[android – 如何读取“adb shell dumpsys alarm”输出](https://codeday.me/bug/20171005/79983.html)

时间：2017-10-05 标签：[adb](https://codeday.me/tag/adb)[alarmmanager](https://codeday.me/tag/alarmmanager)[android](https://codeday.me/tag/android)

我正在努力正确设置闹钟，并了解取消和重新安排闹钟的机制。

我发现，有一个adb命令来检索设备上安排的所有警报，但我还没有找到一个文档，解释输出的格式。

我明白，我在这里要求很多解释，所以如果有人会抛出一个链接与详细的解释关于“adb shell dumpsys报警”，我会真的很感激。

所以，这里有问题：

>

Pending alarm batches: 23

一个。 ’23’是当前活动的多个预定报警吗？  
>

Batch{4293d3a8 num=1 start=1369361 end=1407261}:  
  
  RTC #0: Alarm{4293d358 type 1 com.android.chrome}  
  
    type=1 whenElapsed=1369361 when=+19s304ms window=-1 repeatInterval=0 count=0  
  
    operation=PendingIntent{429e4500: PendingIntentRecord{429dbbc8 com.android.chrome broadcastIntent}}

一个。什么是“num = 1”，“start = 1369361”和“end = 1407261”？  
b。 ‘RTC’代表RTC报警，我假设。  
C。什么’#0’代表？  
d。什么意思’type = 1’？  
e。是’when = 19s304ms’意味着警报将在19秒内触发？  
F。什么意思是’window = -1’？  
G。是’repeatInterval = 0’这意味着这是不重复的报警？  
H。是’count = 0’意味着这个警报没有推迟，由于手机睡眠状态？  
一世。 ‘operation = PendingIntent {…}’表示待处理的意图，将由警报触发，我假设。  
>

Broadcast ref count: 0

一个。这是什么？  
>

Top Alarms:

一个。这是什么？  
>

+47s271ms running, 0 wakeups, 2 alarms: com.username.weatherinfo  
  
  act=com.username.receivers.CyclicWeatherUpdater.WEATHER\_UPDATE\_ACTION  
  
    cmp={com.username.weatherinfo/com.username.receivers.CyclicWeatherUpdater}

一个。是’47s271ms’意思是这个警报将在47秒内触发？  
b。什么是’0 wakeups’ – 警报从未被触发？  
C。什么是“2报警”？  
d。是“com.username.weatherinfo”代表程序包的名称，在上下文字段中被赋予pending intent吗？  
e。 “行为”是指为了意图而发出的行动？  
F。什么是“cmp”？我看到，它是从包名称和类名称组成 – 但从他们被带走？从意图构造函数？  
G。为什么部分报警只有“行为”或只有“cmp”？我假设，没有’cmp’字段的报警是隐式广播意图。然而，为什么没有“行动”字段的报警？  
>

Alarm Stats:

一个。这是什么？

我意识到这个线程是老的，但答案不容易找到，并可以使用。我花了很多时间来研究这些消息的含义。

Q1：批量

Pending alarm batches: 23

警报按批次组织。 [As described in the documentation](http://developer.android.com/reference/android/app/AlarmManager.html#set(int,%20long,%20android.app.PendingIntent))：

Beginning in API 19, the trigger time passed to this method is treated as inexact: the alarm will not be delivered before this time, but may be deferred and delivered some time later. The OS will use this policy in order to **“batch”** alarms together across the entire system, minimizing the number of times the device needs to “wake up” and minimizing battery use. In general, alarms scheduled in the near future will not be deferred as long as alarms scheduled far in the future.

每个批次可能有多个警报。在这种情况下，有23批次的报警，这意味着可能有超过23个报警计划。在dumpsys报警输出中，描述每个批次的行如下所示：

Batch{4293d3a8 num=1 start=1369361 end=1407261}:

其中：

> 4293d3a8是与批处理相关联的内部标识。  
> num = 1是此批处理中的报警数。在这种情况下，批处理中只有一个警报。  
>开始和结束数字表示自从系统最后重新启动以来已过去的毫秒数为[described in this post](https://stackoverflow.com/a/15400014/296725)，并且还粗略地表示应当触发批量中的警报的时间窗口。

Q2：报警

每个报警由三条线描述，如下所示：

RTC #0: Alarm{4293d358 type 1 com.android.chrome}

type=1 whenElapsed=1369361 when=+19s304ms window=-1 repeatInterval=0 count=0

operation=PendingIntent{429e4500: PendingIntentRecord{429dbbc8 com.android.chrome broadcastIntent}}

其中：

>第一部分，即RTC\_WAKEUP，RTC，ELAPSED\_WAKEUP或ELAPSED中的一个，表示报警的类型，分别对应于整数值0-3  
>#0是批次中的报警编号，其中数字从0到n-1，其中n是批次中的报警数。如果您的报警与其他人分开，未来最远的“when =”定义将触发批次中的所有报警的时间。  
> 4293d358是与警报关联的内部标识号  
> com.android.chrome是设置警报的类的包名称  
> type = 1，报警类型，见上面第一个项目符号  
> whenElapsed = 1369361指自系统启动以来将触发此警报的毫秒数(大约)  
> when = 19s304ms表示报警将在调用dumpsys报警时的19秒，即304毫秒内触发。同样，像2d13h29m03s882ms这样的值指的是未来的相对时间2天，13小时，29分钟…  
> window =指与报警批处理的方法有关的两个内部常量之一。 AlarmManager.WINDOW\_EXACT = 0，并在使用setExact()或setAlarmClock()计划报警时设置。 AlarmManager.WINDOW\_HEURISTIC = -1，在使用setInexactRepeating()计划报警时设置。否则，该值由API版本确定。对于API< 19(KitKat)，使用WINDOW\_EXACT，对于API> = 19，使用WINDOW\_HEURISTIC。 (我不得不[dig into the AlarmManager.java source code](https://github.com/android/platform_frameworks_base/blob/master/core/java/android/app/AlarmManager.java)来解决这个问题。)  
> repeatInterval = 900000是警报重复的频率，例如。每900000ms或15分钟。值为0表示报警不重复。  
> count =指警报应该被触发，但不是由于某种原因的次数。 0是一个很好的数字。 > 0表示由于某种原因跳过了闹钟。  
> operation = PendingIntent {…}是对由警报触发的[PendingIntent](http://developer.android.com/reference/android/app/PendingIntent.html)的引用。根据是否使用getService，getBroadcast，getActivity或getActivities实例化PendingIntent，警报将启动服务，发送广播或启动一个或多个活动。

Q3：广播参考计数

要找出这个和其他输出项目后，我不得不[dig into the AlarmManagerService.java source code](https://github.com/android/platform_frameworks_base/blob/master/services/core/java/com/android/server/AlarmManagerService.java)。

为了使一些警报工作，设备必须被唤醒，并且在所有必要的广播被发送之前不应该回到睡眠。内部变量mBroadcastRefCount初始化为0，并且随着要发送的广播排队而递增。当每个广播被发送时，它被递减，并且当它返回到0时，wakeLock被释放，并且设备可以回到睡眠状态。

广播参考计数：0仅仅意味着在dumpsys警报运行时，它不是在发送任何广播的中间。

Q4：顶部警报

这是按报警代码已运行的总聚合时间降序排列的前十个报警。这可以用于找到消耗最大量的系统资源的警报，例如。找到可能出现故障以排除电池寿命的过程。

Q5：报警统计

此部分显示自上次重新启动系统以来运行的所有警报的统计信息。在这里，您可以查看您在过去设置的警报是否已触发，如果他们唤醒了手机等。这些条目的格式是覆盖下面。

Q6：报警统计输入

报警统计信息条目如下所示：

com.example.someapp +1s857ms running, 0 wakeups:

+1s817ms 0 wakes 83 alarms: cmp={com.example.someapp/com.example.someapp.someservice}

+40ms 0 wakes 1 alarms: cmp={com.example.someapp/com.example.someapp.someotherservice}

其中在第一行：

> com.example.someapp是触发警报的进程的包名称  
> 1s857ms running是进程消耗的总系统时间  
> 0 wakeups是设备被这些警报之一唤醒的次数

然后每行之后引用设置的警报之一，其中：

> 1s817ms是系统消耗的总时间  
> 0 wakes是设备必须被唤醒的次数  
> 83报警是报警触发的次数;对于重复报警，这将仅为> 1  
> cmp = {…}触发警报时启动的服务

或者，如果警报触发广播，则条目可能如下：

android +4m51s566ms running, 281 wakeups:

+2m46s583ms 0 wakes 1224 alarms: act=android.intent.action.TIME\_TICK

+1m25s624ms 89 wakes 89 alarms: act=android.content.syncmanager.SYNC\_ALARM

+52s898ms 0 wakes 41 alarms: act=com.android.server.action.NETWORK\_STATS\_POLL

...

具有：

> act = …是被广播的意图的名称

警报可能具有cmp = {…}和act = …条目，这意味着警报既广播意图并启动服务。

概要

调试android报警使用adb shell dumpsys报警的输出可能是棘手的，没有中心位置，dumpsys消息完全解释。并不总是明显的警报如何批处理在一起，有时很难得到一个服务或活动被触发完全在需要的时候。希望这将是人们试图调试他们的报警的有用的参考。

翻译自：https://stackoverflow.com/questions/28742884/how-to-read-adb-shell-dumpsys-alarm-output

# Android Alarm、WakeLock机制与对齐唤醒

发表于 2017-03-27   |

### Android睡眠唤醒机制

Android在标准linux内核原有睡眠唤醒模块基础上增加了WakeLock唤醒锁机制，WakeLock是Android提供给应用程序获取电力资源的唯一方法。只要还有地方在使用WakeLock，系统就不会进入休眠状态。

WakeLock的一般使用方法如下：

|  |  |
| --- | --- |
| 1 2 3 4 5 | PowerManager pm = (PowerManager) getSystemService(Context.POWER\_SERVICE); PowerManager.WakeLock wl = pm.newWakeLock(PowerManager.SCREEN\_DIM\_WAKE\_LOCK, "My Tag"); wl.acquire();  ..screen will stay on during this section.. wl.release(); |

WakeLock构造函数中有个flags参数，该参数决定了不同的wake lock等级，如下所示

### 为什么要做对齐唤醒

目前，在移动互联网的大浪潮下，基本上的APP都集成有消息推送功能，而不仅仅局限于即时通讯类应用。

推送技术的解决方案主要有以下几类：

1、**轮询（pull方式）**: 应用程序阶段性的与服务器进行连接并查询是否有新的消息到达  
由于客户端不知道服务器是否有新消息，只能通过定时向服务器询问，如果频率设定过高将消耗大量的宽带和电量，过低又导致消息延迟严重，该方案不适用于对消息有及时性需求的应用，一般不采用该方案。  
2、**SMS通知（push方式）**: 通过拦截SMS消息并且解析消息内容来了解服务器的意图  
该方案可保证一定的实时性需求，但是需要向运营商缴纳相应的费用，所以该方案的使用也较少。  
3、**长连接（push方式）**: 应用程序和服务器保持一个长连接，服务器的消息可以直接通过这个链接push到应用程序。  
能保证消息的实时性，并且不需要向运营商缴纳高额费用，所以基本上推送方案都是使用长连接来实现。

### 长连接、NAT超时和心跳

**长连接**：客户端与服务器建立连接之后, 不主动断开. 双方互相发送数据, 发完了也不主动断开连接, 之后有需要发送的数据就继续通过这个连接发送.

**NAT超时**：因为IPv4的IP量有限，运营商分配给手机终端的IP是运营商内网的IP，手机要连接Internet，就需要通过运营商的网关做一个网络地址转换(Network Address Translation，NAT)。  
简单的说运营商的网关需要维护一个外网IP、端口到内网IP、端口的对应关系，以确保内网的手机可以跟 Internet 的服务器通讯。  
大部分移动无线网络运营商都在链路一段时间没有数据通讯时，会淘汰 NAT 表中的对应项，造成链路中断。

下表列出一些已测试过的网络的NAT超时时间（内容来源与网络）：

地区/网络 NAT超时时间  
中国移动 5分钟  
中国联通 5分钟  
中国电信 大于28分钟  
美国 大于28分钟  
台湾 大于28分钟

由于NAT超时的存在，推送为了保障消息及时性，必须维持长连接不断开，也就有了心跳机制。所谓心跳也就是客户端需要定时向服务器发送数据包，防止NAT超时。

android平台多样性，产生了诸如小米、个推、友盟、阿里、百度云、极光、信鸽和华为等等众多推送平台。随着我们手机安装的应用的增多，会有各种各样的推送服务在后台运行，它们会定时唤醒机器，向自己服务器发送心跳包，但各个推送发送心跳包的时间不一致，导致机器在短时间内多次重复被唤醒，大大降低手机的续航能力。

因此，对齐唤醒要解决的问题，就是将我们所有我们认为定时唤醒异常的应用，将它们的定时唤醒任务对齐，在一次唤醒中全部触发，以降低机器重复唤醒的频率。如下图，miui的官方宣传ppt中所示。

## 对齐唤醒实现方案

由于市面上的各种管家类应用都存在后台清理之类功能，手机厂商自己定制rom为了续航也会清理后台进程，推送保活就成了推送的首要任务。推送需要定时向服务器发送数据包，必然需要去设定一个定时任务，由于后台清理的存在，显然应用自己实现计时器不是靠谱的方案。因此，需借助于android系统的AlarmManagerService，来定时向服务器发送心跳包。

所以，我们的首要任务就是将alarm对齐。

### 定时唤醒——AlarmManager

AlarmManager类是为应用提供接口，以访问系统的AlarmManagerService服务的类，以便应用可以设定自己在未来某个时间点运行。当alarm到时间响应时，系统会广播之前应用所注册的intent，如果目标应用未启动，系统会将其自动启动。已注册的alarms在系统休眠时也会得到保存，但会在设备重启或者关闭时被清除。AlarmManager将会在在reveiver的onReceive()方法执行的时候一直持有wakelock锁,以保证手机在执行广播时不进入休眠状态。一旦onReceive()执行返回，AlarmManager将会释放该wakelock。

从Android 4.4（API Level19）开始，所有的重复闹钟都将是非精确型的，我们将借助于android原生这套方案来实现对齐唤醒的功能。

Alarm类型有两大类：ELAPSED\_REALTIME 和 REAL\_TIME\_CLOCK（RTC）。  
ELAPSED\_REALTIME从系统启动之后开始计算，REAL\_TIME\_CLOCK使用的是世界统一时间（UTC）。也就是说由于ELAPSED\_REALTIME不受地区和时区的影响，所以它适合于基于时间差的闹钟（例如一个每过30秒触发一次的闹钟）。REAL\_TIME\_CLOCK适合于那些依赖于地区位置的闹钟。

两种类型的闹钟都还有一个唤醒（WAKEUP）版本，也就是可以在设备屏幕关闭，在系统进入休眠的时候唤醒CPU。这可以确保闹钟会在既定的时间被激活，对于那些实时性要求比较高的应用（比如含有一些对执行时间有要求的操作）来说非常有效。如果你没有使用唤醒版本的闹钟，那么所有的重复闹钟会在下一次设备被唤醒时被激活。

下面列出闹钟的具体类型：

ELAPSED\_REALTIME：从设备启动之后开始算起，度过了某一段特定时间后，激活Pending Intent，但不会唤醒设备。其中设备睡眠的时间也会包含在内。  
ELAPSED\_REALTIME\_WAKEUP：从设备启动之后开始算起，度过了某一段特定时间后唤醒设备。  
RTC：在某一个特定时刻激活Pending Intent，但不会唤醒设备。  
RTC\_WAKEUP：在某一个特定时刻唤醒设备并激活Pending Intent。

对齐唤醒，只需要处理WAKEUP类型的闹钟。  
下面是Android官方培训教程中的一段通过AlarmManager来设定定时任务的典型代码：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 | private AlarmManager alarmMgr; private PendingIntent alarmIntent; ... alarmMgr = (AlarmManager)context.getSystemService(Context.ALARM\_SERVICE); Intent intent = new Intent(context, AlarmReceiver.class); alarmIntent = PendingIntent.getBroadcast(context, 0, intent, 0);  alarmMgr.set(AlarmManager.ELAPSED\_REALTIME\_WAKEUP,  SystemClock.elapsedRealtime() +  60 \* 1000, alarmIntent); |

除了set(int type, long triggerAtMillis, PendingIntent operation)接口之外，AlarmManager还提供了以下几个常用接口供应用调用：

setRepeating(int type, long triggerAtMillis,long intervalMillis, PendingIntent operation)：设定一个重复的闹钟  
setInexactRepeating(int type, long triggerAtMillis,long intervalMillis, PendingIntent operation)：设定一个非精准重复的闹钟  
setExact(int type, long triggerAtMillis, PendingIntent operation)：设定一个精准时间响应闹钟  
setWindow(int type, long windowStartMillis, long windowLengthMillis,PendingIntent operation)：设定一个指在指定windowLength内响应的闹钟（对于参数windowLengthMillis我们后续分析源码的时候再作解释）

**注意**：从Android 4.4（API Level19）开始，alarm默认设置为非精准定时，以便减少OS唤醒次数，降低电量损耗，如要实现精准定时需使用新接口setWindow(int, long, long, PendingIntent)或 setExact(int, long, PendingIntent)。

阅读源码，可以很清楚看到，不管通过哪种接口设定alarm，最终调用到AlarmManagerService.java当中的setImpl(type, triggerAtTime, windowLength, interval, operation, flags, workSource, alarmClock, callingUid)方法，如下。（具体调用流程不在本文分析）

|  |  |
| --- | --- |
| 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 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 | ..\frameworks\base\services\core\java\com\android\server\AlarmManagerService.java  void setImpl(int type, long triggerAtTime, long windowLength, long interval,  PendingIntent operation, int flags, WorkSource workSource,  AlarmManager.AlarmClockInfo alarmClock, int callingUid) {   //判断PendingIntent是否为空，一个alarm必须含有对应的PendingIntent  if (operation == null) {  Slog.w(TAG, "set/setRepeating ignored because there is no intent");  return;  }   //检测windowLength参数，纠正粗心的开发者所设定的错误值，如果大于12小时，将被强制纠正设定为1个小时  // Sanity check the window length. This will catch people mistakenly  // trying to pass an end-of-window timestamp rather than a duration.  if (windowLength > AlarmManager.INTERVAL\_HALF\_DAY) {  Slog.w(TAG, "Window length " + windowLength  + "ms suspiciously long; limiting to 1 hour");  windowLength = AlarmManager.INTERVAL\_HOUR;  }   //检查重复闹钟间隔时间，最小为间隔时间为1分钟  // Sanity check the recurrence interval. This will catch people who supply  // seconds when the API expects milliseconds.  final long minInterval = mConstants.MIN\_INTERVAL;  if (interval > 0 && interval < minInterval) {  Slog.w(TAG, "Suspiciously short interval " + interval  + " millis; expanding to " + (minInterval/1000)  + " seconds");  interval = minInterval;  }   //进行alarm type的合法性检查  if (type < RTC\_WAKEUP || type > ELAPSED\_REALTIME) {  throw new IllegalArgumentException("Invalid alarm type " + type);  }   //检查alarm触发时间  if (triggerAtTime < 0) {  final long what = Binder.getCallingPid();  Slog.w(TAG, "Invalid alarm trigger time! " + triggerAtTime + " from uid=" + callingUid  + " pid=" + what);  triggerAtTime = 0;  }   //开机距离现在时长  final long nowElapsed = SystemClock.elapsedRealtime();   //根据alarm type类型进行触发时间转换  final long nominalTrigger = convertToElapsed(triggerAtTime, type);   // 为了防止alarm立刻触发，添加最小触发时间（5s）  // Try to prevent spamming by making sure we aren't firing alarms in the immediate future  final long minTrigger = nowElapsed + mConstants.MIN\_FUTURITY;   // 通过名义上的触发时间与最小触发时间判断，获取真正的触发时间  final long triggerElapsed = (nominalTrigger > minTrigger) ? nominalTrigger : minTrigger;   //计算最长触发时间  final long maxElapsed;  if (windowLength == AlarmManager.WINDOW\_EXACT) {  //精准闹钟立刻触发，最长触发时间等于意图触发时间  maxElapsed = triggerElapsed;  } else if (windowLength < 0) {  //根据算法得出最大触发时间  maxElapsed = maxTriggerTime(nowElapsed, triggerElapsed, interval);  // Fix this window in place, so that as time approaches we don't collapse it.  windowLength = maxElapsed - triggerElapsed;  } else {  //对于设置windowLength>0的alarm，最长触发时间=意图触发时间+windowLength  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));  }  setImplLocked(type, triggerAtTime, triggerElapsed, windowLength, maxElapsed,  interval, operation, flags, true, workSource, alarmClock, callingUid);  } } |

上述方法当中有个重要的参数windowLength（时间窗口长度），这个参数将会决定alarm的真正触发时间。  
我们之前有提到，从android 4.4之后开始，所有的alarm默认都是非精准的，除非显式调用设定为精准模式。  
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。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。

我们要做的工作就是基于xposed框架，hook该函数，将所有我们认为是定时唤醒异常的应用所设定的唤醒类alarm，添加一个windowLength时长，将它们加入到合适的batch当中，来实现在一次唤醒当中处理多次唤醒任务。

### 非定时唤醒

我们上面提到的定时唤醒，AlarmManager在执行PendingIntent中的onReceive方法时会持有一个wakelock锁。其实应用也能直接申请WakeLock唤醒锁，对于这种直接申请WakeLock锁的情形，我们统一将其划分为非定时唤醒类。

对于非定时唤醒处理，由于它不像定时器是在未来某个时间点才触发，而是即刻持有即刻唤醒。由于我们无法判定当一个应用直接申请wakelock锁的，下一个wakelock唤醒锁会在什么时间申请，很难像AlarmManager去设定一个合适的时间维护窗口来管理。并且，我们不清楚应用唤醒锁持有阶段意图执行的具体任务,也就是说无法获知应用在执行acquire()和release()方法之间执行什么样的代码操作，所以无法将其意图执行任务进行缓存，等待其他wakelock申请时一起执行。

对于这种情形，个人认为应该结合实际场景来考虑：

场景一：对于用户非主动触发唤醒（比如按power键）达到一定时长（1小时）的情形  
1.只允许应用申请PARTIAL\_WAKE\_LOCK类型唤醒锁，对于其他申请的唤醒锁类型，强制替换成PARTIAL\_WAKE\_LOCK，也就是说只允许cpu唤醒，不允许屏幕和键盘亮起。

2.同时，设定允许最小唤醒屏幕和键盘亮起唤醒时间，当非PARTIAL\_WAKE\_LOCK唤醒锁申请时，判断上一次屏幕亮起的时间距离现在的时长是否超过所允许的最新唤醒时间，如果是则允许亮屏唤醒。

场景二：新接收到即时通讯类APP通知消息

当通知栏新增第一条该应用通知消息时，允许屏幕唤醒，从此刻开始到在下一次进入黑屏时间内，如果用户没有进入该应用，或未点击该消息通知，或未主动清除消息通知。则在下一次通知消息来临时，判断距离上次亮屏时间是否大于所设定的最小时长（比如15分钟），如果是则允许亮屏，否则不允许亮屏。

增强的Doze模式

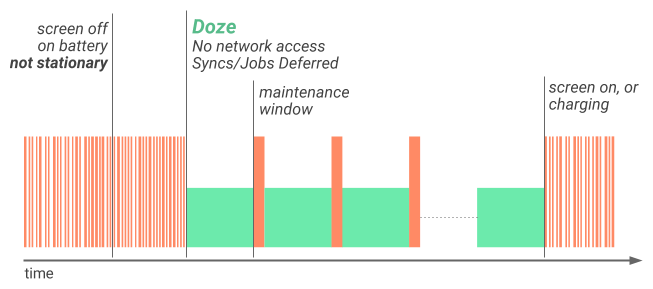
后台优化

Data Saver

一.**增强的Doze模式**

Android N对Android M引进的Doze模式进行了进一步的增强,变化体现在两个方面.一方面是降低了进入Doze模式的条件,Android M中的条件是不插电,屏幕熄灭且静置一段时间,在Android N中去掉了静置的条件,这个改变大大增加了设备进入Doze模式的机会,因而使得Doze对应用程序的影响大大增加.另一方面,Doze模式被分成了两个阶段,当设备切断电源,熄灭屏幕一段时间,会进入到第一阶段,切断网络连接,推迟任务和同步.如果设备在第一阶段的基础上再静置一段时间,就会进入第二阶段,在第一阶段的基础上增加对维持唤醒(PowerManager.WakeLock),定时任务(AlarmManager alarms),GPS和Wi-Fi扫描的限制,如下图所示:

First level



Second level



**应对方案:**

**方案一**:在Android 6.0中AlarmManager中增加了两个方法setAndAllowWhileIdle() and setExactAndAllowWhileIdle(),通过使用这两个方法可以让alarm在Doze模式下运行.

需要注意的是官方文档指出,使用这两个方法时,每个应用每9分钟只能唤醒一次alarm.

**Note:** Neither [setAndAllowWhileIdle()](https://www.cnblogs.com/happyhacking/p/5397391.html#setAndAllowWhileIdle(int,%20long,%20android.app.PendingIntent)) nor [setExactAndAllowWhileIdle()](https://www.cnblogs.com/happyhacking/p/5397391.html#setExactAndAllowWhileIdle(int,%20long,%20android.app.PendingIntent)) can fire alarms more than once per 9 minutes, per app.

以一个定时任务为例进行测试,核心代码如下:

Service:

在Service的onStartCommand()方法中,获取一个AlarmManager的实例,设置任务执行的时间是10s后,处理定时任务的广播接收器是AlarmReceiver.

public class LongRunningService extends Service {  
  
    public static final String *TAG*= "LongRunningService";  
    @Nullable  
    @Override  
    public IBinder onBind(Intent intent) {  
        return null;  
    }  
  
    @Override  
    public int onStartCommand(Intent intent, int flags, int startId) {  
        new Thread(new Runnable() {  
            @Override  
            public void run() {  
                Log.*i*(*TAG*,"executed at " + new Date().toString());  
            }  
        }).start();  
  
        AlarmManager manager = (AlarmManager) getSystemService(*ALARM\_SERVICE*);  
        int offset= 10 \* 1000;//间隔时间10s  
        long triggerAtTime = SystemClock.*elapsedRealtime*() + offset;  
        Intent i = new Intent(this, AlarmReceiver.class);  
        PendingIntent pendingIntent = PendingIntent.*getBroadcast*(this, 0, i, 0);  
  
        manager.setAndAllowWhileIdle(AlarmManager.*ELAPSED\_REALTIME\_WAKEUP*, triggerAtTime, pendingIntent);  
//        manager.setExactAndAllowWhileIdle(AlarmManager.ELAPSED\_REALTIME\_WAKEUP, triggerAtTime, pendingIntent);  
  
        return super.onStartCommand(intent, flags, startId);  
    }  
}

BroadcastReceiver:

重写onReceive()方法,创建一个Intent,启动LongRunningService.这样一来,就形成了一个每隔10s执行一次的定时任务.

public class AlarmReceiver extends BroadcastReceiver {  
    @Override  
    public void onReceive(Context context, Intent intent) {  
        Intent i = new Intent(context, LongRunningService.class);  
        context.startService(i);  
    }  
}

测试结果:

 I/LongRunningService: executed at Thu Apr 14 22:32:58 GMT+08:00 2016

 I/LongRunningService: executed at Thu Apr 14 22:33:08 GMT+08:00 2016

 I/LongRunningService: executed at Thu Apr 14 22:33:18 GMT+08:00 2016

 I/LongRunningService: executed at Thu Apr 14 22:42:18 GMT+08:00 2016

 I/LongRunningService: executed at Thu Apr 14 22:51:18 GMT+08:00 2016

从测试结果可以看出,设备在正常使用的情况下(前三行),每隔10s运行一次,进入到Doze模式后(后三行),每隔9分钟执行一次.

为方便操作,这里介绍一下测试步骤:

Step1.运行应用程序

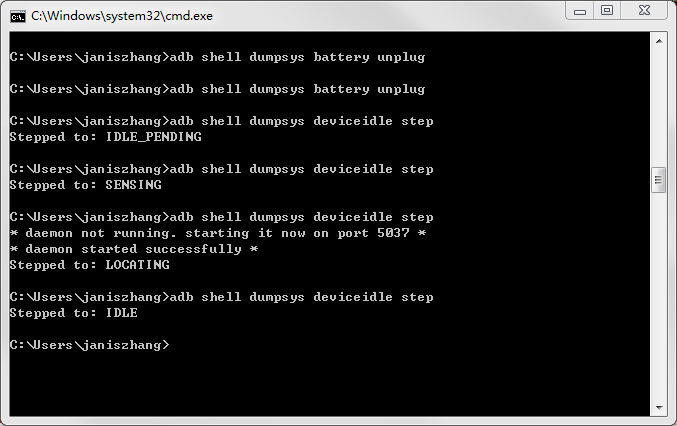
Step2.关闭设备的屏幕

Step3.使用如下命令强制系统进入Doze模式

$ adb shell dumpsys battery unplug

$ adb shell dumpsys deviceidle step

需要多次运行第二条命令,直到设备进入到空闲状态.



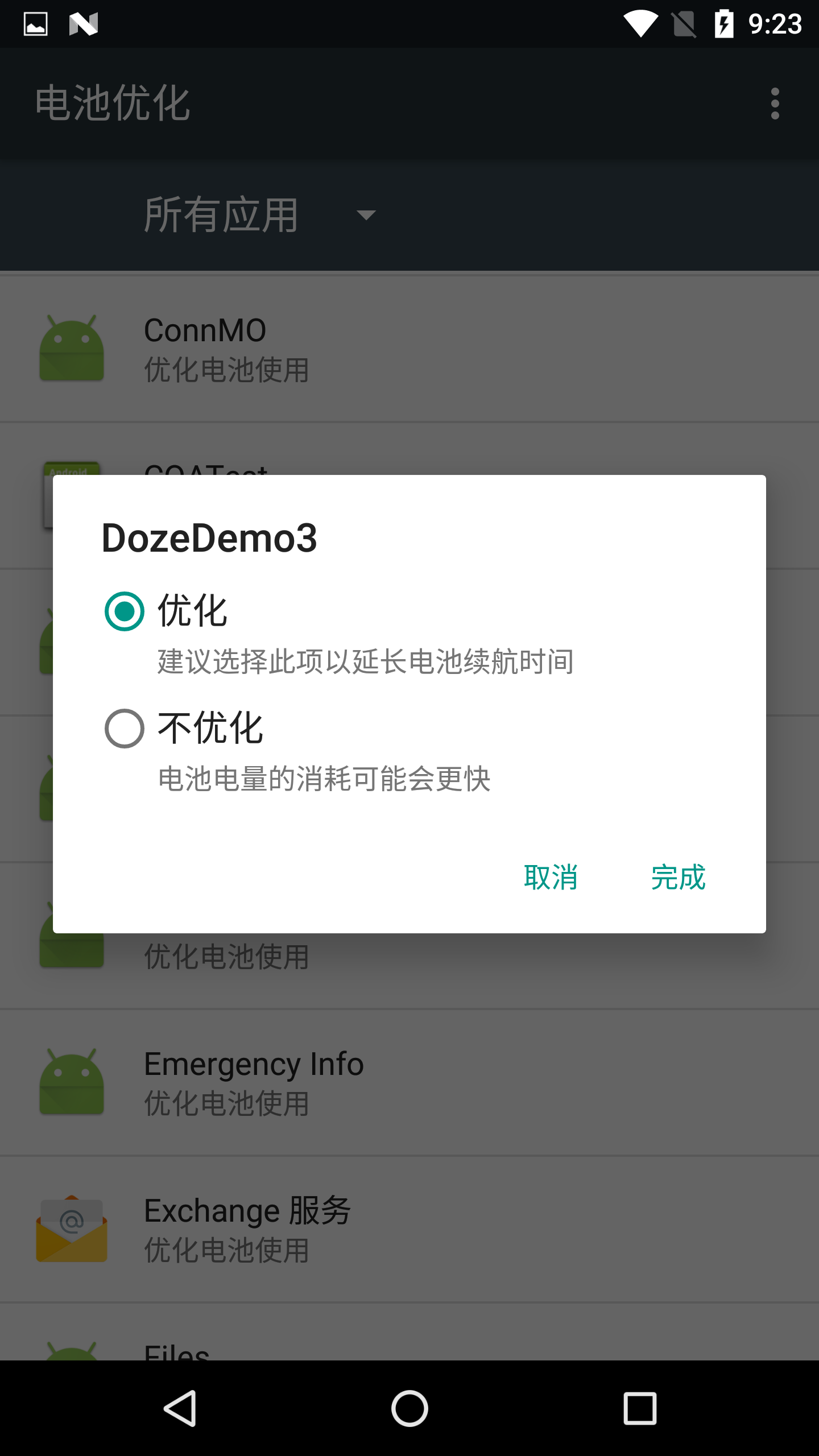
**方案二:**在应用程序运行时,引导用户将该应用添加到白名单中.

实现方式1(不推荐):使用ACTION\_IGNORE\_BATTERY\_OPTIMIZATION\_SETTINGS使用户跳转到电池优化设置页,手动将该应用添加到白名单中.

核心代码如下:

Intent intent = new Intent(Settings.*ACTION\_IGNORE\_BATTERY\_OPTIMIZATION\_SETTINGS*);  
startActivity(intent);

该方法可行,但存在一个缺点:跳转到电池优化页后,用户需要在”所有应用”列表里(应用安装后会默认设置为”优化”)找到该应用进行设置,而且系统提示会引导用户选择优化,如图所示:



实现方式2(推荐):给应用添加权限REQUEST\_IGNORE\_BATTERY\_OPTIMIZATIONS,并使用ACTION\_REQUEST\_IGNORE\_BATTERY\_OPTIMIZATIONS,系统会弹出设置窗口,用户可以直接将该应用添加到白名单中,如下图所示:



核心代码:

在清单文件中添加权限:

<uses-permission android:name="android.permission.REQUEST\_IGNORE\_BATTERY\_OPTIMIZATIONS"/>

使用ACTION\_REQUEST\_IGNORE\_BATTERY\_OPTIMIZATIONS作为参数创建一个Intent,并以"package:com.example.xxx.xxx"的Uri形式将包名传入.

Intent intent = new Intent(Settings.*ACTION\_REQUEST\_IGNORE\_BATTERY\_OPTIMIZATIONS*);  
intent.setData(Uri.*parse*("package:com.example.janiszhang.dozedemo3"));  
startActivity(intent);

方案三:使用GCM(Google Cloud Messaging ),该方法不可行,不再赘述.

**二.后台优化**

优化点1.针对预览版,应用不再接收静态注册的CONNECTIVITY\_ACTION广播.但是应用在前台时仍然能够监听到动态注册的CONNECTIVITY\_CHANGE广播.

优化点2.应用程序不能发送或者接收ACTION\_NEW\_PICTURE 和ACTION\_NEW\_VIDEO广播,这个优化会影响到所有应用,不只是针对预览版.

**应对方案**:

为了应对CONNECTIVITY\_ACTION的变化所带来的影响,官方给出了两种缓解方案.

**方案一**:使用JobScheduler在无计量网络下调度网络任务.

核心代码:

Activity:

在使用JobInfo.Builder()创建JobInfo对象时,调用setRequiredNetworkType()方法,并将

JobInfo.NETWORK\_TYPE\_UNMETERED作为参数传递进去,这段代码的作用是当设备接入无计量网络时将调起MyJobService.

public static final int *MY\_BACKGROUND\_JOB*= 0;  
@TargetApi(Build.VERSION\_CODES.*LOLLIPOP*)  
public static void scheduleJob(Context context) {  
    JobScheduler js =  
            (JobScheduler) context.getSystemService(Context.*JOB\_SCHEDULER\_SERVICE*);  
    JobInfo job = new JobInfo.Builder(  
            *MY\_BACKGROUND\_JOB*,  
            new ComponentName(context, MyJobService.class))  
            .setRequiredNetworkType(JobInfo.*NETWORK\_TYPE\_UNMETERED*)  
            .build();  
    js.schedule(job);  
}

Service:

当条件满足时(在该例中为接入无计量网络),MyJobService中的回调方法onStartJob()将被执行,在实际业务中,可以在这里执行网络任务.

@TargetApi(Build.VERSION\_CODES.*LOLLIPOP*)  
public class MyJobService extends JobService{  
  
    public static final String *TAG*= "MyJobService";  
    @Override  
    public boolean onStartJob(JobParameters jobParameters) {  
        Log.*i*(*TAG*, "on start job: " + jobParameters.getJobId());  
        return false;  
    }  
  
    @Override  
    public boolean onStopJob(JobParameters jobParameters) {  
        Log.*i*(*TAG*, "on stop job: " + jobParameters.getJobId());  
        return false;  
    }  
}

注意:

需要在清单文件中为该Service设置权限:android.permission.BIND\_JOB\_SERVICE”

<service android:name=".MyJobService"  
    android:permission="android.permission.BIND\_JOB\_SERVICE"/>

**方案二**:在应用运行时监控网络连接

方式一:动态注册BroadcastReceiver,监听

“android.net.conn.CONNECTIVITY\_CHANGE”

BroadcastReceiver broadcastReceiver = new BroadcastReceiver() {  
  
    @Override  
    public void onReceive(Context context, Intent intent) {  
        Log.*i*(TAG,"onReceive");  
    }  
};  
  
IntentFilter intentFilter = new IntentFilter();  
intentFilter.addAction("android.net.conn.CONNECTIVITY\_CHANGE");  
this.registerReceiver(broadcastReceiver,intentFilter);

方式二:使用ConnectivityManager

首先,使用NetworkRequest.Builder创建一个NetworkRequest对象,然后使用registerNetworkCallback()把这个NetworkRequest对象传递给系统.当网络条件被满足时,应用将收到一个回调去执行定义在ConnectivityManager.MetworkCallback类中的onAvailable()方法.

ConnectivityManager connectivityManager = (ConnectivityManager) getSystemService(Context.*CONNECTIVITY\_SERVICE*);  
  
NetworkRequest.Builder builder = new NetworkRequest.Builder();  
  
builder.addCapability(NetworkCapabilities.*NET\_CAPABILITY\_NOT\_RESTRICTED*);  
builder.addTransportType(NetworkCapabilities.*TRANSPORT\_WIFI*);  
  
NetworkRequest networkRequest = builder.build();  
  
ConnectivityManager.NetworkCallback networkCallback = new ConnectivityManager.NetworkCallback() {  
  
    @Override  
    public void onAvailable(Network network) {  
        super.onAvailable(network);  
        Log.*i*(*TAG*, "onAvailable");  
    }  
};  
connectivityManager.registerNetworkCallback(networkRequest, networkCallback);

三.**Data Saver**

当用户在计量网络下启用数据节约功能时,系统会封锁后台数据的使用,运行在前台的应用也会尽量少的使用数据流量.用户可以使用白名单允许指定的应用在数据节约模式下使用后台数据.Android N开发者预览版中扩展了ConnectivityManager API的能力,向用户提供了查看和监控数据节约设置的接口.

**检测Data Saver首选项的变化**

动态注册监听ConnectivityManager.ACTION\_RESTRICT\_BACKGROUND\_CHANGED ("android.net.conn.RESTRICT\_BACKGROUND\_CHANGED")的广播接收者.

核心代码:

BroadcastReceiver broadcastReceiver = new BroadcastReceiver() {  
  
    @Override  
    public void onReceive(Context context, Intent intent) {  
        Log.*d*(*TAG*, "Data Saver Changed");  
    }  
};  
  
IntentFilter intentFilter = new IntentFilter();  
intentFilter.addAction("android.net.conn.RESTRICT\_BACKGROUND\_CHANGED");  
this.registerReceiver(broadcastReceiver,intentFilter);

**注意:**必须使用动态注册的方式才能够监听到该广播,不能在清单文件中静态注册.

**检查数据节约设置**

ConnectivityManager的getResrictBackgroundStatus()方法的返回值如下:

RESTRICT\_BACKGROUND\_STATUS\_DISABLED

禁用数据节约

RESTRICT\_BACKGROUND\_STATUS\_ENABLED

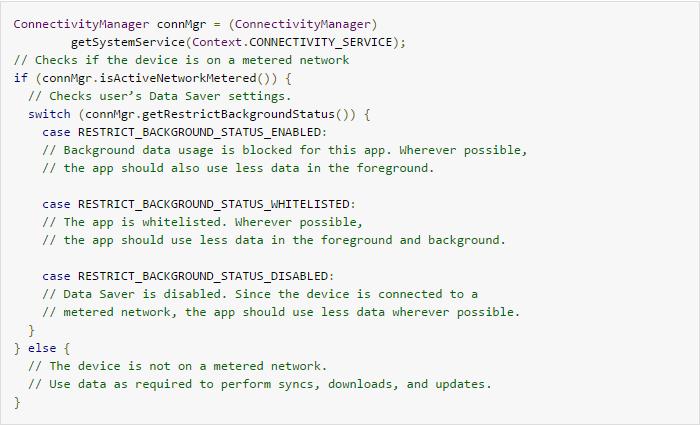
启用数据节约

RESTRICT\_BACKGROUND\_STATUS\_WHITELISTED

用户启用了数据节约,但是该应用在白名单中,故不受限制.

下面的示例代码来自官方文档,给出了使用ConnectivityManager.isActiveNetworkMetered()

 和ConnectivityManager.getRestrictBackgroundStatus()来判断当前Data Saver设置状态的方法.



**注意**:这段代码目前还不能使用,RESTRICT\_BACKGROUND\_STATUS\_ENABLED等三个状态值尚不可用.

**使用adb命令进行测试**

$ adb shell dumpsys netpolicy

生成一个报告,包括当前的全部后台网络限制的设置,白名单中的包的UID,其他已知包的网络限制.

$ adb shell cmd netpolicy

显示一个完整的网络策略管理命令列表.

$ adb shell cmd netpolicy set restrict-background <boolean>

启用或者禁用数据节约命令

$ adb shell cmd netpolicy add restrict-background-whitelist <UID>

将指定的包的UID加入白名单中

$ adb shell cmd netpolicy remove restrict-background-whitelist <UID>

从白名单中将指定包的UID移除

#学习笔记，如有谬误，敬请指正。#

# 从Alarm看Android上层UI到内核代码的流程分析

2017年11月05日 21:59:29 [by\_bai119](https://me.csdn.net/crane_bh) 阅读数：108

Alarm 调用流程，alarm的流程实现了从上层应用一直到下面driver的调用流程，下面简单阐述：  
  
涉及代码;  
./packages/apps/DeskClock/src/com/android/deskclock/Alarms.java  
./frameworks/base/core/java/android/app/AlarmManager.java  
./frameworks/base/services/java/com/android/server/AlarmManagerService.java  
./frameworks/base/services/jni/com\_android\_server\_AlarmManagerService.cpp  
./kernel/kernel/drivers/rtc/alarm-dev.c  
./kernel/kernel/include/linux/android\_alarm.h  
./kernel/kernel/drivers/rtc/alarm.c  
./kernel/kernel/drivers/rtc/interface.c  
./kernel/kernel/drivers/rtc/rtc-pcf8563.c  
  
  
./packages/apps/DeskClock/src/com/android/deskclock/AlarmReceiver.java  
  
  
./kernel/arch/arm/configs/mmp2\_android\_defconfig  
./kernel/kernel/kernel/.config     
  
  
点击Clock 应用程序，然后设置新闹钟，会调到  Alarms.java  里面的  
public static long setAlarm(Context context, Alarm alarm) {  
....  
setNextAlert(context);  
....  
}  
然后这里面也会调用到    
public static void setNextAlert(final Context context) {  
        if (!enableSnoozeAlert(context)) {  
            Alarm alarm = calculateNextAlert(context);   //new 一个新的alarm  
            if (alarm != null) {  
                enableAlert(context, alarm, alarm.time);  
            } else {  
                disableAlert(context);  
            }  
        }  
    }  
然后继续调用到   
private static void enableAlert(Context context, final Alarm alarm, final long atTimeInMillis) {  
.......  
am.set(AlarmManager.RTC\_WAKEUP, atTimeInMillis, sender);   //这里是RTC\_WAKEUP, 这就保证了即使系统睡眠了，都能唤醒，闹钟工作（android平台关机闹钟好像不行）  
.....  
}  
  
然后就调用到了AlarmManager.java 里面方法  
    public void set(int type, long triggerAtTime, PendingIntent operation) {  
        try {  
            mService.set(type, triggerAtTime, operation);  
        } catch (RemoteException ex) {  
        }  
    }  
  
然后就调用到了AlarmManagerService.java  里面方法  
public void set(int type, long triggerAtTime, PendingIntent operation) {  
        setRepeating(type, triggerAtTime, 0, operation);  
    }  
  
然后继续调用  
public void setRepeating(int type, long triggerAtTime, long interval,   
            PendingIntent operation) {  
.....  
synchronized (mLock) {  
            Alarm alarm = new Alarm();  
            alarm.type = type;  
            alarm.when = triggerAtTime;  
            alarm.repeatInterval = interval;  
            alarm.operation = operation;  
  
            // Remove this alarm if already scheduled.  
            removeLocked(operation);  
  
            if (localLOGV) Slog.v(TAG, "set: " + alarm);  
  
            int index = addAlarmLocked(alarm);  
            if (index == 0) {  
                setLocked(alarm);  
            }  
        }  
    }  
  
然后就调用到  
private void setLocked(Alarm alarm)  
    {  
    ......  
    set(mDescriptor, alarm.type, alarmSeconds, alarmNanoseconds);   //mDescriptor  这里的文件是 /dev/alarm  
    .....  
}  
  
这里就调用到jni了  
private native void set(int fd, int type, long seconds, long nanoseconds);  
  
这就调用到了com\_android\_server\_AlarmManagerService.cpp 里面  
static JNINativeMethod sMethods[] = {  
     /\* name, signature, funcPtr \*/  
    {"init", "()I", (void\*)android\_server\_AlarmManagerService\_init},  
    {"close", "(I)V", (void\*)android\_server\_AlarmManagerService\_close},  
    {"set", "(IIJJ)V", (void\*)android\_server\_AlarmManagerService\_set},  
    {"waitForAlarm", "(I)I", (void\*)android\_server\_AlarmManagerService\_waitForAlarm},  
    {"setKernelTimezone", "(II)I", (void\*)android\_server\_AlarmManagerService\_setKernelTimezone},  
};  
  
set 对应的是android\_server\_AlarmManagerService\_set， 具体是  
static void android\_server\_AlarmManagerService\_set(JNIEnv\* env, jobject obj, jint fd, jint type, jlong seconds, jlong nanoseconds)  
{  
#if HAVE\_ANDROID\_OS  
    struct timespec ts;  
    ts.tv\_sec = seconds;  
    ts.tv\_nsec = nanoseconds;  
      
    int result = ioctl(fd, ANDROID\_ALARM\_SET(type), &ts);  
    if (result < 0)  
    {  
        LOGE("Unable to set alarm to %lld.%09lld: %s\n", seconds, nanoseconds, strerror(errno));  
    }  
#endif  
}  
  
然后ioctl 就调用到了alarm-dev.c  
static long alarm\_ioctl(struct file \*file, unsigned int cmd, unsigned long arg)  
{  
....  
    case ANDROID\_ALARM\_SET(0):  
        if (copy\_from\_user(&new\_alarm\_time, (void \_\_user \*)arg,  
            sizeof(new\_alarm\_time))) {  
            rv = -EFAULT;  
            goto err1;  
        }  
from\_old\_alarm\_set:  
        spin\_lock\_irqsave(&alarm\_slock, flags);  
        pr\_alarm(IO, "alarm %d set %ld.%09ld\n", alarm\_type,  
            new\_alarm\_time.tv\_sec, new\_alarm\_time.tv\_nsec);  
        alarm\_enabled |= alarm\_type\_mask;  
        alarm\_start\_range(&alarms[alarm\_type],  
            timespec\_to\_ktime(new\_alarm\_time),  
            timespec\_to\_ktime(new\_alarm\_time));  
        spin\_unlock\_irqrestore(&alarm\_slock, flags);  
        if (ANDROID\_ALARM\_BASE\_CMD(cmd) != ANDROID\_ALARM\_SET\_AND\_WAIT(0)  
            && cmd != ANDROID\_ALARM\_SET\_AND\_WAIT\_OLD)  
            break;  
        /\* fall though \*/  
....  
  
case ANDROID\_ALARM\_SET\_RTC:  
        if (copy\_from\_user(&new\_rtc\_time, (void \_\_user \*)arg,  
            sizeof(new\_rtc\_time))) {  
            rv = -EFAULT;  
            goto err1;  
        }  
        rv = alarm\_set\_rtc(new\_rtc\_time);  
        spin\_lock\_irqsave(&alarm\_slock, flags);  
        alarm\_pending |= ANDROID\_ALARM\_TIME\_CHANGE\_MASK;  
        wake\_up(&alarm\_wait\_queue);  
        spin\_unlock\_irqrestore(&alarm\_slock, flags);  
        if (rv < 0)  
            goto err1;  
        break;  
....  
}  
  
然后这边就调用到了alarm\_start\_range  设置闹钟，   alarm\_set\_rtc  设置RTC  
这两个函数在 android\_alarm.h 声明，在 alarm.c 里实现  
这是android\_alarm.h 里面的声明  
void alarm\_start\_range(struct alarm \*alarm, ktime\_t start, ktime\_t end);  
int alarm\_try\_to\_cancel(struct alarm \*alarm);  
int alarm\_cancel(struct alarm \*alarm);  
ktime\_t alarm\_get\_elapsed\_realtime(void);  
  
/\* set rtc while preserving elapsed realtime \*/  
int alarm\_set\_rtc(const struct timespec ts);  
  
下面看alarm.c 里面实现：  
int alarm\_set\_rtc(struct timespec new\_time)  
{  
....  
ret = rtc\_set\_time(alarm\_rtc\_dev, &rtc\_new\_rtc\_time);  
....  
}  
  
alarm.c  里面实现了 alarm\_suspend  alarm\_resume 函数  
就是如果系统没有suspend的时候，设置闹钟并不会往rtc 芯片的寄存器上写数据，因为不需要唤醒系统，所以闹钟数据时间什么的就通过上层写到设备文件/dev/alarm  
里面就可以了，AlarmThread 会不停的去轮寻下一个时间有没有闹钟，直接从设备文件 /dev/alarm 里面读取  
第二种，系统要是进入susupend的话，alarm 的alarm\_suspend  就会写到下层的rtc芯片的寄存器上去， 然后即使系统suspend之后，闹钟通过rtc 也能唤醒系统  
  
  
  
这里就调用到了interface.c 里面   //这里面 int rtc\_set\_alarm(struct rtc\_device \*rtc, struct rtc\_wkalrm \*alarm) 差不多 也是跟下面一样  
int rtc\_set\_time(struct rtc\_device \*rtc, struct rtc\_time \*tm)  
{  
....  
    err = rtc->ops->set\_time(rtc->dev.parent, tm);  
....  
}  
  
然后set\_time 就看到具体的是那个RTC芯片，这边我们是rtc-pcf8563.c  
static const struct rtc\_class\_ops pcf8563\_rtc\_ops = {  
    .read\_time    = pcf8563\_rtc\_read\_time,  
    .set\_time    = pcf8563\_rtc\_set\_time,  
    .read\_alarm    = pcf8563\_rtc\_read\_alarm,  
    .set\_alarm    = pcf8563\_rtc\_set\_alarm,  
};  
然后就到了  
static int pcf8563\_rtc\_set\_time(struct device \*dev, struct rtc\_time \*tm)  
{  
    unsigned char buf[TIME\_NUM];  
    int ret;  
  
    ret = data\_calc(buf, tm, TIME\_NUM);  
    if (ret < 0)  
        goto out;  
    ret = i2c\_smbus\_write\_i2c\_block\_data(pcf8563\_info->client, PCF8563\_RTC\_SEC, TIME\_NUM, buf);  //这边就调用i2c统一接口，往pcf8563rtc芯片寄存器里面写出数据  
out:  
    return ret;  
}  
  
到此，闹钟时间就已经写到rtc 芯片的寄存器里面，第二个参数就是寄存器的名字，后面的buf就是要写入的时间，rtc芯片是额外供电的，所以系统suspend之后，系统kernel都关了，但是rtc里面还有电，寄存器里面数据还是有的（掉电就会丢失数据），所以闹钟到了，通过硬件中断机制就可以唤醒系统。  
  
上面那个rtc下面有几十个rtc芯片驱动代码，没有结构基本一样，都有基本操作函数，注册函数，都是对各自芯片上特有的寄存器操作，为什么调用的是pcf8563rtc呢？这个要看你系统用的是那个芯片，这个可以通过./kernel/kernel/kernel/.config  查看，这边的pcf8563rtc  是当前系统正在使用的芯片型号    
# CONFIG\_RTC\_DRV\_ISL1208 is not set  
# CONFIG\_RTC\_DRV\_X1205 is not set  
CONFIG\_RTC\_DRV\_PCF8563=y  
# CONFIG\_RTC\_DRV\_PCF8583 is not set  
# CONFIG\_RTC\_DRV\_M41T80 is not set  
  
  
  
下面是系统唤醒之后，闹钟怎么工作的流程，简单阐述  
系统没有suspend的话直接走下面流程，如果suspend的话会被RTC唤醒，然后还是走下面的流程  
  
 private class AlarmThread extends Thread  
    {  
        public AlarmThread()  
        {  
            super("AlarmManager");  
        }  
          
        public void run()  
        {   
        while (true)  
            {  
        int result = waitForAlarm(mDescriptor); //这里调用jni调用static jint android\_server\_AlarmManagerService\_waitForAlarm，主要还是对 /dev/alarm  操作  
        ....  
        Alarm alarm = it.next();  
                        try {  
                            if (localLOGV) Slog.v(TAG, "sending alarm " + alarm);  
                            alarm.operation.send(mContext, 0,  
                                    mBackgroundIntent.putExtra(  
                                            Intent.EXTRA\_ALARM\_COUNT, alarm.count),  
                                    mResultReceiver, mHandler);  
        ....  
        }  
  
    }  
      }  
  
  
static jint android\_server\_AlarmManagerService\_waitForAlarm(JNIEnv\* env, jobject obj, jint fd)  
{  
#if HAVE\_ANDROID\_OS  
    int result = 0;  
      
    do  
    {  
        result = ioctl(fd, ANDROID\_ALARM\_WAIT);  
    } while (result < 0 && errno == EINTR);  
      
    if (result < 0)  
    {  
        LOGE("Unable to wait on alarm: %s\n", strerror(errno));  
        return 0;  
    }  
      
    return result;  
#endif  
}  
  
AlarmManagerService  里面有个AlarmThread  会一直轮询 /dev/alarm文件，如果打开失败就直接返回，成功就会做一些动作，比如查找时间最近的  
alarm，比如睡眠被闹钟唤醒的时候，这边就发一个intent出去，然后在AlarmReceiver.java里面弹出里面会收到就会调用下面的  
        context.startActivity(alarmAlert);  
  
然后弹出alarm  这个界面  
        Class c = AlarmAlert.class;  
其中public class AlarmAlert extends AlarmAlertFullScreen  所以系统睡眠之后被alarm唤醒弹出的alarm就是这边start的  
public class AlarmReceiver extends BroadcastReceiver {  
  
    /\*\* If the alarm is older than STALE\_WINDOW, ignore.  It  
        is probably the result of a time or timezone change \*/  
    private final static int STALE\_WINDOW = 30 \* 60 \* 1000;  
  
    @Override  
    public void onReceive(Context context, Intent intent) {  
    .........  
        Intent alarmAlert = new Intent(context, c);  
        alarmAlert.putExtra(Alarms.ALARM\_INTENT\_EXTRA, alarm);  
        alarmAlert.setFlags(Intent.FLAG\_ACTIVITY\_NEW\_TASK  
                | Intent.FLAG\_ACTIVITY\_NO\_USER\_ACTION);  
        context.startActivity(alarmAlert);  
    ........  
}  
  
到这里alarm 就显示出来了

# Android RTC 自下往上浅析

分类： [Android系统开发](http://www.aiuxian.com/catalog/p-6472.html)  |  标签: android,timezone,struct,server,frameworks,linux内核  |  作者： crycheng [相关](http://www.aiuxian.com/relative/p-46939.html)  |  发布日期 : 2012-02-26  |  热度 : 4°

1.首先搞清楚RTC在kernel内的作用: linux系统有两个时钟：一个是由主板电池驱动的“Real Time Clock”也叫做RTC或者叫CMOS时钟，硬件时钟。当操作系统关机的时候，用这个来记录时间，但是对于运行的系统是不用这个时间的。

另一个时间是 “System clock”也叫内核时钟或者软件时钟，是由软件根据时间中断来进行计数的，内核时钟在系统关机的情况下是不存在的，所以，当操作系统启动的时候，内核时钟是要读取RTC时间来进行时间同步。并且在系统关机的时候将系统时间写回RTC中进行同步。 如前所述，Linux内核与RTC进行互操作的时机只有两个：

1) 内核在启动时从RTC中读取启动时的时间与日期；

2) 内核在需要时将时间与日期回写到RTC中。 系统启动时，内核通过读取RTC来初始化内核时钟,又叫墙上时间，该时间放在xtime变量中。

The current time of day (the wall time) is defined in kernel/timer.c:

struct timespec xtime;

The timespec data structure is defined in <linux/time.h> as:

struct timespec {

　　time\_t tv\_sec; /\* seconds \*/

　　long tv\_nsec; /\* nanoseconds \*/

};

最有可能读取RTC设置内核时钟的位置应该在arch/arm/kernel/time.c里的time\_init函数内. time.c为系统的时钟驱动部分.

time\_init函数会在系统初始化时,由init/main.c里的start\_kernel函数内调用. ARM架构的time\_init代码如下:

/\* arch/arm/kernel/time.c \*/

void \_\_init time\_init(void)

{

system\_timer = machine\_desc->timer;

system\_timer->init();

#ifdef CONFIG\_HAVE\_SCHED\_CLOCK

sched\_clock\_postinit();

#endif

}

2.RTC结构部分 

static const struct rtc\_class\_ops hym8563\_rtc\_ops = {

.read\_time = hym8563\_rtc\_read\_time,

.set\_time = hym8563\_rtc\_set\_time,

.read\_alarm = hym8563\_rtc\_read\_alarm,

.set\_alarm = hym8563\_rtc\_set\_alarm,

.ioctl = hym8563\_rtc\_ioctl,

.proc = hym8563\_rtc\_proc

};

static int \_\_devinit hym8563\_probe(struct i2c\_client \*client, const struct i2c\_device\_id \*id)

{

int rc = 0;

u8 reg = 0;

struct hym8563 \*hym8563;

struct rtc\_device \*rtc = NULL;

struct rtc\_time tm\_read, tm = {

.tm\_wday = 6,

.tm\_year = 111,

.tm\_mon = 0,

.tm\_mday = 1,

.tm\_hour = 12,

.tm\_min = 0,

.tm\_sec = 0,

};

if (!i2c\_check\_functionality(client->adapter, I2C\_FUNC\_I2C))

return -ENODEV;

hym8563 = kzalloc(sizeof(struct hym8563), GFP\_KERNEL);

if (!hym8563) {

return -ENOMEM;

}

gClient = client;

hym8563->client = client;

mutex\_init(&hym8563->mutex);

wake\_lock\_init(&hym8563->wake\_lock, WAKE\_LOCK\_SUSPEND, "rtc\_hym8563");

INIT\_WORK(&hym8563->work, hym8563\_work\_func);

i2c\_set\_clientdata(client, hym8563);

hym8563\_init\_device(client);

// check power down

hym8563\_i2c\_read\_regs(client,RTC\_SEC,®,1);

if (reg&0x80) {

dev\_info(&client->dev, "clock/calendar information is no longer guaranteed\n");

hym8563\_set\_time(client, &tm);

}

hym8563\_read\_datetime(client, &tm\_read); //read time from hym8563

if(((tm\_read.tm\_year < 70) | (tm\_read.tm\_year > 137 )) | (tm\_read.tm\_mon == -1) | (rtc\_valid\_tm(&tm\_read) != 0)) //if the hym8563 haven't initialized

{

hym8563\_set\_time(client, &tm); //initialize the hym8563

}

if(gpio\_request(client->irq, "rtc gpio"))

{

dev\_err(&client->dev, "gpio request fail\n");

gpio\_free(client->irq);

goto exit;

}

hym8563->irq = gpio\_to\_irq(client->irq);

gpio\_pull\_updown(client->irq,GPIOPullUp);

if (request\_irq(hym8563->irq, hym8563\_wakeup\_irq, IRQF\_TRIGGER\_FALLING, client->dev.driver->name, hym8563) < 0)

{

printk("unable to request rtc irq\n");

goto exit;

}

enable\_irq\_wake(hym8563->irq);

rtc = rtc\_device\_register(client->name, &client->dev,

&hym8563\_rtc\_ops, THIS\_MODULE);

if (IS\_ERR(rtc)) {

rc = PTR\_ERR(rtc);

rtc = NULL;

goto exit;

}

hym8563->rtc = rtc;

return 0;

exit:

if (rtc)

rtc\_device\_unregister(rtc);

if (hym8563)

kfree(hym8563);

return rc;

}

看这两个结构体，我认为就已经达到目的，第2个结构体是平台设备中的driver部分，也就是hym8563\_probe,是个很重要的函数，在这里面，第1个结构体被顺利注册进rtc子系统。Rtc的所用到的结构体被定义在，LINUX/include/linux/rtc.h里面。

struct rtc\_device这个结构体是核心部分，内核中就是靠它传递信息，不管在哪使用，都要靠它间接的调用底层信息。比如在alarm.c 中。

alarm\_ioctl这个函数中，多次使用了rtc\_set\_time/rtc\_get\_time，这些函数虽然是定义在rtc目录下的interface.c 中，但实质还是rtc-hym8563.c中结构体 rtc\_class\_ops所指过去的函数。

也就是说在和内核层以上的交互是通过alarm-dev.c里面的alarm\_ioctl及其余的函数交互，但是在这个文件里面的rtc\_set\_time/rtc\_get\_time操作是为了设置RTC时间等的操作是调用alarm.c里面的函数，但是alarm.c驱动本身和硬件没有关系，在这里屏蔽了RTC的硬件操作，比如HYM8563的时间I2C硬件驱动操作在rtc-HYM8563.c驱动里，只需要使用 rtc\_class\_ops进行注册就可以了，完整的实现了硬件对平台无关性的屏蔽。

那么我可以告诉你了，为什么多了一个alarm.c ，因为在android中它为了使得平台无关性提高，因此大量的增加过渡代码层，HAL就是这种性质的存在。alarm.c在用户空间中会多一个/dev/alarm 节点，而rtc-hym8563.c.c 会产生/dev/rtc这样的节点。

3.JNI层

namespace android {

static jint android\_server\_AlarmManagerService\_setKernelTimezone(JNIEnv\* env, jobject obj, jint fd, jint minswest)

{

struct timezone tz;

tz.tz\_minuteswest = minswest;

tz.tz\_dsttime = 0;

int result = settimeofday(NULL, &tz);

if (result < 0) {

LOGE("Unable to set kernel timezone to %d: %s\n", minswest, strerror(errno));

return -1;

} else {

LOGD("Kernel timezone updated to %d minutes west of GMT\n", minswest);

}

return 0;

}

static jint android\_server\_AlarmManagerService\_init(JNIEnv\* env, jobject obj)

{

return open("/dev/alarm", O\_RDWR);

}

static void android\_server\_AlarmManagerService\_close(JNIEnv\* env, jobject obj, jint fd)

{

close(fd);

}

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

{

struct timespec ts;

ts.tv\_sec = seconds;

ts.tv\_nsec = nanoseconds;

int result = ioctl(fd, ANDROID\_ALARM\_SET(type), &ts);

if (result < 0)

{

LOGE("Unable to set alarm to %lld.%09lld: %s\n", seconds, nanoseconds, strerror(errno));

}

}

static jint android\_server\_AlarmManagerService\_waitForAlarm(JNIEnv\* env, jobject obj, jint fd)

{

int result = 0;

do

{

result = ioctl(fd, ANDROID\_ALARM\_WAIT);

} while (result < 0 && errno == EINTR);

if (result < 0)

{

LOGE("Unable to wait on alarm: %s\n", strerror(errno));

return 0;

}

return result;

}

static JNINativeMethod sMethods[] = {

/\* name, signature, funcPtr \*/

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

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

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

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

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

};

int register\_android\_server\_AlarmManagerService(JNIEnv\* env)

{

return jniRegisterNativeMethods(env, "com/android/server/AlarmManagerService",

sMethods, NELEM(sMethods));

}

} /\* namespace android \*/

其实在JNI层这里RTC就和其余的模块一样，直接去通过打开/关闭/设置/等待等来操作节点/dev/alarm和底层进行通信，不仔细解释。

**4、 framework层**

frameworks/base/services/java/com/android/server/AlarmManagerService.java   
    frameworks/base/core/java/android/app/AlarmManager.java

下面的是直接提供给app层的API接口，它是AlarmManagerService.java的一个封装。

这里只是简单的解释下service到底在此做什么了。

其实也没做什么，仅仅是把上面分析的JNI拿来在此调用一下而已。然后包装一下，将功能实现得更完美些。

5.App层

package android.app;

import android.content.Context;

import android.content.Intent;

import android.os.RemoteException;

import android.os.ServiceManager;

public class AlarmManager

{

public static final int RTC\_WAKEUP = 0;

public static final int RTC = 1;

public static final int ELAPSED\_REALTIME\_WAKEUP = 2;

public static final int ELAPSED\_REALTIME = 3;

private final IAlarmManager mService;

AlarmManager(IAlarmManager service) {

mService = service;

}

public void set(int type, long triggerAtTime, PendingIntent operation) {

try {

mService.set(type, triggerAtTime, operation);

} catch (RemoteException ex) {

}

}

public void setRepeating(int type, long triggerAtTime, long interval,

PendingIntent operation) {

try {

mService.setRepeating(type, triggerAtTime, interval, operation);

} catch (RemoteException ex) {

}

}

public static final long INTERVAL\_FIFTEEN\_MINUTES = 15 \* 60 \* 1000;

public static final long INTERVAL\_HALF\_HOUR = 2\*INTERVAL\_FIFTEEN\_MINUTES;

public static final long INTERVAL\_HOUR = 2\*INTERVAL\_HALF\_HOUR;

public static final long INTERVAL\_HALF\_DAY = 12\*INTERVAL\_HOUR;

public static final long INTERVAL\_DAY = 2\*INTERVAL\_HALF\_DAY;

public void setInexactRepeating(int type, long triggerAtTime, long interval,

PendingIntent operation) {

try {

mService.setInexactRepeating(type, triggerAtTime, interval, operation);

} catch (RemoteException ex) {

}

}

public void cancel(PendingIntent operation) {

try {

mService.remove(operation);

} catch (RemoteException ex) {

}

}

public void setTime(long millis) {

try {

mService.setTime(millis);

} catch (RemoteException ex) {

}

}

public void setTimeZone(String timeZone) {

try {

mService.setTimeZone(timeZone);

} catch (RemoteException ex) {

}

}

}

frameworks\base\core\java\android\app  这个目录下，就是系统自带定时器的源代码，比如Alarms.java 中：第一个导入的包就是 import android.app.AlarmManager。