().getSystemService(Context.POWER\_SERVICE);

mWakeLock = pm.newWakeLock(PowerManager.PARTIAL\_WAKE\_LOCK, "\*alarm\*");

// 初始化时间变化的广播发送者，因为这里的接收者是ALL，所以所有的应用程序收到的ACTION\_TIME\_TICK广播都是这里发送的

mTimeTickSender = PendingIntent.getBroadcastAsUser(getContext(), 0,

new Intent(Intent.ACTION\_TIME\_TICK).addFlags(

Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY

| Intent.FLAG\_RECEIVER\_FOREGROUND), 0,

UserHandle.ALL);

// 初始化mDateChangeSender

Intent intent = new Intent(Intent.ACTION\_DATE\_CHANGED);

intent.addFlags(Intent.FLAG\_RECEIVER\_REPLACE\_PENDING);

mDateChangeSender = PendingIntent.getBroadcastAsUser(getContext(), 0, intent,

Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY\_BEFORE\_BOOT, UserHandle.ALL);

// 初始化驱动来调度alarm

// 注册时间/日期变化广播接收者

mClockReceiver = new ClockReceiver();

// 调度时间变化事件，第一次发送时间变化广播，实现：在当前时间延迟一分钟后发送

// 在ClockReceiver的onReceive方法中接收到时间变化的广博就会调用

// scheduleTimeTickEvent方法发送一分钟后的广播

mClockReceiver.scheduleTimeTickEvent();

// 调度日期变化事件，每天0点发送一次广播

mClockReceiver.scheduleDateChangedEvent();

// 初始化监听息屏/亮屏的广播接收者

mInteractiveStateReceiver = new InteractiveStateReceiver();

// 监听应用卸载和SD卡不可用的广播，判断是否移除alarm

mUninstallReceiver = new UninstallReceiver();

if (mNativeData != 0) {

// 初始化AlarmThread并运行它的run方法

AlarmThread waitThread = new AlarmThread();

waitThread.start();

} else {

Slog.w(TAG, "Failed to open alarm driver. Falling back to a handler.");

}

// 发布服务

publishBinderService(Context.ALARM\_SERVICE, mService);

}

private native long init();

## setTimeZoneImpl

// 把当前时区保存到内核

void setTimeZoneImpl(String tz) {

if (TextUtils.isEmpty(tz)) {

return;

}

// 获取要保存的时区

TimeZone zone = TimeZone.getTimeZone(tz);

// 写入时区时，避免同时写

boolean timeZoneWasChanged = false;

synchronized (this) {

// 再次获取要保存的时区

String current = SystemProperties.get(TIMEZONE\_PROPERTY);

// 如果两次获取的时区不一致则说明时区发生变化，并重新把当前时区保存到设置中

if (current == null || !current.equals(zone.getID())) {

if (localLOGV) {

Slog.v(TAG, "timezone changed: " + current + ", new=" + zone.getID());

}

timeZoneWasChanged = true;

SystemProperties.set(TIMEZONE\_PROPERTY, zone.getID());

}

// Update the kernel timezone information

// Kernel tracks time offsets as 'minutes west of GMT'

// 更新内核时区信息

// 内核跟踪时间偏移为GMT以西的分钟数

int gmtOffset = zone.getOffset(System.currentTimeMillis());

// 保存时区到内核

setKernelTimezone(mNativeData, -(gmtOffset / 60000));

}

TimeZone.setDefault(null);

// 如果时区发生改变则发送时区改变的广播

if (timeZoneWasChanged) {

Intent intent = new Intent(Intent.ACTION\_TIMEZONE\_CHANGED);

intent.addFlags(Intent.FLAG\_RECEIVER\_REPLACE\_PENDING);

intent.putExtra("time-zone", zone.getID());

getContext().sendBroadcastAsUser(intent, UserHandle.ALL);

}

*// 保存时区信息到内核中*

private native int setKernelTimezone(long nativeData, int minuteswest);

g.csdn.net/zhangyongfeiyong/article/details/52224300

## ClockReceiver

class ClockReceiver extends BroadcastReceiver {

public ClockReceiver() {

// 注册时间/日期变化的广播接收者

IntentFilter filter = new IntentFilter();

filter.addAction(Intent.ACTION\_TIME\_TICK);

filter.addAction(Intent.ACTION\_DATE\_CHANGED);

getContext().registerReceiver(this, filter);

}

@Override

public void onReceive(Context context, Intent intent) {

if (intent.getAction().equals(Intent.ACTION\_TIME\_TICK)) {

if (DEBUG\_BATCH) {

Slog.v(TAG, "Received TIME\_TICK alarm; rescheduling");

}

// 收到时间变化的广播，发送时间变化的广播给所有的接收者

scheduleTimeTickEvent();

} else if (intent.getAction().equals(Intent.ACTION\_DATE\_CHANGED)) {

// 由于内核不跟踪DST时间，所以需要在每天开始时，基于当前时区的gmt偏移 + 用户空间跟踪保存的信息来重置时区信息

TimeZone zone = TimeZone.getTimeZone(SystemProperties.get(TIMEZONE\_PROPERTY));

int gmtOffset = zone.getOffset(System.currentTimeMillis());

// 日期变化时更新内核保存的时区信息

setKernelTimezone(mNativeData, -(gmtOffset / 60000));

// 调度日期变化事件

scheduleDateChangedEvent();

}

}

// 一分钟后给所有接收ACTION\_TIME\_TICK广播的接收者发送广播

public void scheduleTimeTickEvent() {

final long currentTime = System.currentTimeMillis();

final long nextTime = 60000 \* ((currentTime / 60000) + 1);

// 下一次执行事件的延迟时间

final long tickEventDelay = nextTime - currentTime;

final WorkSource workSource = null; // Let system take blame for time tick events.

// 调用setImpl方法设置给所有接收ACTION\_TIME\_TICK广播的接收者发送广播

setImpl(ELAPSED\_REALTIME, SystemClock.elapsedRealtime() + tickEventDelay, 0,

0, mTimeTickSender, AlarmManager.FLAG\_STANDALONE, workSource, null,

Process.myUid());

}

// 调度日期变化事件

public void scheduleDateChangedEvent() {

Calendar calendar = Calendar.getInstance();

calendar.setTimeInMillis(System.currentTimeMillis());

calendar.set(Calendar.HOUR, 0);

calendar.set(Calendar.MINUTE, 0);

calendar.set(Calendar.SECOND, 0);

calendar.set(Calendar.MILLISECOND, 0);

calendar.add(Calendar.DAY\_OF\_MONTH, 1);

final WorkSource workSource = null; // Let system take blame for date change events.

// 设置mDateChangeSender

setImpl(RTC, calendar.getTimeInMillis(), 0, 0, mDateChangeSender,

AlarmManager.FLAG\_STANDALONE, workSource, null, Process.myUid());

}

}

// 修正参数并设置传递过来的PendingIntent

void setImpl(int type, long triggerAtTime, long windowLength, long interval,

PendingIntent operation, int flags, WorkSource workSource,

AlarmManager.AlarmClockInfo alarmClock, int callingUid) {

if (operation == null) {

Slog.w(TAG, "set/setRepeating ignored because there is no intent");

return;

}

/\*下面要修正传递过来的PendingIntent各相关参数了，所以可以在这里增加对alarm的管控规则\*/

// 检查时间窗长度，如果超过半天就转换成一小时。

if (windowLength > AlarmManager.INTERVAL\_HALF\_DAY) {

Slog.w(TAG, "Window length " + windowLength

+ "ms suspiciously long; limiting to 1 hour");

windowLength = AlarmManager.INTERVAL\_HOUR;

}

// 检查alarm的执行周期，如果alarm的执行周期小于1分钟，则转换为1分钟。

// 从这里可以得知周期性alarm的最小有效周期为1分钟，若要执行周期小于1分钟的事，不能使用alarm实现

final long minInterval = mConstants.MIN\_INTERVAL;// 1分钟

if (interval > 0 && interval < minInterval) {

Slog.w(TAG, "Suspiciously short interval " + interval

+ " millis; expanding to " + (minInterval/1000)

+ " seconds");

interval = minInterval;

}

// 检查type的合法性

if (type < RTC\_WAKEUP || type > RTC\_POWEROFF\_WAKEUP) {

throw new IllegalArgumentException("Invalid alarm type " + type);

}

// 检查triggerAtTime的合法性

if (triggerAtTime < 0) {

final long what = Binder.getCallingPid();

Slog.w(TAG, "Invalid alarm trigger time! " + triggerAtTime + " from uid=" + callingUid

+ " pid=" + what);

triggerAtTime = 0;

}

final long nowElapsed = SystemClock.elapsedRealtime();

// 把RTC格式的转换成elapsedRealtime格式的触发时间

final long nominalTrigger = convertToElapsed(triggerAtTime, type);

// 为了防止alarm频繁滥发执行，这里把当前时间延迟5秒当做alarm的最早触发时间

final long minTrigger = nowElapsed + mConstants.MIN\_FUTURITY;// 5秒

final long triggerElapsed = (nominalTrigger > minTrigger) ? nominalTrigger : minTrigger;

final long maxElapsed;

if (windowLength == AlarmManager.WINDOW\_EXACT) {

// 如果alarm的时间窗为0，说明为精确时间执行的alarm，则把上面计算得到的触发时间赋值给最晚触发时间

maxElapsed = triggerElapsed;

} else if (windowLength < 0) {

// 计算alarm的最晚触发时间

maxElapsed = maxTriggerTime(nowElapsed, triggerElapsed, interval);

// 根据最晚触发时间修正alarm的时间窗时间

windowLength = maxElapsed - triggerElapsed;

} else {

// 如果时间窗时间大于0，则把触发时间+时间窗时间赋值给最晚触发时间

maxElapsed = triggerElapsed + windowLength;

}

synchronized (mLock) {

if (DEBUG\_BATCH) {

Slog.v(TAG, "set(" + operation + ") : type=" + type

+ " triggerAtTime=" + triggerAtTime + " win=" + windowLength

+ " tElapsed=" + triggerElapsed + " maxElapsed=" + maxElapsed

+ " interval=" + interval + " flags=0x" + Integer.toHexString(flags));

}

// 根据修正后的各参数进一步设置alarm

setImplLocked(type, triggerAtTime, triggerElapsed, windowLength, maxElapsed,

interval, operation, flags, true, workSource, alarmClock, callingUid);

}

}

// 如果alarm的执行时间设置的是RTC则需要转换成开机时间

static long convertToElapsed(long when, int type) {

final boolean isRtc = (type == RTC || type == RTC\_WAKEUP

|| type == RTC\_POWEROFF\_WAKEUP);

if (isRtc) {

when -= System.currentTimeMillis() - SystemClock.elapsedRealtime();

}

return when;

}

// 最小重复周期

static final long MIN\_FUZZABLE\_INTERVAL = 10000;

// 最晚触发时间

static long maxTriggerTime(long now, long triggerAtTime, long interval) {

long futurity = (interval == 0)

? (triggerAtTime - now)

: interval;

// 如果下一次触发的时间小于10秒则不为alarm增加最晚触发时间

if (futurity < MIN\_FUZZABLE\_INTERVAL) {

futurity = 0;

}

// 把0.75倍的下次触发时间+触发时间赋值给alarm的maxElapsed

return triggerAtTime + (long)(.75 \* futurity);

}

// 根据传递来的各alarm相关参数进一步设置alarm

private void setImplLocked(int type, long when, long whenElapsed, long windowLength,

long maxWhen, long interval, PendingIntent operation, int flags,

boolean doValidate, WorkSource workSource, AlarmManager.AlarmClockInfo alarmClock,

int uid) {

// 根据传递过来的参数重新创建一个alarm实例

Alarm a = new Alarm(type, when, whenElapsed, windowLength, maxWhen, interval,

operation, workSource, flags, alarmClock, uid);

// 根据operation参数从所在列表中移除该alarm

removeLocked(operation);

// 把新创建的alarm添加到相关列表

setImplLocked(a, false, doValidate);

}

// alarm批处理对象列表

final ArrayList<Batch> mAlarmBatches = new ArrayList<>();

// 延迟到空闲状态时执行的alarm列表

ArrayList<Alarm> mPendingWhileIdleAlarms = new ArrayList<>();

// 由于空闲时开始执行的alarm只有一个，故后续其他也需要在空闲时开始执行的alarm就需要保存在mPendingWhileIdleAlarms列表中

Alarm mPendingIdleUntil = null;

// 下一个请求退出空闲模式的alarm

Alarm mNextWakeFromIdle = null;

// 根据operation从各列表中查找alarm并移除

private void removeLocked(PendingIntent operation) {

boolean didRemove = false;

// 遍历mAlarmBatches查找目标alarm

for (int i = mAlarmBatches.size() - 1; i >= 0; i--) {

Batch b = mAlarmBatches.get(i);

ArrayList<Alarm> alarmList = b.alarms;

Alarm alarm = null;

for (int j = alarmList.size() - 1; j >= 0; j--) {

alarm = alarmList.get(j);

if (alarm.type == RTC\_POWEROFF\_WAKEUP && alarm.operation.equals(operation)) {

long alarmSeconds, alarmNanoseconds;

alarmSeconds = alarm.when / 1000;

alarmNanoseconds = (alarm.when % 1000) \* 1000 \* 1000;

// 遍历所有alarm，找到和目标operation相同的alarm，

// 如果该alarm的类型是Rtc wakeup的则清空在驱动中的相关设置

clear(mNativeData, alarm.type, alarmSeconds, alarmNanoseconds);

mNextRtcWakeup = 0;

}

}

// 把和目标operation相同的alarm从批处理中移除掉

didRemove |= b.remove(operation);

// 移除alarm后，如果批处理的长度为0，则从mAlarmBatches中移除掉该批处理对象

if (b.size() == 0) {

mAlarmBatches.remove(i);

}

}

// 遍历mPendingWhileIdleAlarms查找目标alarm并移除

for (int i = mPendingWhileIdleAlarms.size() - 1; i >= 0; i--) {

if (mPendingWhileIdleAlarms.get(i).operation.equals(operation)) {

// 从mPendingWhileIdleAlarms列表中移除alarm不用设置didRemove字段，

// 因为该列表中的alarm都是在手机转变到空闲状态时执行的，并不影响alarm的调度。

mPendingWhileIdleAlarms.remove(i);

}

}

// 如果从相关列表中移除相关alarm，则需要重新处理各相关列表中的alarm

if (didRemove) {

if (DEBUG\_BATCH) {

Slog.v(TAG, "remove(operation) changed bounds; rebatching");

}

boolean restorePending = false;

// 查找目标alarm是否为mPendingIdleUntil，如果是则赋null，下面会重新存储空闲时延迟执行的alarm

if (mPendingIdleUntil != null && mPendingIdleUntil.operation.equals(operation)) {

mPendingIdleUntil = null;

restorePending = true;

}

// 查找目标alarm是否为mNextWakeFromIdle，如果是则赋null

if (mNextWakeFromIdle != null && mNextWakeFromIdle.operation.equals(operation)) {

mNextWakeFromIdle = null;

}

// 对所有的alarm进行重新批处理操作

rebatchAllAlarmsLocked(true);

if (restorePending) {

// 由于mPendingIdleUntil变量改变，所以需要重新存储空闲时延迟执行的alarm

restorePendingWhileIdleAlarmsLocked();

}

// 更新下一个alarm的alarmClock

updateNextAlarmClockLocked();

}

}

// 根据alarm的类型、触发时间等参数从驱动中移除掉该alarm

private native void clear(long nativeData, int type, long seconds, long nanoseconds);

// 对所有的alarm进行重新批处理操作

void rebatchAllAlarmsLocked(boolean doValidate) {

ArrayList<Batch> oldSet = (ArrayList<Batch>) mAlarmBatches.clone();

mAlarmBatches.clear();

Alarm oldPendingIdleUntil = mPendingIdleUntil;

final long nowElapsed = SystemClock.elapsedRealtime();

final int oldBatches = oldSet.size();

for (int batchNum = 0; batchNum < oldBatches; batchNum++) {

Batch batch = oldSet.get(batchNum);

final int N = batch.size();

for (int i = 0; i < N; i++) {

// 重新添加alarm到mAlarmBatches列表中

reAddAlarmLocked(batch.get(i), nowElapsed, doValidate);

}

}

if (oldPendingIdleUntil != null && oldPendingIdleUntil != mPendingIdleUntil) {

Slog.wtf(TAG, "Rebatching: idle until changed from " + oldPendingIdleUntil

+ " to " + mPendingIdleUntil);

if (mPendingIdleUntil == null) {

// mPendingIdleUntil变化导致需要重新存储空闲时延迟执行的alarm

restorePendingWhileIdleAlarmsLocked();

}

}

// 重新设置几个核心alarm

rescheduleKernelAlarmsLocked();

// 更新下一个alarm的alarmClock

updateNextAlarmClockLocked();

}

// 重新添加alarm到mAlarmBatches列表中

void reAddAlarmLocked(Alarm a, long nowElapsed, boolean doValidate) {

a.when = a.origWhen;

long whenElapsed = convertToElapsed(a.when, a.type);

final long maxElapsed;

// windowLength：alarm执行的时间窗时间

if (a.windowLength == AlarmManager.WINDOW\_EXACT) {

// 精确执行时间

maxElapsed = whenElapsed;

} else {

// 非精确执行时间，即允许在时间窗误差内触发

maxElapsed = (a.windowLength > 0)

? (whenElapsed + a.windowLength)

: maxTriggerTime(nowElapsed, whenElapsed, a.repeatInterval);

}

a.whenElapsed = whenElapsed;

a.maxWhenElapsed = maxElapsed;

// 重新添加alarm到各相关列表中

setImplLocked(a, true, doValidate);

}

Random mRandom;

// 下一个将要执行的alarm的开始执行时间是否可能改变

private boolean mNextAlarmClockMayChange;

// 把alarm添加各相关列表中

private void setImplLocked(Alarm a, boolean rebatching, boolean doValidate) {

// 判断alarm是否设置了FLAG\_IDLE\_UNTIL

if ((a.flags&AlarmManager.FLAG\_IDLE\_UNTIL) != 0) {

// 这是一个特殊的alarm，它将系统进入空闲模式，直到该alarm执行完毕。

// 虽然调用者A设置了alarm的开始执行时间，但是如果存在在更早的时间请求退出空闲状态的alarm，则需要让A的alarm提前执行

if (mNextWakeFromIdle != null && a.whenElapsed > mNextWakeFromIdle.whenElapsed) {

// 令alarm的开始执行时间等于请求退出空闲状态的alarm的开始执行时间

a.when = a.whenElapsed = a.maxWhenElapsed = mNextWakeFromIdle.whenElapsed;

}

// 由于上面令alarm的开始执行时间等于请求退出空闲状态的alarm的开始执行时间，所以alarm的开始执行时间还要提前。

// 这里增加模糊时间，让alarm比实际开始执行时间早一点执行。

final long nowElapsed = SystemClock.elapsedRealtime();

// 计算模糊时间

final int fuzz = fuzzForDuration(a.whenElapsed-nowElapsed);

if (fuzz > 0) {

if (mRandom == null) {

mRandom = new Random();

}

// 根据模糊时间取一个随机数

final int delta = mRandom.nextInt(fuzz);

// alarm的开始执行时间减去随机数，让alarm早执行一点

a.whenElapsed -= delta;

if (false) {

Slog.d(TAG, "Alarm when: " + a.whenElapsed);

Slog.d(TAG, "Delta until alarm: " + (a.whenElapsed-nowElapsed));

Slog.d(TAG, "Applied fuzz: " + fuzz);

Slog.d(TAG, "Final delta: " + delta);

Slog.d(TAG, "Final when: " + a.whenElapsed);

}

a.when = a.maxWhenElapsed = a.whenElapsed;

}

} else if (mPendingIdleUntil != null) {

// 调用该方法时，该alarm可能设置了FLAG\_ALLOW\_WHILE\_IDLE，所以只要mPendingIdleUntil不为null

// 且没有明确表示要在空闲状态时运行，则就需要将它添加到mPendingWhileIdleAlarms延迟开始执行列表中

if ((a.flags&(AlarmManager.FLAG\_ALLOW\_WHILE\_IDLE

| AlarmManager.FLAG\_ALLOW\_WHILE\_IDLE\_UNRESTRICTED

| AlarmManager.FLAG\_WAKE\_FROM\_IDLE))

== 0) {

mPendingWhileIdleAlarms.add(a);

return;

}

}

// 获取批处理对象在mAlarmBatches列表中的索引值

int whichBatch = ((a.flags&AlarmManager.FLAG\_STANDALONE) != 0)

? -1 : attemptCoalesceLocked(a.whenElapsed, a.maxWhenElapsed);

if (whichBatch < 0) {

// 如果alarm是独立的，则它不能和其他alarm一起进行批处理操作

Batch batch = new Batch(a);

// 把批处理对象添加到mAlarmBatches列表中

addBatchLocked(mAlarmBatches, batch);

} else {

Batch batch = mAlarmBatches.get(whichBatch);

// 把alarm添加到批处理对象中，如果批处理对象的start时间变大则返回true

if (batch.add(a)) {

// 批处理的开始时间提前导致批处理的alarm顺序可能被打破。所以要把批处理对象移动到正确的位置

mAlarmBatches.remove(whichBatch);

// 重新把批处理对象添加到mAlarmBatches列表中的正确位置上

addBatchLocked(mAlarmBatches, batch);

}

}

if (a.alarmClock != null) {

mNextAlarmClockMayChange = true;

}

boolean needRebatch = false;

// 判断alarm是否设置了FLAG\_IDLE\_UNTIL

if ((a.flags&AlarmManager.FLAG\_IDLE\_UNTIL) != 0) {

// 由于该alarm设置了FLAG\_ALLOW\_WHILE\_IDLE,所以把该alarm赋值给mPendingIdleUntil变量

mPendingIdleUntil = a;

// 更新可以开始执行flag为ALLOW\_WHILE\_IDLE的alarm的最小时间间隔

mConstants.updateAllowWhileIdleMinTimeLocked();

// 给mPendingIdleUntil变量赋值可能导致alarm的执行时间延后，故需要重新进行批处理

needRebatch = true;

// 判断alarm是否设置了FLAG\_WAKE\_FROM\_IDLE

} else if ((a.flags&AlarmManager.FLAG\_WAKE\_FROM\_IDLE) != 0) {

// 如果下一个请求退出空闲状态的alarm为null或者下一个请求退出空闲状态的alarm的开始执行时间大于a的开始执行时间

if (mNextWakeFromIdle == null || mNextWakeFromIdle.whenElapsed > a.whenElapsed) {

// 则把a赋值给mNextWakeFromIdle

mNextWakeFromIdle = a;

// 如果延迟到空闲时开始执行的alarm不为null，因为两者开始执行时间可能会有冲突，所以需要重新进行批处理

if (mPendingIdleUntil != null) {

needRebatch = true;

}

}

}

if (!rebatching) {

if (needRebatch) {

// 由于相关alarm变量改变，所以需要对所有的alarm进行重新批处理操作

rebatchAllAlarmsLocked(false);

}

// 重新设置几个核心alarm

rescheduleKernelAlarmsLocked();

// 更新下一个alarm的alarmClock

updateNextAlarmClockLocked();

}

}

// 计算模糊时间

static int fuzzForDuration(long duration) {

if (duration < 15\*60\*1000) {

// 如果距离开始执行alarm的时间duration小于15分钟，则fuzz就等于duration

return (int)duration;

} else if (duration < 90\*60\*1000) {

// 如果duration小于1.5小时，则令fuzz等于15分钟

return 15\*60\*1000;

} else {

// 否则，令fuzz等于30分钟

return 30\*60\*1000;

}

}

// 返回匹配的批处理对象的索引，如果不存在则返回-1

int attemptCoalesceLocked(long whenElapsed, long maxWhen) {

final int N = mAlarmBatches.size();

for (int i = 0; i < N; i++) {

Batch b = mAlarmBatches.get(i);

if ((b.flags&AlarmManager.FLAG\_STANDALONE) == 0 && b.canHold(whenElapsed, maxWhen)) {

return i;

}

}

return -1;

}

// 根据批处理对象的start时间从小到大排序

static final BatchTimeOrder sBatchOrder = new BatchTimeOrder();

// 如果批处理对象添加到列表的第一位则返回true

static boolean addBatchLocked(ArrayList<Batch> list, Batch newBatch) {

// 根据sBatchOrder在list中用二分法查找newBatch元素，查到返回索引值，否则返回-1

int index = Collections.binarySearch(list, newBatch, sBatchOrder);

if (index < 0) {

index = 0 - index - 1;

}

list.add(index, newBatch);

return (index == 0);

}

static class BatchTimeOrder implements Comparator<Batch> {

public int compare(Batch b1, Batch b2) {

long when1 = b1.start;

long when2 = b2.start;

if (when1 - when2 > 0) {

return 1;

}

if (when1 - when2 < 0) {

return -1;

}

return 0;

}

}

// 延迟执行的非wakeup的alarm列表

ArrayList<Alarm> mPendingNonWakeupAlarms = new ArrayList<>();

// 下一次非wakeup类型的alarm传递的时间

long mNextNonWakeupDeliveryTime;

// 重新设置几个核心alarm（几个单独命名的特定alarm）

void rescheduleKernelAlarmsLocked() {

// Schedule the next upcoming wakeup alarm. If there is a deliverable batch

// prior to that which contains no wakeups, we schedule that as well.

// 调度下一个即将执行的wakeup alarm。如果之前有一个非wakeup的批处理对象，我们也调度它。

long nextNonWakeup = 0;

if (mAlarmBatches.size() > 0) {

// 获取第一个包含wakeup的批处理对象

final Batch firstWakeup = findFirstWakeupBatchLocked();

// 获取mAlarmBatches列表中的第一个批处理对象

final Batch firstBatch = mAlarmBatches.get(0);

// 获取第一个包含Rtc wakeup的批处理对象

final Batch firstRtcWakeup = findFirstRtcWakeupBatchLocked();

if (firstWakeup != null && mNextWakeup != firstWakeup.start) {

// 更新mNextWakeup

mNextWakeup = firstWakeup.start;

// 把alarm设置到驱动中

setLocked(ELAPSED\_REALTIME\_WAKEUP, firstWakeup.start);

}

if (firstRtcWakeup != null && mNextRtcWakeup != firstRtcWakeup.start) {

// 更新mNextRtcWakeup

mNextRtcWakeup = firstRtcWakeup.start;

// 根据alarm.whenElapsed获取alarm.when

long when = firstRtcWakeup.getWhenByElapsedTime(mNextRtcWakeup);

if (when != 0) {

setLocked(RTC\_POWEROFF\_WAKEUP, when);

}

}

if (firstBatch != firstWakeup) {

// 把mAlarmBatches列表中的第一个批处理对象的start赋值给nextNonWakeup

nextNonWakeup = firstBatch.start;

}

}

if (mPendingNonWakeupAlarms.size() > 0) {

if (nextNonWakeup == 0 || mNextNonWakeupDeliveryTime < nextNonWakeup) {

// 更新nextNonWakeup

nextNonWakeup = mNextNonWakeupDeliveryTime;

}

}

if (nextNonWakeup != 0 && mNextNonWakeup != nextNonWakeup) {

// 更新mNextNonWakeup

mNextNonWakeup = nextNonWakeup;

setLocked(ELAPSED\_REALTIME, nextNonWakeup);

}

}

// 查找第一个包含wakeup的批处理对象

private Batch findFirstWakeupBatchLocked() {

final int N = mAlarmBatches.size();

for (int i = 0; i < N; i++) {

Batch b = mAlarmBatches.get(i);

if (b.hasWakeups()) {

return b;

}

}

return null;

}

// 查找第一个包含Rtc wakeup的批处理对象

private Batch findFirstRtcWakeupBatchLocked() {

final int N = mAlarmBatches.size();

for (int i = 0; i < N; i++) {

Batch b = mAlarmBatches.get(i);

long intervalTime = b.start - SystemClock.elapsedRealtime();

if (b.isRtcPowerOffWakeup()) {

return b;

}

}

return null;

}

// 把alarm设置到驱动中

private void setLocked(int type, long when) {

if (mNativeData != 0) {

// The kernel never triggers alarms with negative wakeup times

// so we ensure they are positive.

// 秒、纳秒

long alarmSeconds, alarmNanoseconds;

if (when < 0) {

alarmSeconds = 0;

alarmNanoseconds = 0;

} else {

alarmSeconds = when / 1000;

alarmNanoseconds = (when % 1000) \* 1000 \* 1000;

}

// 把alarm设置到驱动中

set(mNativeData, type, alarmSeconds, alarmNanoseconds);

} else {

Message msg = Message.obtain();

msg.what = ALARM\_EVENT;

mHandler.removeMessages(ALARM\_EVENT);

mHandler.sendMessageAtTime(msg, when);

}

}

// 设置alarm只跟类型、触发时间相关，跟alarm的其他参数无关

private native void set(long nativeData, int type, long seconds, long nanoseconds);

private final SparseArray<AlarmManager.AlarmClockInfo> mTmpSparseAlarmClockArray =

new SparseArray<>();

private final SparseArray<AlarmManager.AlarmClockInfo> mNextAlarmClockForUser =

new SparseArray<>();

/\*\*

\* 更新下一个alarm的alarmClock

\*/

private void updateNextAlarmClockLocked() {

// 如果下一个将要执行的alarm的开始执行时间没有改变则直接返回

if (!mNextAlarmClockMayChange) {

return;

}

mNextAlarmClockMayChange = false;

SparseArray<AlarmManager.AlarmClockInfo> nextForUser = mTmpSparseAlarmClockArray;

nextForUser.clear();

final int N = mAlarmBatches.size();

for (int i = 0; i < N; i++) {

ArrayList<Alarm> alarms = mAlarmBatches.get(i).alarms;

final int M = alarms.size();

for (int j = 0; j < M; j++) {

Alarm a = alarms.get(j);

if (a.alarmClock != null) {

final int userId = UserHandle.getUserId(a.uid);

if (DEBUG\_ALARM\_CLOCK) {

Log.v(TAG, "Found AlarmClockInfo at " +

formatNextAlarm(getContext(), a.alarmClock, userId) +

" for user " + userId);

}

// Alarms and batches are sorted by time, no need to compare times here.

if (nextForUser.get(userId) == null) {

// 把userId和alarmClock以键值对的形式添加到nextForUser中

nextForUser.put(userId, a.alarmClock);

}

}

}

}

// Update mNextAlarmForUser with new values.

final int NN = nextForUser.size();

for (int i = 0; i < NN; i++) {

AlarmManager.AlarmClockInfo newAlarm = nextForUser.valueAt(i);

int userId = nextForUser.keyAt(i);

AlarmManager.AlarmClockInfo currentAlarm = mNextAlarmClockForUser.get(userId);

if (!newAlarm.equals(currentAlarm)) {

// 更新mNextAlarmClockForUser

updateNextAlarmInfoForUserLocked(userId, newAlarm);

}

}

// Remove users without any alarm clocks scheduled.

final int NNN = mNextAlarmClockForUser.size();

for (int i = NNN - 1; i >= 0; i--) {

int userId = mNextAlarmClockForUser.keyAt(i);

if (nextForUser.get(userId) == null) {

// 更新mNextAlarmClockForUser

updateNextAlarmInfoForUserLocked(userId, null);

}

}

}

private final SparseBooleanArray mPendingSendNextAlarmClockChangedForUser =

new SparseBooleanArray();

// 更新mNextAlarmClockForUser

private void updateNextAlarmInfoForUserLocked(int userId,

AlarmManager.AlarmClockInfo alarmClock) {

if (alarmClock != null) {

if (DEBUG\_ALARM\_CLOCK) {

Log.v(TAG, "Next AlarmClockInfoForUser(" + userId + "): " +

formatNextAlarm(getContext(), alarmClock, userId));

}

mNextAlarmClockForUser.put(userId, alarmClock);

} else {

if (DEBUG\_ALARM\_CLOCK) {

Log.v(TAG, "Next AlarmClockInfoForUser(" + userId + "): None");

}

mNextAlarmClockForUser.remove(userId);

}

mPendingSendNextAlarmClockChangedForUser.put(userId, true);

mHandler.removeMessages(AlarmHandler.SEND\_NEXT\_ALARM\_CLOCK\_CHANGED);

mHandler.sendEmptyMessage(AlarmHandler.SEND\_NEXT\_ALARM\_CLOCK\_CHANGED);

}

// 重新存储延迟到空闲时执行的alarm

void restorePendingWhileIdleAlarmsLocked() {

if (mPendingWhileIdleAlarms.size() > 0) {

ArrayList<Alarm> alarms = mPendingWhileIdleAlarms;

mPendingWhileIdleAlarms = new ArrayList<>();

final long nowElapsed = SystemClock.elapsedRealtime();

for (int i=alarms.size() - 1; i >= 0; i--) {

Alarm a = alarms.get(i);

// 重新添加alarm到mAlarmBatches列表中

reAddAlarmLocked(a, nowElapsed, false);

}

}

// 更新可以开始执行flag为ALLOW\_WHILE\_IDLE的alarm的最小时间间隔

mConstants.updateAllowWhileIdleMinTimeLocked();

// Reschedule everything.

// 重新设置几个核心alarm

rescheduleKernelAlarmsLocked();

// 更新下一个alarm的alarmClock

updateNextAlarmClockLocked();

// And send a TIME\_TICK right now, since it is important to get the UI updated.

try {

// 执行和PendingIntent关联的操作

mTimeTickSender.send();

} catch (PendingIntent.CanceledException e) {

}

}

// 监听息屏/亮屏的广播接收者

class InteractiveStateReceiver extends BroadcastReceiver {

public InteractiveStateReceiver() {

IntentFilter filter = new IntentFilter();

filter.addAction(Intent.ACTION\_SCREEN\_OFF);

filter.addAction(Intent.ACTION\_SCREEN\_ON);

filter.setPriority(IntentFilter.SYSTEM\_HIGH\_PRIORITY);

getContext().registerReceiver(this, filter);

}

@Override

public void onReceive(Context context, Intent intent) {

synchronized (mLock) {

// 息屏/亮屏状态发生变化

interactiveStateChangedLocked(Intent.ACTION\_SCREEN\_ON.equals(intent.getAction()));

}

}

}

// 记录上次亮屏状态，亮屏为true

boolean mInteractive = true;

// 延迟执行的非wakeup的alarm的开始延迟的时间点

long mStartCurrentDelayTime;

// 延迟执行的非wakeup的alarm的总延迟时间

long mTotalDelayTime = 0;

// 延迟执行的非wakeup的alarm的最大一次延迟时间

long mMaxDelayTime = 0;

// 最长的息屏时间，后面打log用

long mNonInteractiveTime;

// 开始息屏的时间

long mNonInteractiveStartTime;

// 息屏/亮屏状态发生变化

void interactiveStateChangedLocked(boolean interactive) {

if (mInteractive != interactive) {

mInteractive = interactive;

final long nowELAPSED = SystemClock.elapsedRealtime();

// 如果当前是亮屏状态

if (interactive) {

// 判断延迟执行的非wakeup的alarm列表是否大于0

if (mPendingNonWakeupAlarms.size() > 0) {

// 计算本次延迟时间

final long thisDelayTime = nowELAPSED - mStartCurrentDelayTime;

// 累加到总延迟时间中

mTotalDelayTime += thisDelayTime;

// 更新最大一次延迟时间

if (mMaxDelayTime < thisDelayTime) {

mMaxDelayTime = thisDelayTime;

}

// 执行mPendingNonWakeupAlarms列表中的alarm

deliverAlarmsLocked(mPendingNonWakeupAlarms, nowELAPSED);

// 清空列表

mPendingNonWakeupAlarms.clear();

}

// 保存最长的息屏时间

if (mNonInteractiveStartTime > 0) {

long dur = nowELAPSED - mNonInteractiveStartTime;

if (dur > mNonInteractiveTime) {

mNonInteractiveTime = dur;

}

}

} else {

// 如果当前是息屏状态则开始息屏的时间

mNonInteractiveStartTime = nowELAPSED;

}

}

}

// 上次传递alarm的时间

long mLastAlarmDeliveryTime;

static final boolean RECORD\_ALARMS\_IN\_HISTORY = true;

static final Intent mBackgroundIntent = new Intent()

.addFlags(Intent.FLAG\_FROM\_BACKGROUND);

int mBroadcastRefCount = 0;

// 执行中的alarm的信息列表

ArrayList<InFlight> mInFlight = new ArrayList<>();

// 触发执行的alarm所在应用程序的uid列表

private final ArrayList<Integer> mTriggeredUids = new ArrayList<Integer>();

final ResultReceiver mResultReceiver = new ResultReceiver();

// 执行triggerList中的alarm

void deliverAlarmsLocked(ArrayList<Alarm> triggerList, long nowELAPSED) {

mLastAlarmDeliveryTime = nowELAPSED;

for (int i=0; i<triggerList.size(); i++) {

Alarm alarm = triggerList.get(i);

final boolean allowWhileIdle = (alarm.flags&AlarmManager.FLAG\_ALLOW\_WHILE\_IDLE) != 0;

try {

if (localLOGV) {

Slog.v(TAG, "sending alarm " + alarm);

}

if (RECORD\_ALARMS\_IN\_HISTORY) {

if (alarm.workSource != null && alarm.workSource.size() > 0) {

for (int wi=0; wi<alarm.workSource.size(); wi++) {

// 通过Binder对象调用AMS的noteAlarmStart方法

ActivityManagerNative.noteAlarmStart(

alarm.operation, alarm.workSource.get(wi), alarm.tag);

}

} else {

ActivityManagerNative.noteAlarmStart(

alarm.operation, -1, alarm.tag);

}

}

// 执行和PendingIntent关联的操作

alarm.operation.send(getContext(), 0,

mBackgroundIntent.putExtra(

Intent.EXTRA\_ALARM\_COUNT, alarm.count),

mResultReceiver, mHandler, null, allowWhileIdle ? mIdleOptions : null);

// we have an active broadcast so stay awake.

// 获取wakelock

if (mBroadcastRefCount == 0 || !mWakeLock.isHeld()) {

setWakelockWorkSource(alarm.operation, alarm.workSource,

alarm.type, alarm.tag, true);

mWakeLock.acquire();

}

final InFlight inflight = new InFlight(AlarmManagerService.this,

alarm.operation,

alarm.workSource,

alarm.type, alarm.tag,

nowELAPSED, alarm.uid);

mInFlight.add(inflight);

mBroadcastRefCount++;

mTriggeredUids.add(new Integer(alarm.uid));

// 释放wakelock

if (checkReleaseWakeLock()) {

if (mWakeLock.isHeld()) {

mWakeLock.release();

if (localLOGV) Slog.v(TAG, "AM WakeLock Released Internally deliverAlarms");

}

}

if (allowWhileIdle) {

// Record the last time this uid handled an ALLOW\_WHILE\_IDLE alarm.

mLastAllowWhileIdleDispatch.put(alarm.uid, nowELAPSED);

}

final BroadcastStats bs = inflight.mBroadcastStats;

bs.count++;

if (bs.nesting == 0) {

bs.nesting = 1;

bs.startTime = nowELAPSED;

} else {

bs.nesting++;

}

final FilterStats fs = inflight.mFilterStats;

fs.count++;

if (fs.nesting == 0) {

fs.nesting = 1;

fs.startTime = nowELAPSED;

} else {

fs.nesting++;

}

// 判断是否为WAKEUP类型的alarm

if (alarm.type == ELAPSED\_REALTIME\_WAKEUP

|| alarm.type == RTC\_WAKEUP

|| alarm.type == RTC\_POWEROFF\_WAKEUP) {

bs.numWakeup++;

fs.numWakeup++;

if (alarm.workSource != null && alarm.workSource.size() > 0) {

for (int wi=0; wi<alarm.workSource.size(); wi++) {

// 通过Binder对象调用AMS中的noteWakeupAlarm方法

ActivityManagerNative.noteWakeupAlarm(

alarm.operation, alarm.workSource.get(wi),

alarm.workSource.getName(wi), alarm.tag);

}

} else {

ActivityManagerNative.noteWakeupAlarm(

alarm.operation, -1, null, alarm.tag);

}

}

} catch (PendingIntent.CanceledException e) {

if (alarm.repeatInterval > 0) {

// This IntentSender is no longer valid, but this

// is a repeating alarm, so toss the hoser.

removeImpl(alarm.operation);

}

} catch (RuntimeException e) {

Slog.w(TAG, "Failure sending alarm.", e);

}

}

}

// 等待执行alarm所在应用程序的uid列表

private final ArrayList<Integer> mBlockedUids = new ArrayList<Integer>();

// 判断是否可以释放wakelock

boolean checkReleaseWakeLock() {

if (mTriggeredUids.size() == 0 || mBlockedUids.size() == 0)

return false;

int uid;

for (int i = 0; i < mTriggeredUids.size(); i++) {

uid = mTriggeredUids.get(i);

// 只要等待执行列表中有未处理的alarm就返回false

if (!mBlockedUids.contains(uid)) {

return false;

}

}

return true;

}

————————————————

版权声明：本文为CSDN博主「张永飞」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/zhangyongfeiyong/article/details/52224300>

## UninstallReceiver

// 监听应用卸载和SD卡不可用的广播，判断是否移除alarm

class UninstallReceiver extends BroadcastReceiver {

public UninstallReceiver() {

IntentFilter filter = new IntentFilter();

filter.addAction(Intent.ACTION\_PACKAGE\_REMOVED);

filter.addAction(Intent.ACTION\_PACKAGE\_RESTARTED);

filter.addAction(Intent.ACTION\_QUERY\_PACKAGE\_RESTART);

filter.addDataScheme("package");

getContext().registerReceiverAsUser(this, UserHandle.ALL, filter, null, null);

IntentFilter sdFilter = new IntentFilter();

sdFilter.addAction(Intent.ACTION\_EXTERNAL\_APPLICATIONS\_UNAVAILABLE);

sdFilter.addAction(Intent.ACTION\_USER\_STOPPED);

sdFilter.addAction(Intent.ACTION\_UID\_REMOVED);

getContext().registerReceiverAsUser(this, UserHandle.ALL, sdFilter, null, null);

}

@Override

public void onReceive(Context context, Intent intent) {

synchronized (mLock) {

String action = intent.getAction();

String pkgList[] = null;

int userHandle = intent.getIntExtra(Intent.EXTRA\_USER\_HANDLE, -1);

if (Intent.ACTION\_QUERY\_PACKAGE\_RESTART.equals(action)) {

pkgList = intent.getStringArrayExtra(Intent.EXTRA\_PACKAGES);

for (String packageName : pkgList) {

if (lookForPackageLocked(packageName, userHandle)) {

setResultCode(Activity.RESULT\_OK);

return;

}

}

return;

} else if (Intent.ACTION\_EXTERNAL\_APPLICATIONS\_UNAVAILABLE.equals(action)) {

pkgList = intent.getStringArrayExtra(Intent.EXTRA\_CHANGED\_PACKAGE\_LIST);

} else if (Intent.ACTION\_USER\_STOPPED.equals(action)) {

if (userHandle >= 0) {

removeUserLocked(userHandle);

}

} else if (Intent.ACTION\_UID\_REMOVED.equals(action)) {

int uid = intent.getIntExtra(Intent.EXTRA\_UID, -1);

if (uid >= 0) {

mLastAllowWhileIdleDispatch.delete(uid);

}

} else {

if (Intent.ACTION\_PACKAGE\_REMOVED.equals(action)

&& intent.getBooleanExtra(Intent.EXTRA\_REPLACING, false)) {

// This package is being updated; don't kill its alarms.

return;

}

Uri data = intent.getData();

if (data != null) {

String pkg = data.getSchemeSpecificPart();

if (pkg != null) {

pkgList = new String[]{pkg};

}

}

}

if (pkgList != null && (pkgList.length > 0)) {

for (String pkg : pkgList) {

removeLocked(pkg, userHandle);

removePackagePrio(pkg, userHandle);

for (int i=mBroadcastStats.size()-1; i>=0; i--) {

ArrayMap<String, BroadcastStats> uidStats = mBroadcastStats.valueAt(i);

BroadcastStats stat = uidStats.get(pkg);

if (stat != null && UserHandle.getUserId(stat.mUid) == userHandle &&

stat.mPackageName.equals(pkg)) {

if (uidStats.remove(pkg) != null) {

if (uidStats.size() <= 0) {

mBroadcastStats.removeAt(i);

}

}

}

}

}

}

}

}

}

## AlarmThread

static final int TIME\_CHANGED\_MASK = 1 << 16;

// 上一次时间变化的时钟时间

long mLastTimeChangeClockTime;

// 上一次时间变化的开机时间

long mLastTimeChangeRealtime;

static final boolean WAKEUP\_STATS = false;

private class AlarmThread extends Thread

{

public AlarmThread()

{

super("AlarmManager");

}

public void run()

{

ArrayList<Alarm> triggerList = new ArrayList<Alarm>();

// 死循环

while (true)

{

// 等待驱动触发alarm的执行

int result = waitForAlarm(mNativeData);

triggerList.clear();

final long nowRTC = System.currentTimeMillis();

final long nowELAPSED = SystemClock.elapsedRealtime();

if ((result & TIME\_CHANGED\_MASK) != 0) {

// 由于内核内部小的调整可能导致内核给我们发送虚假的时间更改通知，所以这里要过滤出来。

final long lastTimeChangeClockTime;

final long expectedClockTime;

synchronized (mLock) {

lastTimeChangeClockTime = mLastTimeChangeClockTime;

expectedClockTime = lastTimeChangeClockTime

+ (nowELAPSED - mLastTimeChangeRealtime);

}

if (lastTimeChangeClockTime == 0 || nowRTC < (expectedClockTime-500)

|| nowRTC > (expectedClockTime+500)) {

// The change is by at least +/- 500 ms (or this is the first change),

// let's do it!

if (DEBUG\_BATCH) {

Slog.v(TAG, "Time changed notification from kernel; rebatching");

}

// 由于时间变化，所以要重新批处理所有的alarm

removeImpl(mTimeTickSender);

rebatchAllAlarms();

mClockReceiver.scheduleTimeTickEvent();

synchronized (mLock) {

mNumTimeChanged++;

mLastTimeChangeClockTime = nowRTC;

mLastTimeChangeRealtime = nowELAPSED;

}

// 发送时间变化的广播

Intent intent = new Intent(Intent.ACTION\_TIME\_CHANGED);

intent.addFlags(Intent.FLAG\_RECEIVER\_REPLACE\_PENDING

| Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY\_BEFORE\_BOOT);

getContext().sendBroadcastAsUser(intent, UserHandle.ALL);

// The world has changed on us, so we need to re-evaluate alarms

// regardless of whether the kernel has told us one went off.

result |= IS\_WAKEUP\_MASK;

}

}

if (result != TIME\_CHANGED\_MASK) {

// If this was anything besides just a time change, then figure what if

// anything to do about alarms.

synchronized (mLock) {

if (localLOGV) Slog.v(

TAG, "Checking for alarms... rtc=" + nowRTC

+ ", elapsed=" + nowELAPSED);

// 触发开始执行alarm，如果有wakeup类型的alarm则返回true

boolean hasWakeup = triggerAlarmsLocked(triggerList, nowELAPSED, nowRTC);

if (!hasWakeup && checkAllowNonWakeupDelayLocked(nowELAPSED)) {

// 如果是非wakeup类型的alarm且处于息屏状态，可以延迟执行alarm

if (mPendingNonWakeupAlarms.size() == 0) {

// 记录延迟执行的非wakeup的alarm的开始延迟的时间点

mStartCurrentDelayTime = nowELAPSED;

// 计算下一次非wakeup类型的alarm传递的时间

mNextNonWakeupDeliveryTime = nowELAPSED

+ ((currentNonWakeupFuzzLocked(nowELAPSED)\*3)/2);

}

// 把所有的alarm都添加到延迟执行列表中

mPendingNonWakeupAlarms.addAll(triggerList);

mNumDelayedAlarms += triggerList.size();

rescheduleKernelAlarmsLocked();

updateNextAlarmClockLocked();

} else {

// 现在传递alarm的intents，如果有延迟的非wakeup类型的alarm，则需要合并它们到列表中。

rescheduleKernelAlarmsLocked();

updateNextAlarmClockLocked();

if (mPendingNonWakeupAlarms.size() > 0) {

// 计算延迟执行alarm的传递优先级

calculateDeliveryPriorities(mPendingNonWakeupAlarms);

triggerList.addAll(mPendingNonWakeupAlarms);

Collections.sort(triggerList, mAlarmDispatchComparator);

final long thisDelayTime = nowELAPSED - mStartCurrentDelayTime;

mTotalDelayTime += thisDelayTime;

if (mMaxDelayTime < thisDelayTime) {

mMaxDelayTime = thisDelayTime;

}

mPendingNonWakeupAlarms.clear();

}

// 执行alarm

deliverAlarmsLocked(triggerList, nowELAPSED);

}

}

}

}

}

}

void removeImpl(PendingIntent operation) {

if (operation == null) {

return;

}

synchronized (mLock) {

// 移除掉alarm的PendingIntent

removeLocked(operation);

}

}

/\*\*

\* key是uid，value是alarm的上一次执行时间

\*/

final SparseLongArray mLastAllowWhileIdleDispatch = new SparseLongArray();

// 可以开始执行flag为ALLOW\_WHILE\_IDLE的alarm的最小时间间隔

long mAllowWhileIdleMinTime;

int mCurrentSeq = 0;

// 执行alarm

boolean triggerAlarmsLocked(ArrayList<Alarm> triggerList, final long nowELAPSED,

final long nowRTC) {

boolean hasWakeup = false;

while (mAlarmBatches.size() > 0) {

// 从列表中取出第一个批处理对象

Batch batch = mAlarmBatches.get(0);

// 如果第一个批处理对象的开始时间大于手机开机时间，说明当前还没有要开始执行的alarm，那么直接结束循环

if (batch.start > nowELAPSED) {

// Everything else is scheduled for the future

break;

}

// 下面要处理这个批处理对象了，所以这里将该批处理对象从列表中移除，以避免其他地方访问该批处理对象

mAlarmBatches.remove(0);

// 获取批处理对象中Alarm的数量

final int N = batch.size();

for (int i = 0; i < N; i++) {

Alarm alarm = batch.get(i);

// 判断alarm是否设置了空闲时允许执行任务的flag

if ((alarm.flags&AlarmManager.FLAG\_ALLOW\_WHILE\_IDLE) != 0) {

// 如果该alarm设置了ALLOW\_WHILE\_IDLE，则需要限制应用程序调度这些alarm的频率，这里是增加了执行间隔时间。

// 根据uid获取alarm的上一次开始执行时间

long lastTime = mLastAllowWhileIdleDispatch.get(alarm.uid, 0);

// alarm下一次最早开始执行的时间：上一次开始执行时间+最小时间间隔

long minTime = lastTime + mAllowWhileIdleMinTime;

// 判断当前开机时间是否小于alarm的下一次最早开始执行时间

if (nowELAPSED < minTime) {

// 自上次执行完此alarm后还没有足够的时间以再次开始执行。所以要重新调度该alarm的正确开始执行时间。

// 把下一次的最早开始执行时间赋值给alarm的whenElapsed变量

alarm.whenElapsed = minTime;

// 判断alarm的最迟开始执行时间是否小于alarm的下一次最早开始执行时间

if (alarm.maxWhenElapsed < minTime) {

alarm.maxWhenElapsed = minTime;

}

// 由于alarm相关的参数发生变化，所以需要重新把该alarm添加到相关列表中

setImplLocked(alarm, true, false);

continue;

}

}

// 设置alarm的执行次数为1

alarm.count = 1;

// 如果alarm不需要重新调度，则就把该alarm添加到triggerList列表中

triggerList.add(alarm);

if ((alarm.flags&AlarmManager.FLAG\_WAKE\_FROM\_IDLE) != 0) {

EventLogTags.writeDeviceIdleWakeFromIdle(mPendingIdleUntil != null ? 1 : 0,

alarm.tag);

}

// 判断该alarm是否是延迟到空闲时开始执行的alarm

if (mPendingIdleUntil == alarm) {

// 因为该alarm就要开始执行，所以要把mPendingIdleUntil置为null，后面会根据各alarm的开始执行时间重新赋值

mPendingIdleUntil = null;

// 由于mPendingIdleUntil变量改变，所以需要对所有的alarm进行重新批处理操作

rebatchAllAlarmsLocked(false);

// 由于mPendingIdleUntil变量改变，所以需要重新存储空闲时延迟执行的alarm

restorePendingWhileIdleAlarmsLocked();

}

// 判断该alarm是否是下一个请求退出空闲模式的alarm

if (mNextWakeFromIdle == alarm) {

// 因为该alarm就要开始执行，所以要把mNextWakeFromIdle置为null，后面会根据各alarm的执行时间重新赋值

mNextWakeFromIdle = null;

// 对所有的alarm进行重新批处理操作

rebatchAllAlarmsLocked(false);

}

if (alarm.repeatInterval > 0) {

// 计算重复执行的alarm从开始执行到现在的执行次数

alarm.count += (nowELAPSED - alarm.whenElapsed) / alarm.repeatInterval;

// 计算alarm的下一次开始执行时间

final long delta = alarm.count \* alarm.repeatInterval;

final long nextElapsed = alarm.whenElapsed + delta;

// 根据新得到的alarm参数，设置下次触发的alarm

setImplLocked(alarm.type, alarm.when + delta, nextElapsed, alarm.windowLength,

maxTriggerTime(nowELAPSED, nextElapsed, alarm.repeatInterval),

alarm.repeatInterval, alarm.operation, alarm.flags, true,

alarm.workSource, alarm.alarmClock, alarm.uid);

}

if (alarm.wakeup) {

hasWakeup = true;

}

// We removed an alarm clock. Let the caller recompute the next alarm clock.

if (alarm.alarmClock != null) {

mNextAlarmClockMayChange = true;

}

}

}

// This is a new alarm delivery set; bump the sequence number to indicate that

// all apps' alarm delivery classes should be recalculated.

// 这是一个新的alarm传递集合,序列号表示所有应用程序的alarm传递等级都需要重新计算

mCurrentSeq++;

// 重新计算alarm的传递优先级

calculateDeliveryPriorities(triggerList);

// 根据优先级排序

Collections.sort(triggerList, mAlarmDispatchComparator);

if (localLOGV) {

for (int i=0; i<triggerList.size(); i++) {

Slog.v(TAG, "Triggering alarm #" + i + ": " + triggerList.get(i));

}

}

return hasWakeup;

}

// alarm传递优先级定义

static final int PRIO\_TICK = 0;

static final int PRIO\_WAKEUP = 1;

static final int PRIO\_NORMAL = 2;

// 计算alarm的传递优先级

void calculateDeliveryPriorities(ArrayList<Alarm> alarms) {

final int N = alarms.size();

for (int i = 0; i < N; i++) {

Alarm a = alarms.get(i);

final int alarmPrio;

if (Intent.ACTION\_TIME\_TICK.equals(a.operation.getIntent().getAction())) {

alarmPrio = PRIO\_TICK;

} else if (a.wakeup) {

alarmPrio = PRIO\_WAKEUP;

} else {

alarmPrio = PRIO\_NORMAL;

}

PriorityClass packagePrio = a.priorityClass;

if (packagePrio == null) {

packagePrio = a.priorityClass = new PriorityClass(); // lowest prio & stale sequence

}

a.priorityClass = packagePrio;

if (packagePrio.seq != mCurrentSeq) {

// first alarm we've seen in the current delivery generation from this package

packagePrio.priority = alarmPrio;

packagePrio.seq = mCurrentSeq;

} else {

// Multiple alarms from this package being delivered in this generation;

// bump the package's delivery class if it's warranted.

// TICK < WAKEUP < NORMAL

if (alarmPrio < packagePrio.priority) {

packagePrio.priority = alarmPrio;

}

}

}

}

final class PriorityClass {

int seq;

int priority;

PriorityClass() {

seq = mCurrentSeq - 1;

priority = PRIO\_NORMAL;

}

}

// 根据优先级排序，优先级相同的则根据开始执行时间排序

final Comparator<Alarm> mAlarmDispatchComparator = new Comparator<Alarm>() {

@Override

public int compare(Alarm lhs, Alarm rhs) {

// priority class trumps everything. TICK < WAKEUP < NORMAL

if (lhs.priorityClass.priority < rhs.priorityClass.priority) {

return -1;

} else if (lhs.priorityClass.priority > rhs.priorityClass.priority) {

return 1;

}

// within each class, sort by nominal delivery time

if (lhs.whenElapsed < rhs.whenElapsed) {

return -1;

} else if (lhs.whenElapsed > rhs.whenElapsed) {

return 1;

}

// same priority class + same target delivery time

return 0;

}

};

private native int waitForAlarm(long nativeData);

// 检查是否允许非wakeup类型的alarm延迟执行

boolean checkAllowNonWakeupDelayLocked(long nowELAPSED) {

/\*由于非wakeup的alarm在息屏状态可能会有延迟，若不希望此情况发生，可直接在这里返回false\*/

// 如果当前是亮屏状态，则返回false

if (mInteractive) {

return false;

}

if (mLastAlarmDeliveryTime <= 0) {

return false;

}

if (mPendingNonWakeupAlarms.size() > 0 && mNextNonWakeupDeliveryTime < nowELAPSED) {

// This is just a little paranoia, if somehow we have pending non-wakeup alarms

// and the next delivery time is in the past, then just deliver them all. This

// avoids bugs where we get stuck in a loop trying to poll for alarms.

return false;

}

long timeSinceLast = nowELAPSED - mLastAlarmDeliveryTime;

// 判断上次alarm传递的时间是否小于延迟时间

return timeSinceLast <= currentNonWakeupFuzzLocked(nowELAPSED);

}

// 根据息屏状态持续的时间决定延迟的时间

long currentNonWakeupFuzzLocked(long nowELAPSED) {

// 息屏状态持续时间=当前时间-开始息屏的时间

long timeSinceOn = nowELAPSED - mNonInteractiveStartTime;

if (timeSinceOn < 5\*60\*1000) {

// If the screen has been off for 5 minutes, only delay by at most two minutes.

return 2\*60\*1000;

} else if (timeSinceOn < 30\*60\*1000) {

// If the screen has been off for 30 minutes, only delay by at most 15 minutes.

return 15\*60\*1000;

} else {

// Otherwise, we will delay by at most an hour.

return 60\*60\*1000;

}

}

private final IAlarmManager mService;

private final IBinder mService = new IAlarmManager.Stub() {

@Override

public void set(int type, long triggerAtTime, long windowLength, long interval, int flags,

PendingIntent operation, WorkSource workSource,

AlarmManager.AlarmClockInfo alarmClock) {

final int callingUid = Binder.getCallingUid();

if (workSource != null) {

getContext().enforcePermission(

android.Manifest.permission.UPDATE\_DEVICE\_STATS,

Binder.getCallingPid(), callingUid, "AlarmManager.set");

}

// No incoming callers can request either WAKE\_FROM\_IDLE or

// ALLOW\_WHILE\_IDLE\_UNRESTRICTED -- we will apply those later as appropriate.

flags &= ~(AlarmManager.FLAG\_WAKE\_FROM\_IDLE

| AlarmManager.FLAG\_ALLOW\_WHILE\_IDLE\_UNRESTRICTED);

// Only the system can use FLAG\_IDLE\_UNTIL -- this is used to tell the alarm

// manager when to come out of idle mode, which is only for DeviceIdleController.

if (callingUid != Process.SYSTEM\_UID) {

flags &= ~AlarmManager.FLAG\_IDLE\_UNTIL;

}

// If the caller is a core system component, and not calling to do work on behalf

// of someone else, then always set ALLOW\_WHILE\_IDLE\_UNRESTRICTED. This means we

// will allow these alarms to go off as normal even while idle, with no timing

// restrictions.

if (callingUid < Process.FIRST\_APPLICATION\_UID && workSource == null) {

flags |= AlarmManager.FLAG\_ALLOW\_WHILE\_IDLE\_UNRESTRICTED;

}

// If this is an exact time alarm, then it can't be batched with other alarms.

if (windowLength == AlarmManager.WINDOW\_EXACT) {

flags |= AlarmManager.FLAG\_STANDALONE;

}

// If this alarm is for an alarm clock, then it must be standalone and we will

// use it to wake early from idle if needed.

if (alarmClock != null) {

flags |= AlarmManager.FLAG\_WAKE\_FROM\_IDLE | AlarmManager.FLAG\_STANDALONE;

}

// 调用AlarmManagerService类的setImpl方法设置alarm

setImpl(type, triggerAtTime, windowLength, interval, operation,

flags, workSource, alarmClock, callingUid);

}

@Override

public boolean setTime(long millis) {

getContext().enforceCallingOrSelfPermission(

"android.permission.SET\_TIME",

"setTime");

if (mNativeData == 0) {

Slog.w(TAG, "Not setting time since no alarm driver is available.");

return false;

}

synchronized (mLock) {

return setKernelTime(mNativeData, millis) == 0;

}

}

@Override

public void setTimeZone(String tz) {

getContext().enforceCallingOrSelfPermission(

"android.permission.SET\_TIME\_ZONE",

"setTimeZone");

final long oldId = Binder.clearCallingIdentity();

try {

setTimeZoneImpl(tz);

} finally {

Binder.restoreCallingIdentity(oldId);

}

}

@Override

public void remove(PendingIntent operation) {

removeImpl(operation);

}

@Override

public long getNextWakeFromIdleTime() {

return getNextWakeFromIdleTimeImpl();

}

@Override

public AlarmManager.AlarmClockInfo getNextAlarmClock(int userId) {

userId = ActivityManager.handleIncomingUser(Binder.getCallingPid(),

Binder.getCallingUid(), userId, false /\* allowAll \*/, false /\* requireFull \*/,

"getNextAlarmClock", null);

return getNextAlarmClockImpl(userId);

}

@Override

protected void dump(FileDescriptor fd, PrintWriter pw, String[] args) {

. . .

}

@Override

/\* updates the blocked uids, so if a wake lock is acquired to only fire

\* alarm for it, it can be released.

\*/

public void updateBlockedUids(int uid, boolean isBlocked) {

if (localLOGV) Slog.v(TAG, "UpdateBlockedUids: uid = " + uid +

" isBlocked = " + isBlocked);

if (Binder.getCallingUid() != Process.SYSTEM\_UID) {

if (localLOGV) Slog.v(TAG, "UpdateBlockedUids is not allowed");

return;

}

synchronized(mLock) {

if(isBlocked) {

mBlockedUids.add(new Integer(uid));

if (checkReleaseWakeLock()) {

/\* all the uids for which the alarms are triggered

\* are either blocked or have called onSendFinished.

\*/

if (mWakeLock.isHeld()) {

mWakeLock.release();

if (localLOGV)

Slog.v(TAG, "AM WakeLock Released Internally in updateBlockedUids");

}

}

} else {

mBlockedUids.clear();

}

}

}

};

SystemService类中的publishBinderService方法：

/\*\*

\* 发布服务，以便其他服务或应用程序能调用该服务。

\*/

protected final void publishBinderService(String name, IBinder service) {

publishBinderService(name, service, false);

}

/\*\*

\* 发布服务，以便其他服务或应用程序能调用该服务。

\*/

protected final void publishBinderService(String name, IBinder service,

boolean allowIsolated) {

// 把服务添加到ServiceManager中，以便其他服务或应用程序能够调用该服务

// getSystemService时就是根据name获取到该服务的

ServiceManager.addService(name, service, allowIsolated);

## AlarmHandler

版权声明：本文为CSDN博主「张永飞」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/zhangyongfeiyong/article/details/52224300>

final Object mLock = new Object();

private class AlarmHandler extends Handler {

public static final int ALARM\_EVENT = 1;

public static final int MINUTE\_CHANGE\_EVENT = 2;

public static final int DATE\_CHANGE\_EVENT = 3;

public static final int SEND\_NEXT\_ALARM\_CLOCK\_CHANGED = 4;

public AlarmHandler() {

}

public void handleMessage(Message msg) {

if (msg.what == ALARM\_EVENT) {

ArrayList<Alarm> triggerList = new ArrayList<Alarm>();

synchronized (mLock) {

final long nowRTC = System.currentTimeMillis();

final long nowELAPSED = SystemClock.elapsedRealtime();

// 触发开始执行alarm

triggerAlarmsLocked(triggerList, nowELAPSED, nowRTC);

// 更新下一个alarm的alarmClock

updateNextAlarmClockLocked();

}

// now trigger the alarms without the lock held

for (int i=0; i<triggerList.size(); i++) {

Alarm alarm = triggerList.get(i);

try {

// 执行和PendingIntent关联的操作

alarm.operation.send();

} catch (PendingIntent.CanceledException e) {

if (alarm.repeatInterval > 0) {

// 移除掉alarm的PendingIntent

removeImpl(alarm.operation);

}

}

}

} else if (msg.what == SEND\_NEXT\_ALARM\_CLOCK\_CHANGED) {

// 处理下一个alarm的开始执行时间改变的逻辑

sendNextAlarmClockChanged();

}

}

}

private final SparseArray<AlarmManager.AlarmClockInfo> mHandlerSparseAlarmClockArray =

new SparseArray<>();

private static final Intent NEXT\_ALARM\_CLOCK\_CHANGED\_INTENT = new Intent(

AlarmManager.ACTION\_NEXT\_ALARM\_CLOCK\_CHANGED);

// 处理下一个alarm的开始执行时间改变的逻辑

private void sendNextAlarmClockChanged() {

SparseArray<AlarmManager.AlarmClockInfo> pendingUsers = mHandlerSparseAlarmClockArray;

pendingUsers.clear();

synchronized (mLock) {

final int N = mPendingSendNextAlarmClockChangedForUser.size();

// 循环把mPendingSendNextAlarmClockChangedForUser中的值添加到pendingUsers中

for (int i = 0; i < N; i++) {

int userId = mPendingSendNextAlarmClockChangedForUser.keyAt(i);

pendingUsers.append(userId, mNextAlarmClockForUser.get(userId));

}

mPendingSendNextAlarmClockChangedForUser.clear();

}

final int N = pendingUsers.size();

for (int i = 0; i < N; i++) {

int userId = pendingUsers.keyAt(i);

AlarmManager.AlarmClockInfo alarmClock = pendingUsers.valueAt(i);

Settings.System.putStringForUser(getContext().getContentResolver(),

Settings.System.NEXT\_ALARM\_FORMATTED,

// 格式化时间

formatNextAlarm(getContext(), alarmClock, userId),

userId);

// 发送广播，接收广播的是状态栏

getContext().sendBroadcastAsUser(NEXT\_ALARM\_CLOCK\_CHANGED\_INTENT,

new UserHandle(userId));

}

}

/\*\*

\* Formats an alarm like platform/packages/apps/DeskClock used to.

\*/

private static String formatNextAlarm(final Context context, AlarmManager.AlarmClockInfo info,

int userId) {

String skeleton = DateFormat.is24HourFormat(context, userId) ? "EHm" : "Ehma";

String pattern = DateFormat.getBestDateTimePattern(Locale.getDefault(), skeleton);

return (info == null) ? "" :

DateFormat.format(pattern, info.getTriggerTime()).toString();

}

————————————————

版权声明：本文为CSDN博主「张永飞」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/zhangyongfeiyong/article/details/52224300

# native层

frameworks/base/services/core/jni/com\_android\_server\_AlarmManagerService.cpp

## init

ALOGV为何不打印呢？

**static** jlong android\_server\_AlarmManagerService\_init(JNIEnv\*, jobject)  
{  
 jlong ret = init\_alarm\_driver();  
 **if** (ret) {  
 **return** ret;  
 }  
  
 **return** init\_timerfd();  
}

**static** jlong init\_alarm\_driver()  
{  
 **int** fd = open(**"/dev/alarm"**, O\_RDWR);  
 **if** (fd < 0) {  
 ALOGV(**"opening alarm driver failed: %s"**, strerror(errno));  
 **return** 0;  
 }  
  
 AlarmImpl \*ret = **new** AlarmImplAlarmDriver(fd);  
 **return reinterpret\_cast**<jlong>(ret);  
}

## waitForAlarm

**static** jint android\_server\_AlarmManagerService\_waitForAlarm(JNIEnv\*, jobject, jlong nativeData)  
{  
 AlarmImpl \*impl = **reinterpret\_cast**<AlarmImpl \*>(nativeData);  
 **int** result = 0;  
  
 **do** {  
 result = impl->waitForAlarm();  
 } **while** (result < 0 && errno == EINTR);  
  
 **if** (result < 0)  
 {  
 ALOGE(**"Unable to wait on alarm: %s\n"**, strerror(errno));  
 **return** 0;  
 }  
  
 **return** result;  
}

附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/zhangyongfeiyong/article/details/52224300>

————————————————

版权声明：本文为CSDN博主「张永飞」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/zhangyongfeiyong/article/details/52224300

# 时区

拿到GPS为何能够自动更新时区原理

# 时间设置ACTION\_TIME\_CHANGED

AlarmThread等待waitForAlarm事件

**while** (**true**)  
{  
 **int** result = waitForAlarm(**mNativeData**);

**if** ((result & ***TIME\_CHANGED\_MASK***) != 0)

# setTime

**final long** nowRTC = System.*currentTimeMillis*();

<https://juejin.im/post/5d70d0236fb9a06af824fb3f>

**final long** nowELAPSED = SystemClock.*elapsedRealtime*();

shell命令执行

date 0829010207.03

# App

监听时间变化<https://www.jianshu.com/p/ddbb8c386e38>

ntentFilter = new IntentFilter();

intentFilter.addAction(Intent.ACTION\_TIME\_TICK);//每分钟变化

intentFilter.addAction(Intent.ACTION\_TIMEZONE\_CHANGED);//设置了系统时区

intentFilter.addAction(Intent.ACTION\_TIME\_CHANGED);//设置了系统时间

无需在AndroidManifest.xml中做任何改动？？？

# 参考

Android之AlarmManagerService源码分析

<http://blog.csdn.net/zhangyongfeiyong/article/details/52224300>