**Android中Alarm的机制**

2015年03月16日 11:27:12 [wh\_19910525](https://me.csdn.net/wh_19910525) 阅读数：11575更多 个人分类： [android-源码开发](https://blog.csdn.net/wh_19910525/article/category/1302634)

本次给大家分析的是Android中Alarm的机制所用源码为最新的Android4.4.4。首先简单介绍如何使用Alarm并给出其工作原理，接着分析Alarm和Timer以及Handler在完成定时任务上的差别，最后分析Alarm机制的源码。

**什么是Alarm**

Alarm是android提供的用于完成闹钟式定时任务的类，系统通过AlarmManager来管理所有的Alarm，Alarm支持一次性定时任务和循环定时任务，它的使用方式很简单，这里不多做介绍，只给出一个简单的示例：

**[java]** [view plaincopy](http://blog.csdn.net/singwhatiwanna/article/details/18448997)

1. AlarmManager alarmMgr = (AlarmManager) getSystemService(Context.ALARM\_SERVICE);
2. Intent intent = **new** Intent(getApplicationContext(), TestActivity.**class**);
3. PendingIntent pendIntent = PendingIntent.getActivity(getApplicationContext(),
4. 0, intent, PendingIntent.FLAG\_UPDATE\_CURRENT);
5. //5秒后发送广播，只发送一次
6. **int** triggerAtTime = SystemClock.elapsedRealtime() + 5 \* 1000;
7. alarmMgr.set(AlarmManager.ELAPSED\_REALTIME, triggerAtTime, pendIntent);

**Alarm和Timer以及Handler在定时任务上的区别**

**相同点**：

三者都可以完成定时任务，都支持一次性定时和循环定时（注：Handler可以间接支持循环定时任务）

**不同点**：

Handler和Timer在定时上是类似的，二者在系统休眠的情况下无法正常工作，定时任务不会按时触发。Alarm在系统休眠的情况下可以正常工作，并且还可以决定是否唤醒系统，同时Alarm在自身不启动的情况下仍能正常收到定时任务提醒，但是当系统重启或者应用被杀死的情况下，Alarm定时任务会被取消。另外，从Android4.4开始，Alarm事件默认采用非精准方式，即定时任务可能会有小范围的提前或延后，当然我们可以强制采用精准方式，而在此之前，Alarm事件都是精准方式。

**Alarm与Binder的交互**

Alarm由AlarmManager来管理，从使用方式来看，AlarmManager很简单，我们只要得到了AlarmManager的对象，就可以调用set方法来设定定时任务了，而如何得到AlarmManager对象呢？也很简单，AlarmManager alarmMgr = (AlarmManager) getSystemService(Context.ALARM\_SERVICE)；下面我们去看看AlarmManager的set方法，当然AlarmManager还有setRepeating方法，但是二者是类似的。为了更好地理解下面的内容，需要你了解AIDL，如果你还不了解，请参看[android跨进程通信(IPC):使用AIDL](http://blog.csdn.net/singwhatiwanna/article/details/17041691)。

code：AlarmManager#set

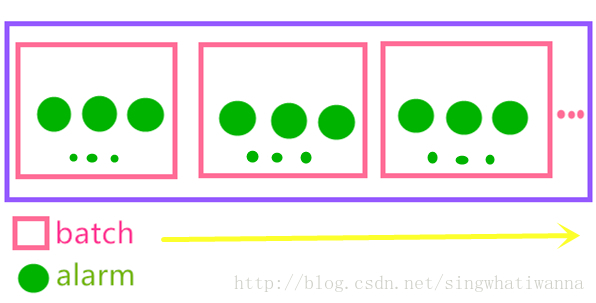
**[java]** [view plaincopy](http://blog.csdn.net/singwhatiwanna/article/details/18448997)

1. **public** **void** set(**int** type, **long** triggerAtMillis, PendingIntent operation) {
2. setImpl(type, triggerAtMillis, legacyExactLength(), 0, operation, **null**);
3. }
5. **public** **void** set(**int** type, **long** triggerAtMillis, **long** windowMillis, **long** intervalMillis,
6. PendingIntent operation, WorkSource workSource) {
7. setImpl(type, triggerAtMillis, windowMillis, intervalMillis, operation, workSource);
8. }
10. **private** **void** setImpl(**int** type, **long** triggerAtMillis, **long** windowMillis, **long** intervalMillis,
11. PendingIntent operation, WorkSource workSource) {
12. **if** (triggerAtMillis < 0) {
13. /\* NOTYET
14. if (mAlwaysExact) {
15. // Fatal error for KLP+ apps to use negative trigger times
16. throw new IllegalArgumentException("Invalid alarm trigger time "
17. + triggerAtMillis);
18. }
19. \*/
20. triggerAtMillis = 0;
21. }
23. **try** {
24. //定时任务实际上都有mService来完成，也就是说AlarmManager只是一个空壳
25. //从下面的构造方法可以看出，这个mService是IAlarmManager类型的，而IAlarmManager是一个接口
26. //如果大家了解AIDL就应该知道IAlarmManager应该是一个AIDL接口
27. mService.set(type, triggerAtMillis, windowMillis, intervalMillis, operation,
28. workSource);
29. } **catch** (RemoteException ex) {
30. }
31. }
33. AlarmManager(IAlarmManager service, Context ctx) {
34. mService = service;
36. **final** **int** sdkVersion = ctx.getApplicationInfo().targetSdkVersion;
37. mAlwaysExact = (sdkVersion < Build.VERSION\_CODES.KITKAT);
38. }

说明：我对代码进行了注释，从注释可以看出，现在我们需要去找到这个mService，其实我已经帮大家找到了，它就是AlarmManagerService

**Alarm机制分析**

通过上面的一系列分析，我们知道AlarmManager的所有功能都是通过AlarmManagerService来完成的，在分析源码之前，我先来描述下Alarm的工作原理：从Android4.4开始，Alarm默认为非精准模式，除非显示指定采用精准模式。在非精准模式下，Alarm是批量提醒的，每个alarm根据其触发时间和最大触发时间的不同会被加入到不同的batch中，同一个batch的不同alarm是同时发生的，这样就无法实现精准闹钟，官方的解释是批量处理可以减少设备被唤醒次数以及节约电量，不过针对精准闹钟，官方预留的方法是setExact和setWindow，二者都是通过将时间窗口定义为0来实现精准闹钟的，因为时间窗口为0，意味着触发时间和最大触发时间是一样的，因为典型的情况下：最大触发时间= 触发时间 + 时间窗口。同时所有的batch是按开始时间升序排列的，在一个batch内部，不同的闹钟也是按触发时间升序排列的，所以闹钟的唤醒顺序是按照batch的排序依次触发的，而同一个batch中的alarm是同时触发的，可以用下面这个示意图来描述：



上图是示意图，系统中可以有多个batch，每个batch中可以有多个alarm。下面我们分析一下AlarmManagerService中的代码。其入口方法为set，set又调用了setImplLocked，所以我们直接看setImplLocked。

code：AlarmManagerService#setImplLocked

**[java]** [view plaincopy](http://blog.csdn.net/singwhatiwanna/article/details/18448997)

1. **private** **void** setImplLocked(**int** type, **long** when, **long** whenElapsed, **long** maxWhen, **long** interval,
2. PendingIntent operation, **boolean** isStandalone, **boolean** doValidate,
3. WorkSource workSource) {
4. /\*\*创建一个alarm，其中各参数的含义如下：
5. \* type 闹钟类型 ELAPSED\_REALTIME、RTC、RTC\_WAKEUP等
6. \* when 触发时间 UTC类型，绝对时间，通过System.currentTimeMillis()得到
7. \* whenElapsed 相对触发时间，自开机算起，含休眠，通过SystemClock.elapsedRealtime()得到
8. \* maxWhen 最大触发时间
9. \* interval 触发间隔，针对循环闹钟有效
10. \* operation 闹钟触发时的行为，PendingIntent类型
11. \*/
12. Alarm a = **new** Alarm(type, when, whenElapsed, maxWhen, interval, operation, workSource);
13. //根据PendingIntent删除之前已有的同一个闹钟
14. removeLocked(operation);
16. **boolean** reschedule;
17. //尝试将alarm加入到合适的batch中，如果alarm是独立的或者无法找到合适的batch去容纳此alarm，返回-1
18. **int** whichBatch = (isStandalone) ? -1 : attemptCoalesceLocked(whenElapsed, maxWhen);
19. **if** (whichBatch < 0) {
20. //没有合适的batch去容纳alarm，则新建一个batch
21. Batch batch = **new** Batch(a);
22. batch.standalone = isStandalone;
23. //将batch加入mAlarmBatches中，并对mAlarmBatches进行排序：按开始时间升序排列
24. reschedule = addBatchLocked(mAlarmBatches, batch);
25. } **else** {
26. //如果找到合适了batch去容纳此alarm，则将其加入到batch中
27. Batch batch = mAlarmBatches.get(whichBatch);
28. //如果当前alarm的加入引起了batch开始时间和结束时间的改变，则reschedule为true
29. reschedule = batch.add(a);
30. **if** (reschedule) {
31. //由于batch的起始时间发生了改变，所以需要从列表中删除此batch并重新加入、重新对batch列表进行排序
32. mAlarmBatches.remove(whichBatch);
33. addBatchLocked(mAlarmBatches, batch);
34. }
35. }
37. **if** (DEBUG\_VALIDATE) {
38. **if** (doValidate && !validateConsistencyLocked()) {
39. Slog.v(TAG, "Tipping-point operation: type=" + type + " when=" + when
40. + " when(hex)=" + Long.toHexString(when)
41. + " whenElapsed=" + whenElapsed + " maxWhen=" + maxWhen
42. + " interval=" + interval + " op=" + operation
43. + " standalone=" + isStandalone);
44. rebatchAllAlarmsLocked(**false**);
45. reschedule = **true**;
46. }
47. }
49. **if** (reschedule) {
50. **rescheduleKernelAlarmsLocked**();
51. }
52. }

说明：通过上述代码可以看出，当我们创建一个alarm的时候，仅仅是将这个alarm加入到某个batch中，系统中有一个batch列表，专门用于存储所有的alarm。可是仅仅把alarm加入到batch中还不行，系统还必须提供一个类似于Looper的东西一直去遍历这个列表，一旦它发现有些alarm的时间已经到达就要把它取出来去执行。事实上，AlarmManagerService中的确有一个类似于Looper的东西去干这个事情，只不过它是个线程，叫做AlarmThread。下面看它的代码：

code：AlarmManagerService#AlarmThread

**[java]** [view plaincopy](http://blog.csdn.net/singwhatiwanna/article/details/18448997)

1. **private** **class** AlarmThread **extends** Thread
2. {
3. **public** AlarmThread()
4. {
5. **super**("AlarmManager");
6. }
8. **public** **void** run()
9. {
10. //当前时间触发的alarm列表
11. ArrayList<Alarm> triggerList = **new** ArrayList<Alarm>();
13. **while** (**true**)
14. {
15. //jni方法，顾名思义，阻塞式方法，当有alarm的时候会被唤醒
16. **int** result = waitForAlarm(mDescriptor);
18. triggerList.clear();
20. **if** ((result & TIME\_CHANGED\_MASK) != 0) {
21. **if** (DEBUG\_BATCH) {
22. Slog.v(TAG, "Time changed notification from kernel; rebatching");
23. }
24. remove(mTimeTickSender);
25. //将所有的alarm重新排序
26. rebatchAllAlarms();
27. mClockReceiver.scheduleTimeTickEvent();
28. Intent intent = **new** Intent(Intent.ACTION\_TIME\_CHANGED);
29. intent.addFlags(Intent.FLAG\_RECEIVER\_REPLACE\_PENDING
30. | Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY\_BEFORE\_BOOT);
31. mContext.sendBroadcastAsUser(intent, UserHandle.ALL);
32. }
34. **synchronized** (mLock) {
35. **final** **long** nowRTC = System.currentTimeMillis();
36. **final** **long** nowELAPSED = SystemClock.elapsedRealtime();
37. **if** (localLOGV) Slog.v(
38. TAG, "Checking for alarms... rtc=" + nowRTC
39. + ", elapsed=" + nowELAPSED);
41. **if** (WAKEUP\_STATS) {
42. **if** ((result & IS\_WAKEUP\_MASK) != 0) {
43. **long** newEarliest = nowRTC - RECENT\_WAKEUP\_PERIOD;
44. **int** n = 0;
45. **for** (WakeupEvent event : mRecentWakeups) {
46. **if** (event.when > newEarliest) **break**;
47. n++; // number of now-stale entries at the list head
48. }
49. **for** (**int** i = 0; i < n; i++) {
50. mRecentWakeups.remove();
51. }
53. recordWakeupAlarms(mAlarmBatches, nowELAPSED, nowRTC);
54. }
55. }
56. //这个方法会把batch列表中的第一个batch取出来然后加到触发列表中
57. //当然，前提是此batch的开始时间不大于当前时间
58. //同时，如果是循环闹钟，则会对下次任务进行再次定时
59. triggerAlarmsLocked(triggerList, nowELAPSED, nowRTC);
60. rescheduleKernelAlarmsLocked();
62. // 遍历触发列表，发送PendingIntent
63. **for** (**int** i=0; i<triggerList.size(); i++) {
64. Alarm alarm = triggerList.get(i);
65. **try** {
66. **if** (localLOGV) Slog.v(TAG, "sending alarm " + alarm);
67. //这里PendingIntent会被send，结果就是我们的定时任务被执行了
68. alarm.operation.send(mContext, 0,
69. mBackgroundIntent.putExtra(
70. Intent.EXTRA\_ALARM\_COUNT, alarm.count),
71. mResultReceiver, mHandler);
73. // we have an active broadcast so stay awake.
74. **if** (mBroadcastRefCount == 0) {
75. setWakelockWorkSource(alarm.operation, alarm.workSource);
76. mWakeLock.acquire();
77. }
78. **final** InFlight inflight = **new** InFlight(AlarmManagerService.**this**,
79. alarm.operation, alarm.workSource);
80. mInFlight.add(inflight);
81. mBroadcastRefCount++;
83. **final** BroadcastStats bs = inflight.mBroadcastStats;
84. bs.count++;
85. **if** (bs.nesting == 0) {
86. bs.nesting = 1;
87. bs.startTime = nowELAPSED;
88. } **else** {
89. bs.nesting++;
90. }
91. **final** FilterStats fs = inflight.mFilterStats;
92. fs.count++;
93. **if** (fs.nesting == 0) {
94. fs.nesting = 1;
95. fs.startTime = nowELAPSED;
96. } **else** {
97. fs.nesting++;
98. }
99. **if** (alarm.type == ELAPSED\_REALTIME\_WAKEUP
100. || alarm.type == RTC\_WAKEUP) {
101. bs.numWakeup++;
102. fs.numWakeup++;
103. //针对能唤醒设备的闹钟，这里会做一些唤醒设备的事情
104. ActivityManagerNative.noteWakeupAlarm(
105. alarm.operation);
106. }
107. } **catch** (PendingIntent.CanceledException e) {
108. **if** (alarm.repeatInterval > 0) {
109. // This IntentSender is no longer valid, but this
110. // is a repeating alarm, so toss the hoser.
111. remove(alarm.operation);
112. }
113. } **catch** (RuntimeException e) {
114. Slog.w(TAG, "Failure sending alarm.", e);
115. }
116. }
117. }
118. }
119. }
120. }

说明：上述代码中，AlarmThread会一直循环的跑着，一旦有新的alarm触发，它就会取出一个batch然后逐个发送PendingIntent，具体alarm的触发是由底层来完成的，我没法再继续分析下去。还有就是Alarm中有一些细节，我没有进行很具体的分析，实际上很简单，大家一看就懂。到此为止，Alarm机制的主要流程也分析完了。

**总结**

本文没有详细介绍如何使用Alarm，因为很简单，看一下官方文档或者网上搜一下，到处都是。关于Alarm，有一点需要强调一下：当手机重启或者应用被杀死的时候，Alarm会被删除，因此，如果想通过Alarm来完成长久定时任务是不可靠的，如果非要完成长久定时任务，可以这样：将应用的所有Alarm信息存到数据库中，每次应用启动的时候都重新注册Alarm并更新Alarm的触发时间，通过这种方式就不存在Alarm丢失的情况了。本文很长，耗时8个小时才完成的，感谢大家阅读本文，希望本文能给大家带来一点帮助。

# Android Alarm驱动源代码分析（Alarm.c）

2017年07月05日 22:44:53 [xiaopohaibebo](https://me.csdn.net/xiaopohaibebo) 阅读数：394

**前言：**

[**Android**](http://lib.csdn.net/base/android)在[**Linux**](http://lib.csdn.net/base/linux) Kernel的基础上增加了很多的驱动程序，Alarm驱动是其中最简单的一个，整个文件只有500多行。作为驱动代码分析的一系列文章的开始，我试图仔细的分析此驱动的几乎所有函数代码，希望籍此作为温习[**android**](http://lib.csdn.net/base/android)驱动源代码一个良好的开端。

Android的增加了一个Alarm驱动，在kernel\_root/driver/rtc/alarm.c文件中实现。Android希望提供一种递增的时钟（monotonic ），此时钟应基于硬件，使得Android的应用程序能够在设备进入休眠状态的时候，仍然运行或者唤醒设备，以此达到节电的目的。

**1.概况**

**1.1 实现机理**

Android Alarm基于硬件时钟，运行在硬中断中。但他本身没有实现硬件RTC的驱动，它是基于[**linux**](http://lib.csdn.net/base/linux)高精度时钟hrtimer来实现的。

Linux高精度时钟的实现在kernel\_root/kernel/Hrtimer.c文件中。

关于此部分（高精度时钟hrtimer）的分析可以参见如下两篇别人的博文：

<http://blog.csdn.net/walkingman321/article/details/6133171>

<http://blog.csdn.net/walkingman321/article/details/6151172>

**1.2实现方式**

Alarm.c在Linux Kernel中注册了一个平台设备（最终注册了一个RTC设备，保存在变量static struct rtc\_device \*alarm\_rtc\_dev;中），设备节点为/dev/alarm。

用户空间的程序通过fopen打开此节点还获得控制句柄，然后通过ioctrl来向此驱动程序发送请求。

如下文件直接使用了此驱动：

android\_root/system/core/toolbox/alarm.c

android\_root/frameworks/base/serices/jni/com\_android\_server\_AlarmManagerService.cpp

android\_root/system/core/toolbox/Date.c

android\_root/frameworks/base/cmds/runtime/Main\_runtime.cpp

android\_root/frameworks/base/libs/utils/SystemClock.cpp

android\_root/system/core/toolbox/Uptime.c

**1.3使用Alarm驱动的Demo代码**

如果你需要直接操作此驱动，可以参考上面文件中的使用方法。

如下示例摘自SystemClock.cpp:

#if HAVE\_ANDROID\_OS  
    fd = open("/dev/alarm", O\_RDWR);  
    if(fd < 0) {  
        LOGW("Unable to open alarm driver: %s\n", strerror(errno));  
        return -1;  
    }  
    ts.tv\_sec = tv.tv\_sec;  
    ts.tv\_nsec = tv.tv\_usec \* 1000;  
    res = ioctl(fd, ANDROID\_ALARM\_SET\_RTC, &ts);  
    if(res < 0) {  
        LOGW("Unable to set rtc to %ld: %s\n", tv.tv\_sec, strerror(errno));  
        ret = -1;  
    }  
    close(fd);  
#else  
    if (settimeofday(&tv, NULL) != 0) {  
        LOGW("Unable to set clock to %d.%d: %s\n",  
            (int) tv.tv\_sec, (int) tv.tv\_usec, strerror(errno));  
        ret = -1;  
    }  
#endif

**2.驱动代码分析**

**2.1 主要函数清单**

alarm\_start\_hrtimer 工具函数，alarm驱动的其他函数通过此函数为对应类型（android\_alarm\_type）的Alarm建立一个hrtimer定时器。

alarm\_ioctl alarm驱动的ioctl函数，上层代码通过此接口向驱动程序发送操作请求。

alarm\_open

alarm\_release

alarm\_timer\_triggered

alarm\_triggered\_func

alarm\_suspend

alarm\_resume

rtc\_alarm\_add\_device

rtc\_alarm\_remove\_device

alarm\_late\_init 驱动的后置初始化函数，在alarm\_init后运行（参见本节的备注）。

alarm\_init 驱动的初始化函数

alarm\_exit 驱动的退出函数

**备注:** 关于Linux驱动程序的启动顺序（alarm\_init 和alarm\_late\_init）的相关说明，请参见如下博文：

<http://blog.csdn.net/cstk502/article/details/6579231>

**2.2 alarm\_start\_hrtimer 函数**

 代码及注释如下：

static void alarm\_start\_hrtimer(enum android\_alarm\_type alarm\_type)  
{  
 struct timespec hr\_alarm\_time;  
 if (!(alarm\_enabled & (1U << alarm\_type))) //根据掩码alarm\_enabled来判断此类型的enabled.  
  return;  
 hr\_alarm\_time = alarm\_time[alarm\_type]; //保存该类型的timespec到局部变量  
 if (alarm\_type == ANDROID\_ALARM\_ELAPSED\_REALTIME\_WAKEUP ||  
     alarm\_type == ANDROID\_ALARM\_ELAPSED\_REALTIME)  
  set\_normalized\_timespec(&hr\_alarm\_time,  //设置时间格式  
   hr\_alarm\_time.tv\_sec + elapsed\_rtc\_delta.tv\_sec,  
   hr\_alarm\_time.tv\_nsec + elapsed\_rtc\_delta.tv\_nsec);

//记录日志，当前版本已经Disable了这个宏，所以不会有日志出现。  
 ANDROID\_ALARM\_DPRINTF(ANDROID\_ALARM\_PRINT\_FLOW,  
  "alarm start hrtimer %d at %ld.%09ld\n",  
  alarm\_type, hr\_alarm\_time.tv\_sec, hr\_alarm\_time.tv\_nsec);

//为alarm\_timer[alarm\_type]添加一个hrtimer, 如果alarm\_timer[alarm\_type]已经存在一个hrtimer,则先删除然后再添加一个新的。请查看hrtimer.c中hrtimer\_start\_range\_ns函数的实现。  
 hrtimer\_start(&alarm\_timer[alarm\_type],  
        timespec\_to\_ktime(hr\_alarm\_time), HRTIMER\_MODE\_ABS);  
}

总结：此函数最终为alarm\_type类型在数组alarm\_timer[alarm\_type]上创建了一个hrtimer.

2.3 alarm\_init函数

未完待续。

**AlarmManagerService是如何工作的？**

2018年09月19日 16:44:06 [weixin\_42935803](https://me.csdn.net/weixin_42935803) 阅读数：92

系统开发过程中，经常会遇到Android心跳机制，待机耗流问题，几乎都跟AlarmMangerService有关，本文从应用层到kernel记录下整个处理流程。

**应用层**  
样例代码：

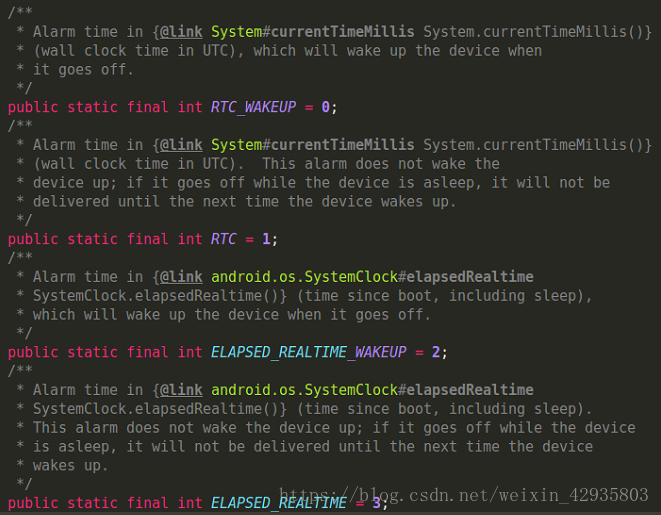
AlarmManager mAlarmManager = (AlarmManager) getContext().getSystemService(Context.ALARM\_SERVICE);

...

mAlarmManager.set(AlarmManager.ELAPSED\_REALTIME\_WAKEUP,

SystemClock.elapsedRealtime() + delayInMs, mDataStallAlarmIntent);

* 1
* 2
* 3
* 4

可以通过获取ALARM\_SERVICE来设置Alarm，  
第一个参数表示Alarm的类型，可选择0,1,2,3；（当系统休眠时，0和2会主动唤醒，1和3不会）  
  
第二个参数表示Alarm到期的时间；  
第三个参数表示Alarm到期时处理的intent事件。

**framework层**  
文件frameworks/base/services/core/java/com/android/server/AlarmManagerService.java  
在Alarm service运行时会启动AlarmThread线程，waitForAlarm一直等待JNI层传递闹钟到期事件。

@Override

public void onStart() {

//初始化jni层，打开alarm节点设备

mNativeData = init();

mNextWakeup = mNextNonWakeup = 0;

// We have to set current TimeZone info to kernel

// because kernel doesn't keep this after reboot

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

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

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

//启动Alarm线程，kernel层rtc唤醒系统立即得到执行

if (mNativeData != 0) {

AlarmThread waitThread = new AlarmThread();

waitThread.start();

} else {

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

}

...

//向SystemService注册ALARM service

publishBinderService(Context.ALARM\_SERVICE, mService);

publishLocalService(LocalService.class, new LocalService());

}

//Alarm线程

private class AlarmThread extends Thread

{

public void run()

{

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

while (true)

{

int result = waitForAlarm(mNativeData);

mLastWakeup = SystemClock.elapsedRealtime();

......

}

}

}

//向jni层设置alarm

private void setLocked(int type, long when) {

if (mNativeData != 0 && mNativeData != -1) {

// The kernel never triggers alarms with negative wakeup times

// so we ensure they are positive.

long alarmSeconds, alarmNanoseconds;

if (when < 0) {

alarmSeconds = 0;

alarmNanoseconds = 0;

} else {

alarmSeconds = when / 1000;

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

}

Slog.d(TAG, "set alarm to RTC " + when + " Type: "+ type);

set(mNativeData, type, alarmSeconds, alarmNanoseconds);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58

**jni层**  
文件frameworks/base/services/core/jni/com\_android\_server\_AlarmManagerService.cpp  
初始化时打开alarm节点设备，  
ioctl如下方法：  
ANDROID\_ALARM\_SET //设置alarm  
ANDROID\_ALARM\_SET\_RTC //设置系统时间  
ANDROID\_ALARM\_WAIT //等待rtc唤醒事件

static const JNINativeMethod sMethods[] = {

/\* name, signature, funcPtr \*/

{"init", "()J", (void\*)android\_server\_AlarmManagerService\_init},

{"close", "(J)V", (void\*)android\_server\_AlarmManagerService\_close},

{"set", "(JIJJ)V", (void\*)android\_server\_AlarmManagerService\_set},

{"waitForAlarm", "(J)I", (void\*)android\_server\_AlarmManagerService\_waitForAlarm},

{"setKernelTime", "(JJ)I", (void\*)android\_server\_AlarmManagerService\_setKernelTime},

{"setKernelTimezone", "(JI)I", (void\*)android\_server\_AlarmManagerService\_setKernelTimezone},

{"bootFromAlarm", "(I)Z", (void\*)android\_server\_AlarmManagerService\_bootFromAlarm},

};

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);

}

int AlarmImplAlarmDriver::set(int type, struct timespec \*ts)

{

return ioctl(fds[0], ANDROID\_ALARM\_SET(type), ts);

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27

**kernel层**  
文件kernel/drivers/staging/android/alarm-dev.c  
注册alarm设备，对alarm节点的ioctrl操作处理。

static const struct file\_operations alarm\_fops = {

.owner = THIS\_MODULE,

.unlocked\_ioctl = alarm\_ioctl,

.open = alarm\_open,

.release = alarm\_release,

#ifdef CONFIG\_COMPAT

.compat\_ioctl = alarm\_compat\_ioctl,

#endif

};

static struct miscdevice alarm\_device = {

.minor = MISC\_DYNAMIC\_MINOR,

.name = "alarm",

.fops = &alarm\_fops,

};

static long alarm\_do\_ioctl(struct file \*file, unsigned int cmd,

struct timespec \*ts, struct rtc\_wkalrm \*alm)

{

enum android\_alarm\_type alarm\_type = ANDROID\_ALARM\_IOCTL\_TO\_TYPE(cmd);

switch (ANDROID\_ALARM\_BASE\_CMD(cmd)) {

case ANDROID\_ALARM\_CLEAR(0):

alarm\_clear(alarm\_type, ts);

break;

case ANDROID\_ALARM\_SET(0):

alarm\_set(alarm\_type, ts);

break;

case ANDROID\_ALARM\_SET\_AND\_WAIT(0):

alarm\_set(alarm\_type, ts);

/\* fall though \*/

case ANDROID\_ALARM\_WAIT:

rv = alarm\_wait();

break;

default:

rv = -EINVAL;

}

return rv;

}

alarm\_set()

devalarm\_start()

alarm\_start()

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43

然后调用rtc设备，各芯片商实现不一样，以MTK为例：  
文件kernel-4.4/drivers/misc/mediatek/rtc/mtk\_rtc\_common.c  
注册rtc设备。

static struct rtc\_class\_ops rtc\_ops = {

.read\_time = rtc\_ops\_read\_time,

.set\_time = rtc\_ops\_set\_time,

.read\_alarm = rtc\_ops\_read\_alarm,

.set\_alarm = rtc\_ops\_set\_alarm,

.ioctl = rtc\_ops\_ioctl,

};

* 1
* 2
* 3
* 4
* 5
* 6
* 7

文件kernel-4.4/drivers/misc/mediatek/rtc/mtk\_rtc\_hal\_common.c  
设置rtc寄存器。

void hal\_rtc\_set\_alarm\_time(struct rtc\_time \*tm)

{

hal\_rtc\_xinfo("mon = %d, day = %d, hour = %d\n",

(rtc\_read(RTC\_AL\_MTH) & ~(RTC\_AL\_MTH\_MASK)) | tm->tm\_mon,

(rtc\_read(RTC\_AL\_DOM) & ~(RTC\_AL\_DOM\_MASK)) | tm->tm\_mday,

(rtc\_read(RTC\_AL\_HOU) & ~(RTC\_AL\_HOU\_MASK)) | tm->tm\_hour);

rtc\_write(RTC\_AL\_YEA,

(rtc\_read(RTC\_AL\_YEA) & ~(RTC\_AL\_YEA\_MASK)) | (tm->tm\_year & RTC\_AL\_YEA\_MASK));

rtc\_write(RTC\_AL\_MTH,

(rtc\_read(RTC\_AL\_MTH) & ~(RTC\_AL\_MTH\_MASK)) | (tm->tm\_mon & RTC\_AL\_MTH\_MASK));

rtc\_write(RTC\_AL\_DOM,

(rtc\_read(RTC\_AL\_DOM) & ~(RTC\_AL\_DOM\_MASK)) | (tm->tm\_mday & RTC\_AL\_DOM\_MASK));

rtc\_write(RTC\_AL\_HOU,

(rtc\_read(RTC\_AL\_HOU) & ~(RTC\_AL\_HOU\_MASK)) | (tm->tm\_hour & RTC\_AL\_HOU\_MASK));

rtc\_write(RTC\_AL\_MIN,

(rtc\_read(RTC\_AL\_MIN) & ~(RTC\_AL\_MIN\_MASK)) | (tm->tm\_min & RTC\_AL\_MIN\_MASK));

rtc\_write(RTC\_AL\_SEC,

(rtc\_read(RTC\_AL\_SEC) & ~(RTC\_AL\_SEC\_MASK)) | (tm->tm\_sec & RTC\_AL\_SEC\_MASK));

rtc\_write(RTC\_AL\_MASK, RTC\_AL\_MASK\_DOW); /\* mask DOW \*/

rtc\_write\_trigger();

}

# Android5.0 alarm实现原理

[](https://www.jianshu.com/u/183339cdc7ae)

[Hly\_Coder](https://www.jianshu.com/u/183339cdc7ae) 关注

2016.07.25 20:02\* 字数 1078 阅读 2263评论 0喜欢 5

转载请标明出处：[http://www.jianshu.com/users/183339cdc7ae/latest\_articles](https://www.jianshu.com/users/183339cdc7ae/latest_articles)

## 概述

这片文章会分析alarm的实现原理（以驱动支持的情况为例），从而可以知道的是：为什么在5.0以上进程被杀死后没有办法在执行alarm的指定动作，  
alarm的系统架构十分简单，

alarm\_arch.png

## AlarmManager

API的使用，大家肯定不会陌生，直接给出代码，以Broadcast为例（省略掉了AndroidManifest.xml里面的权限）

Intent intent = new Intent();

intent.setAction("xxxxx");

PendingIntent pendingIntent = PendingIntent.getBroadcast(this, 0,intent, 0);

AlarmManager am = (AlarmManager) getSystemService(ALARM\_SERVICE);

am.setRepeating(AlarmManager.RTC\_WAKEUP, System.currentTimeMillis(), 5 \* 1000, pendingIntent);

5 \* 1000：单位是毫秒，意思是5s后，以broadcast的方式发送intent  
来看看setRepeating的实现

public void setRepeating(int type, long triggerAtMillis,

long intervalMillis, PendingIntent operation) {

setImpl(type, triggerAtMillis, legacyExactLength(), intervalMillis, operation, null, null);

}

private void setImpl(int type, long triggerAtMillis, long windowMillis, long intervalMillis,

PendingIntent operation, WorkSource workSource, AlarmClockInfo alarmClock) {

...

mService.set(type, triggerAtMillis, windowMillis, intervalMillis, operation,

workSource, alarmClock);

....

}

其实是调用了mService的set方法，mService是IAlarmManager的对象，看名字就可以知道，这里是一个代理类，具体的实现实在AlarmManagerService里面

## AlarmManagerService

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

@Override

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

PendingIntent operation, WorkSource workSource,

AlarmManager.AlarmClockInfo alarmClock) {

...

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

windowLength == AlarmManager.WINDOW\_EXACT, workSource, alarmClock);

}

来看看setImpl的代码

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

PendingIntent operation, boolean isStandalone, WorkSource workSource,

AlarmManager.AlarmClockInfo alarmClock) {

....

if (interval > 0 && interval < MIN\_INTERVAL) {

interval = MIN\_INTERVAL;

}

....

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

interval, operation, isStandalone, true, workSource, alarmClock, userId,

private static final long MIN\_INTERVAL = 60 \* 1000;

在**5.0**的android上面，如果app里面设置的触发时间<1min，系统会强制将触发时间设置成1min。  
setImplLocked这个方法会更新AlarmManagerService中的list，该list保存的是整个系统中所有的alarm。  
然后调用rescheduleKernelAlarmsLocked-->setLocked

private void setLocked(int type, long when) {

if (mNativeData != 0) {...}

...

set(mNativeData, type, alarmSeconds, alarmNanoseconds);

} else {

Message msg = Message.obtain();

msg.what = ALARM\_EVENT;

mHandler.removeMessages(ALARM\_EVENT);

mHandler.sendMessageAtTime(msg, when);

}

}

这个mNativeData成员变量如果！=0，表示的是该系统支持驱动设置时间（笔者这里猜想的是可以通过设置时间到驱动中，通过硬件中断来完成时间到点后的回调）。如果=0，则使用handler的方式来执行，handler的具体实现其实是使用的Linux中的一个叫epoll的api来实现的，这里不做具体讲解。 我们先假设mNativeData ！=0的情况，会调用一个native的set方法

###### com\_android\_server\_AlarmManagerService.cpp

static JNINativeMethod sMethods[] = {

....

{"set", "(JIJJ)V", (void\*)android\_server\_AlarmManagerService\_set},

....

};

static void android\_server\_AlarmManagerService\_set(JNIEnv\*, jobject, jlong nativeData, jint type, jlong seconds, jlong nanoseconds)

{

AlarmImpl \*impl = reinterpret\_cast<AlarmImpl \*>(nativeData);

...

int result = impl->set(type, &ts);

...

}

AlarmImpl的set方法是一个虚方法，有2个类实现该方法

1. AlarmImplAlarmDriver
2. AlarmImplTimerFd

到底是使用的哪个对象来实现，先来看看初始化的时候使用的哪一个

static jlong android\_server\_AlarmManagerService\_init(JNIEnv\*, jobject) {

jlong ret = init\_alarm\_driver();

if (ret) {

return ret;

}

return init\_timerfd();

}

init\_alarm\_driver

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);

}

可以知道如果驱动支持，会在dev目录下面生产一个alarm的节点，在初始化alarm系统的时候fd就不会<0  
本文最开始说了，这里以驱动支持为例.

当我们系统不支持驱动设置时间的时候，会调用init\_timerfd方法，大家可以自行去看看该方法，使用的就是epoll的机制，前文也提到过

所以来看看AlarmImplAlarmDriver的set方法

int AlarmImplAlarmDriver::set(int type, struct timespec \*ts)

{

return ioctl(fds[0], ANDROID\_ALARM\_SET(type), ts);

}

通过ioctl的方式将时间设置给驱动，后面的实现需要去看驱动代码，因为各个厂商的实现可能会不同，这里不做详细讲解  
但是最后的作用应该都是**一样**的，就是把时间设置给驱动，等到硬件中断返回后，再回调到native层，native再回调到framework

## native回调

那硬件中断产生后是如何回调的呢？来看看AlarmManagerService里做的一个重要事情（假设驱动支持，即mNativeData !=0）

如果不支持的话会以epoll的方式回调

###### AlarmManagerService

@Override

public void onStart() {

...

if (mNativeData != 0) {

AlarmThread waitThread = new AlarmThread();

waitThread.start();

} else {

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

}

...

}

对，你没有看错，这里启动一个thread

private class AlarmThread extends Thread {

...

public void run() {

while (true) {

int result = waitForAlarm(mNativeData);

...

deliverAlarmsLocked(triggerList, nowELAPSED);

}

}

}

一个死循环来等待驱动的返回，waitForAlarm是一个native方法具体的实现在驱动中。如果整个系统中没有alarm的时间回调，waitForAlarm则阻塞在这，直到有回调的时候才往后执行，这样会减少CPU的开销。

## framework回调

###### deliverAlarmsLocked

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

...

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

mBackgroundIntent.putExtra(

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

mResultReceiver, mHandler);

...

这里的operation是PendingIntent对象，最后回调其实是PendingIntent的send方法

public void send(Context context, int code, Intent intent,

OnFinished onFinished, Handler handler, String requiredPermission)

throws CanceledException {

try {

String resolvedType = intent != null ?

intent.resolveTypeIfNeeded(context.getContentResolver())

: null;

int res = mTarget.send(code, intent, resolvedType,

onFinished != null

? new FinishedDispatcher(this, onFinished, handler)

: null,

requiredPermission);

...

}

其中最重要的是**mTarget**，来看看mTarget是何方圣神

private final IIntentSender mTarget;

/\*package\*/ PendingIntent(IIntentSender target) {

mTarget = target;

}

那又是合适调用的PendingIntent(IIntentSender target)方法呢，这里以getBroadcast为例

public static PendingIntent getBroadcast(Context context, int requestCode,

Intent intent, @Flags int flags) {

return getBroadcastAsUser(context, requestCode, intent, flags,

new UserHandle(UserHandle.myUserId()));

}

public static PendingIntent getBroadcastAsUser(Context context, int requestCode,

Intent intent, int flags, UserHandle userHandle) {

...

try {

...

IIntentSender target =

ActivityManagerNative.getDefault().getIntentSender(

ActivityManager.INTENT\_SENDER\_BROADCAST, packageName,

null, null, requestCode, new Intent[] { intent },

resolvedType != null ? new String[] { resolvedType } : null,

flags, null, userHandle.getIdentifier());

return target != null ? new PendingIntent(target) : null;

} catch (RemoteException e) {

}

return null;

}

当我们再app中使用PendingIntent的getBroadcast方法的时候，返回的PendingIntent对象中**包含了一个私有变量target，该taget就是驱动回调到framework层后正真做事情的发起点。**  
如果当我们app所在的进程被杀死后，进程中的变量会被回收，当然这个target也会被回收掉，所以**没有办法进行回调**  
那taget的send方法是怎么实现的呢？

###### PendingIntentRecord.java

final class PendingIntentRecord extends IIntentSender.Stub {

...

public int send(int code, Intent intent, String resolvedType,

IIntentReceiver finishedReceiver, String requiredPermission) {

return sendInner(code, intent, resolvedType, finishedReceiver,

requiredPermission, null, null, 0, 0, 0, null, null);

}

...

int sendInner(int code, Intent intent, String resolvedType,

IIntentReceiver finishedReceiver, String requiredPermission,

IBinder resultTo, String resultWho, int requestCode,

int flagsMask, int flagsValues, Bundle options, IActivityContainer container) {

...

switch (key.type) {

case ActivityManager.INTENT\_SENDER\_ACTIVITY:

...

owner.startActivitiesInPackage(uid, key.packageName, allIntents,

allResolvedTypes, resultTo, options, userId);

case ActivityManager.INTENT\_SENDER\_BROADCAST:

...

owner.broadcastIntentInPackage(key.packageName, uid,

finalIntent, resolvedType,

finishedReceiver, code, null, null,

requiredPermission, (finishedReceiver != null), false, userId);

...

case ActivityManager.INTENT\_SENDER\_SERVICE:

...

owner.startServiceInPackage(uid,

finalIntent, resolvedType, userId);

...

}

...

}

}

最后根据不同的类型调用ams的方法启动Broadcast / Activity / Service

## 总结

老规矩，上一张流程图

**小礼物走一走，来简书关注我**