**Android5.0 Alarm服务简单分析**

2017年02月07日 23:50:52 [freeMos](https://me.csdn.net/lgglkk) 阅读数：641 标签： [alarm](http://so.csdn.net/so/search/s.do?q=alarm&t=blog)[batch](http://so.csdn.net/so/search/s.do?q=batch&t=blog) 更多

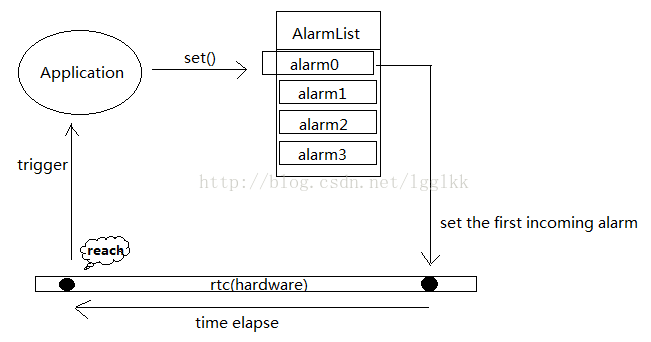
个人分类： [PMS](https://blog.csdn.net/lgglkk/article/category/6226388)

版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/lgglkk/article/details/54918362

**Alarm的触发过程**

下图1是alarm触发的一个大致流程。

首先应用会调用AlarmManager提供的set接口，将一个含有触发时间和触发事件的alarm对象添加到一个由AlarmManager服务维护的alarm列表中，同时alarm列表会计算出最优先的alarm对象（最先起来）设置到底层RTC设备中。但这个时间点到了，RTC会上发信息并通知AlarmManager服务去触发相应的事件。



**Alarm管理机制的变化**

Android4.4以及后面的版本对Alarm的管理发生了比较大的变化，首先来看下AlarmManager的构造方法。

1. AlarmManager(IAlarmManager service, Context ctx) {
2. mService = service;
4. final int sdkVersion = ctx.getApplicationInfo().targetSdkVersion;
5. mAlwaysExact = (sdkVersion < Build.VERSION\_CODES.KITKAT);
6. }
7. private long legacyExactLength() {
8. return (mAlwaysExact ? WINDOW\_EXACT : WINDOW\_HEURISTIC);
9. }

从Android 4.4版本开始引入了Exact闹钟的概念，既然有Exact（精确），那自然有非精确，但为了兼容旧版本，可以看到legacyExactLength方法中，如果sdkVersion是API19（Android4.4）之前的，所有的闹钟都是精确的闹钟，否则是Heuristic的。

接下来看看set方法的注释：

     \* 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.

     \* With the new batching policy, delivery ordering guarantees are not as

     \* strong as they were previously.  If the application sets multiple alarms,

     \* it is possible that these alarms' <em>actual</em> delivery ordering may not match

     \* the order of their <em>requested</em> delivery times.  If your application has

     \* strong ordering requirements there are other APIs that you can use to get

     \* the necessary behavior; see {@link #setWindow(int, long, long, PendingIntent)}

     \* and {@link #setExact(int, long, PendingIntent)}.

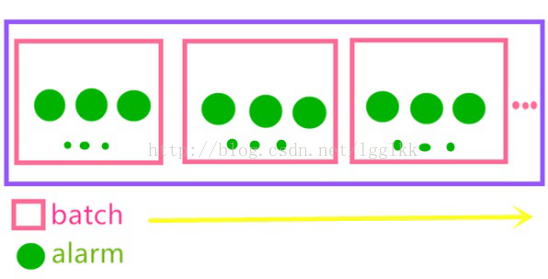
     \* Applications whose {@code targetSdkVersion} is before API 19 will

     \* continue to get the previous alarm behavior: all of their scheduled alarms

     \* will be treated as exact.

翻译：从API19开始，通过这个方法设置的alarm都被认为是非精确的，这个alarm不会再设置的时间点之前触发，但可能会再设置的时间点后适当延迟触发。**Android系统会根据一套规则，将一组触发时间类似的alarm以batch的形式保存下来，一旦batch的时间到了，便会将这个batch中的alarm一起触发，这样做的目的是将系统唤醒（如果是wake up类型的闹钟，首先需要唤醒系统）的次数降到最低，以达到省电的目的。**

Batch的结构如图2：



# [Android之AlarmManager(全局定时器/闹钟）指定时长或以周期形式执行某项操作](https://www.cnblogs.com/zyw-205520/p/4040923.html)

**1、AlarmManager，顾名思义，就是“提醒”，是Android中常用的一种系统级别的提示服务，可以实现从指定时间开始，以一个固定的间隔时间执行某项操作，所以常常与广播（Broadcast）连用，实现闹钟等提示功能**

**2、AlarmManager的常用方法有三个：**

**（1）set(int type，long startTime，PendingIntent pi)；**

**该方法用于设置一次性闹钟，第一个参数表示闹钟类型，第二个参数表示闹钟执行时间，第三个参数表示闹钟响应动作。**

**（2）setRepeating(int type，long startTime，long intervalTime，PendingIntent pi)；**

**该方法用于设置重复闹钟，第一个参数表示闹钟类型，第二个参数表示闹钟首次执行时间，第三个参数表示闹钟两次执行的间隔时间，第三个参数表示闹钟响应动作。**

**（3）setInexactRepeating（int type，long startTime，long intervalTime，PendingIntent pi）；**

**该方法也用于设置重复闹钟，与第二个方法相似，不过其两个闹钟执行的间隔时间不是固定的而已。**

**3、三个方法各个参数详悉：**

**（1）int type：闹钟的类型，常用的有5个值：AlarmManager.ELAPSED\_REALTIME、AlarmManager.ELAPSED\_REALTIME\_WAKEUP、AlarmManager.RTC、AlarmManager.RTC\_WAKEUP、AlarmManager.POWER\_OFF\_WAKEUP。**

**AlarmManager.ELAPSED\_REALTIME表示闹钟在手机睡眠状态下不可用，该状态下闹钟使用相对时间（相对于系统启动开始），状态值为3；**

**AlarmManager.ELAPSED\_REALTIME\_WAKEUP表示闹钟在睡眠状态下会唤醒系统并执行提示功能，该状态下闹钟也使用相对时间，状态值为2；**

**AlarmManager.RTC表示闹钟在睡眠状态下不可用，该状态下闹钟使用绝对时间，即当前系统时间，状态值为1；**

**AlarmManager.RTC\_WAKEUP表示闹钟在睡眠状态下会唤醒系统并执行提示功能，该状态下闹钟使用绝对时间，状态值为0；**

**AlarmManager.POWER\_OFF\_WAKEUP表示闹钟在手机关机状态下也能正常进行提示功能，所以是5个状态中用的最多的状态之一，该状态下闹钟也是用绝对时间，状态值为4；不过本状态好像受SDK版本影响，某些版本并不支持；**

**（2）long startTime：闹钟的第一次执行时间，以毫秒为单位，可以自定义时间，不过一般使用当前时间。需要注意的是，本属性与第一个属性（type）密切相关，如果第一个参数对应的闹钟使用的是相对时间（ELAPSED\_REALTIME和ELAPSED\_REALTIME\_WAKEUP），那么本属性就得使用相对时间（相对于系统启动时间来说），比如当前时间就表示为：SystemClock.elapsedRealtime()；如果第一个参数对应的闹钟使用的是绝对时间（RTC、RTC\_WAKEUP、POWER\_OFF\_WAKEUP），那么本属性就得使用绝对时间，比如当前时间就表示为：System.currentTimeMillis()。**

**（3）long intervalTime：对于后两个方法来说，存在本属性，表示两次闹钟执行的间隔时间，也是以毫秒为单位。**

**（4）PendingIntent pi：是闹钟的执行动作，比如发送一个广播、给出提示等等。PendingIntent是Intent的封装类。需要注意的是，如果是通过启动服务来实现闹钟提示的话，PendingIntent对象的获取就应该采用Pending.getService(Context c,int i,Intent intent,int j)方法；如果是通过广播来实现闹钟提示的话，PendingIntent对象的获取就应该采用PendingIntent.getBroadcast(Context c,int i,Intent intent,int j)方法；如果是采用Activity的方式来实现闹钟提示的话，PendingIntent对象的获取就应该采用PendingIntent.getActivity(Context c,int i,Intent intent,int j)方法。如果这三种方法错用了的话，虽然不会报错，但是看不到闹钟提示效果。**

AlarmManager的使用机制有的称呼为全局定时器，有的称呼为闹钟。通过对它的使用，个人觉得叫全局定时器比较合适，其实它的作用和Timer有点相似。都有两种相似的用法：（1）在指定时长后执行某项操作（2）周期性的执行某项操作

AlarmManager对象配合Intent使用，可以定时的开启一个Activity,发送一个BroadCast,或者开启一个Service.

下面的代码详细的介绍了两种定时方式的使用：

**（1）在指定时长后执行某项操作**

[复制代码](javascript:void(0);)

//操作：发送一个广播，广播接收后Toast提示定时操作完成

Intent intent =new Intent(Main.this, alarmreceiver.class);

intent.setAction("short");

PendingIntent sender=

PendingIntent.getBroadcast(Main.this, 0, intent, 0);

//设定一个五秒后的时间

Calendar calendar=Calendar.getInstance();

calendar.setTimeInMillis(System.currentTimeMillis());

calendar.add(Calendar.SECOND, 5);

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

alarm.set(AlarmManager.RTC\_WAKEUP, calendar.getTimeInMillis(), sender);

//或者以下面方式简化

//alarm.set(AlarmManager.RTC\_WAKEUP, System.currentTimeMillis()+5\*1000, sender);

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

//注意：receiver记得在manifest.xml注册

public static class alarmreceiver extends BroadcastReceiver{

@Override

public void onReceive(Context context, Intent intent) {

// TODO Auto-generated method stub

if(intent.getAction().equals("short")){

Toast.makeText(context, "short alarm", Toast.LENGTH\_LONG).show();

}else{

Toast.makeText(context, "repeating alarm",Toast.LENGTH\_LONG).show();

}

}

}

[复制代码](javascript:void(0);)

**AlarmManager类型如下：**

AlarmManager.RTC，硬件闹钟，不唤醒手机（也可能是其它设备）休眠；当手机休眠时不发射闹钟。

AlarmManager.RTC\_WAKEUP，硬件闹钟，当闹钟发射时唤醒手机休眠；

AlarmManager.ELAPSED\_REALTIME，真实时间流逝闹钟，不唤醒手机休眠；当手机休眠时不发射闹钟。

AlarmManager.ELAPSED\_REALTIME\_WAKEUP，真实时间流逝闹钟，当闹钟发射时唤醒手机休眠；

RTC闹钟和ELAPSED\_REALTIME最大的差别就是前者可以通过修改手机时间触发闹钟事件，后者要通过真实时间的流逝，即使在休眠状态，时间也会被计算。

**（2）周期性的执行某项操作**

[复制代码](javascript:void(0);)

Intent intent =new Intent(Main.this, alarmreceiver.class);

intent.setAction("repeating");

PendingIntent sender=PendingIntent

.getBroadcast(Main.this, 0, intent, 0);

//开始时间

long firstime=SystemClock.elapsedRealtime();

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

　//5秒一个周期，不停的发送广播

am.setRepeating(AlarmManager.ELAPSED\_REALTIME\_WAKEUP, firstime, 5\*1000, sender);

[复制代码](javascript:void(0);)

AlarmManager的setRepeating()相当于Timer的Schedule(task,delay,peroid);有点差异的地方时Timer这个方法是指定延迟多长时间以后开始周期性的执行task;

AlarmManager的取消：（其中需要注意的是取消的Intent必须与启动Intent保持绝对一致才能支持取消AlarmManager）

Intent intent =new Intent(Main.this, alarmreceiver.class);

intent.setAction("repeating");

PendingIntent sender=PendingIntent

.getBroadcast(Main.this, 0, intent, 0);

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

alarm.cancel(sender);

不要让昨天的沮丧 让今天的梦想黯然失色 成功的人总是修改方法而不修改目标 微信公众号：javenlife

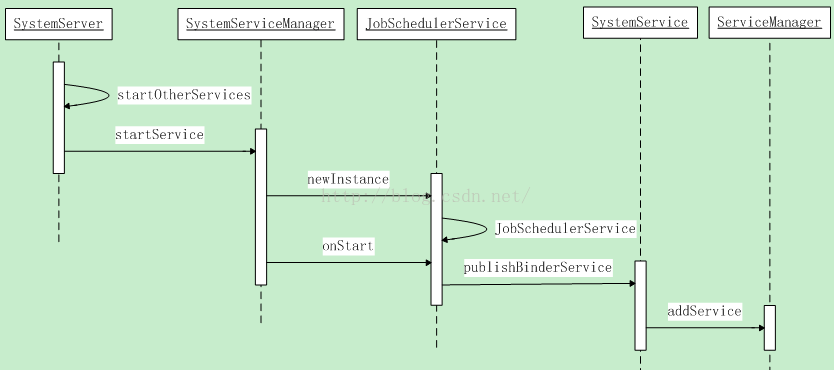
**Android 6.0 之AlarmManagerService源码分析**

2017年09月06日 15:39:46 [\_dowork](https://me.csdn.net/q1183345443) 阅读数：1240

AlarmManager实质上是一个全局定时器，是Android中常用的一种系统服务级别的提示服务，在指定时间或周期性启动其他组件（Activity、Service、BroadcastReceiver）。

之前有篇文章专门介绍了AlarmManager定时器的使用方法，获取到服务后，调用该服务的一些设置方法，在设定时间到达后就会启动指定的组件。

AlarmManagerService和JobSchedulerService一样都是系统服务，故它们的启动流程也类似，先看下时序图：



Zygote进程启动后会启动System进程，在System进程启动过程中会启动系统中的关键服务，如AMS、PMS、JobSchedulerService以及这里要分析的AlarmManagerService。

SystemServer启动AlarmManagerService服务调用的是SystemServiceManager类的startService方法：

**[java]** [view plain](http://blog.csdn.net/zhangyongfeiyong/article/details/52224300) [copy](http://blog.csdn.net/zhangyongfeiyong/article/details/52224300)

1. **private** **void** startOtherServices() {
2. **try** {
3. . . .
4. mSystemServiceManager.startService(AlarmManagerService.**class**);
5. . . .
6. } **catch** (RuntimeException e) {
7. Slog.e("System", "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");
8. Slog.e("System", "\*\*\*\*\*\*\*\*\*\*\*\* Failure starting core service", e);
9. }
10. }

SystemServiceManager类的startService方法在JobSchedulerService的分析中已经分析过，这里粘贴写代码：

**[java]** [view plain](http://blog.csdn.net/zhangyongfeiyong/article/details/52224300) [copy](http://blog.csdn.net/zhangyongfeiyong/article/details/52224300)

1. **private** **final** ArrayList<SystemService> mServices = **new** ArrayList<SystemService>();
3. /\*\*
4. \* 创建并启动一个继承自SystemService类的系统服务。
5. \*
6. \* @param 一个继承自SystemService类的服务类
7. \* @return 服务类的实例
8. \* @throws 如果服务启动失败则抛RuntimeException异常
9. \*/
10. @SuppressWarnings("unchecked")
11. **public** <T **extends** SystemService> T startService(Class<T> serviceClass) {
12. // 获取服务类的类名
13. **final** String name = serviceClass.getName();
14. Slog.i(TAG, "Starting " + name);
16. // 判断服务类是否是SystemService的子类
17. **if** (!SystemService.**class**.isAssignableFrom(serviceClass)) {
18. **throw** **new** RuntimeException("Failed to create " + name
19. + ": service must extend " + SystemService.**class**.getName());
20. }
21. **final** T service;
22. **try** {
23. // 获取服务类包含一个Context参数的构造方法
24. Constructor<T> constructor = serviceClass.getConstructor(Context.**class**);
25. // 创建这个服务类的实例
26. service = constructor.newInstance(mContext);
27. } **catch** (InstantiationException ex) {
28. **throw** **new** RuntimeException("Failed to create service " + name
29. + ": service could not be instantiated", ex);
30. } **catch** (IllegalAccessException ex) {
31. **throw** **new** RuntimeException("Failed to create service " + name
32. + ": service must have a public constructor with a Context argument", ex);
33. } **catch** (NoSuchMethodException ex) {
34. **throw** **new** RuntimeException("Failed to create service " + name
35. + ": service must have a public constructor with a Context argument", ex);
36. } **catch** (InvocationTargetException ex) {
37. **throw** **new** RuntimeException("Failed to create service " + name
38. + ": service constructor threw an exception", ex);
39. }
41. // 把服务添加到mServices列表中，方便后续使用时取出
42. mServices.add(service);
44. **try** {
45. // 回调服务的onStart方法
46. service.onStart();
47. } **catch** (RuntimeException ex) {
48. **throw** **new** RuntimeException("Failed to start service " + name
49. + ": onStart threw an exception", ex);
50. }
51. **return** service;
52. }

在开启AlarmManagerService服务时，会创建服务的实例，看下该服务的创建过程：

**[java]** [view plain](http://blog.csdn.net/zhangyongfeiyong/article/details/52224300) [copy](http://blog.csdn.net/zhangyongfeiyong/article/details/52224300)

1. **final** AlarmHandler mHandler = **new** AlarmHandler();
2. **final** Constants mConstants;
4. **public** AlarmManagerService(Context context) {
5. **super**(context);
6. // 初始化Handler和常量Constants类
7. mConstants = **new** Constants(mHandler);
8. }

初始化Handler的代码后面调用时再分析，这里先看下常量类的实现：

**[java]** [view plain](http://blog.csdn.net/zhangyongfeiyong/article/details/52224300) [copy](http://blog.csdn.net/zhangyongfeiyong/article/details/52224300)

1. /\*\*
2. \* 该类中所有的时间单位都是毫秒。
3. \* 这些常量保持与系统全局设置一致。
4. \* 任何访问该类或该类中的字段都要持有AlarmManagerService.mLock锁
5. \*/
6. **private** **final** **class** Constants **extends** ContentObserver {
7. // 在设置中保存的键值
8. **private** **static** **final** String KEY\_MIN\_FUTURITY = "min\_futurity";
9. **private** **static** **final** String KEY\_MIN\_INTERVAL = "min\_interval";
10. **private** **static** **final** String KEY\_ALLOW\_WHILE\_IDLE\_SHORT\_TIME = "allow\_while\_idle\_short\_time";
11. **private** **static** **final** String KEY\_ALLOW\_WHILE\_IDLE\_LONG\_TIME = "allow\_while\_idle\_long\_time";
12. **private** **static** **final** String KEY\_ALLOW\_WHILE\_IDLE\_WHITELIST\_DURATION
13. = "allow\_while\_idle\_whitelist\_duration";
15. **private** **static** **final** **long** DEFAULT\_MIN\_FUTURITY = 5 \* 1000;
16. **private** **static** **final** **long** DEFAULT\_MIN\_INTERVAL = 60 \* 1000;
17. **private** **static** **final** **long** DEFAULT\_ALLOW\_WHILE\_IDLE\_SHORT\_TIME = DEFAULT\_MIN\_FUTURITY;
18. **private** **static** **final** **long** DEFAULT\_ALLOW\_WHILE\_IDLE\_LONG\_TIME = 9\*60\*1000;
19. **private** **static** **final** **long** DEFAULT\_ALLOW\_WHILE\_IDLE\_WHITELIST\_DURATION = 10\*1000;
21. // Minimum futurity of a new alarm
22. **public** **long** MIN\_FUTURITY = DEFAULT\_MIN\_FUTURITY;
24. // Minimum alarm recurrence interval
25. **public** **long** MIN\_INTERVAL = DEFAULT\_MIN\_INTERVAL;
27. // 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
28. **public** **long** ALLOW\_WHILE\_IDLE\_SHORT\_TIME = DEFAULT\_ALLOW\_WHILE\_IDLE\_SHORT\_TIME;
30. // 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
31. **public** **long** ALLOW\_WHILE\_IDLE\_LONG\_TIME = DEFAULT\_ALLOW\_WHILE\_IDLE\_LONG\_TIME;
33. // BroadcastOptions.setTemporaryAppWhitelistDuration() to use for FLAG\_ALLOW\_WHILE\_IDLE.
34. **public** **long** ALLOW\_WHILE\_IDLE\_WHITELIST\_DURATION
35. = DEFAULT\_ALLOW\_WHILE\_IDLE\_WHITELIST\_DURATION;
37. **private** ContentResolver mResolver;
38. **private** **final** KeyValueListParser mParser = **new** KeyValueListParser(',');
39. **private** **long** mLastAllowWhileIdleWhitelistDuration = -1;
41. **public** Constants(Handler handler) {
42. **super**(handler);
43. // 更新可以开始执行flag为ALLOW\_WHILE\_IDLE的alarm的最小时间间隔
44. updateAllowWhileIdleMinTimeLocked();
45. updateAllowWhileIdleWhitelistDurationLocked();
46. }
48. // 系统启动后会调用该方法，注册数据库监听
49. **public** **void** start(ContentResolver resolver) {
50. mResolver = resolver;
51. // 监听数据库变化
52. mResolver.registerContentObserver(Settings.Global.getUriFor(
53. Settings.Global.ALARM\_MANAGER\_CONSTANTS), **false**, **this**);
54. updateConstants();
55. }
57. // 更新可以开始执行flag为ALLOW\_WHILE\_IDLE的alarm的最小时间间隔
58. **public** **void** updateAllowWhileIdleMinTimeLocked() {
59. mAllowWhileIdleMinTime = mPendingIdleUntil != **null**
60. ? ALLOW\_WHILE\_IDLE\_LONG\_TIME : ALLOW\_WHILE\_IDLE\_SHORT\_TIME;
61. }
63. **public** **void** updateAllowWhileIdleWhitelistDurationLocked() {
64. **if** (mLastAllowWhileIdleWhitelistDuration != ALLOW\_WHILE\_IDLE\_WHITELIST\_DURATION) {
65. mLastAllowWhileIdleWhitelistDuration = ALLOW\_WHILE\_IDLE\_WHITELIST\_DURATION;
66. BroadcastOptions opts = BroadcastOptions.makeBasic();
67. opts.setTemporaryAppWhitelistDuration(ALLOW\_WHILE\_IDLE\_WHITELIST\_DURATION);
68. mIdleOptions = opts.toBundle();
69. }
70. }
72. @Override
73. **public** **void** onChange(**boolean** selfChange, Uri uri) {
74. // 数据库内容变化时，更新一些本地变量
75. updateConstants();
76. }
77. . . .
78. }

看下AlarmManagerService类的onStart方法：

**[java]** [view plain](http://blog.csdn.net/zhangyongfeiyong/article/details/52224300) [copy](http://blog.csdn.net/zhangyongfeiyong/article/details/52224300)

1. **long** mNativeData;
2. // 下一个包含wakeup的batch的start时间
3. **private** **long** mNextWakeup;
4. // 下一个包含Rtc wakeup的batch的start时间
5. **private** **long** mNextRtcWakeup;
6. // 下一个非wakeup的batch的start时间
7. **private** **long** mNextNonWakeup;
8. **static** **final** String TIMEZONE\_PROPERTY = "persist.sys.timezone";
9. PowerManager.WakeLock mWakeLock;
10. // 时间变化发送者
11. PendingIntent mTimeTickSender;
12. // 日期变化发送者
13. PendingIntent mDateChangeSender;
14. // 时间变化的广播接收者
15. ClockReceiver mClockReceiver;
16. // 监听息屏/亮屏的广播接收者
17. InteractiveStateReceiver mInteractiveStateReceiver;
18. // 监听卸载的广播接收者
19. **private** UninstallReceiver mUninstallReceiver;
21. @Override
22. **public** **void** onStart() {
23. // 通过JNI对mNativeData进行初始化操作:
24. // 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
25. mNativeData = init();
26. mNextWakeup = mNextRtcWakeup = mNextNonWakeup = 0;
28. // 把当前时区信息保存到内核
29. setTimeZoneImpl(SystemProperties.get(TIMEZONE\_PROPERTY));
31. // 初始化wakelock
32. PowerManager pm = (PowerManager) getContext().getSystemService(Context.POWER\_SERVICE);
33. mWakeLock = pm.newWakeLock(PowerManager.PARTIAL\_WAKE\_LOCK, "\*alarm\*");
35. // 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
36. mTimeTickSender = PendingIntent.getBroadcastAsUser(getContext(), 0,
37. **new** Intent(Intent.ACTION\_TIME\_TICK).addFlags(
38. Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY
39. | Intent.FLAG\_RECEIVER\_FOREGROUND), 0,
40. UserHandle.ALL);
41. // 初始化mDateChangeSender
42. Intent intent = **new** Intent(Intent.ACTION\_DATE\_CHANGED);
43. intent.addFlags(Intent.FLAG\_RECEIVER\_REPLACE\_PENDING);
44. mDateChangeSender = PendingIntent.getBroadcastAsUser(getContext(), 0, intent,
45. Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY\_BEFORE\_BOOT, UserHandle.ALL);
47. // 初始化驱动来调度alarm
48. // 注册时间/日期变化广播接收者
49. mClockReceiver = **new** ClockReceiver();
50. // 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
51. // 在ClockReceiver的onReceive方法中接收到时间变化的广博就会调用
52. // scheduleTimeTickEvent方法发送一分钟后的广播
53. mClockReceiver.scheduleTimeTickEvent();
54. // 调度日期变化事件，每天0点发送一次广播
55. mClockReceiver.scheduleDateChangedEvent();
56. // 初始化监听息屏/亮屏的广播接收者
57. mInteractiveStateReceiver = **new** InteractiveStateReceiver();
58. // 监听应用卸载和SD卡不可用的广播，判断是否移除alarm
59. mUninstallReceiver = **new** UninstallReceiver();
61. **if** (mNativeData != 0) {
62. // 初始化AlarmThread并运行它的run方法
63. AlarmThread waitThread = **new** AlarmThread();
64. waitThread.start();
65. } **else** {
66. Slog.w(TAG, "Failed to open alarm driver. Falling back to a handler.");
67. }
69. // 发布服务
70. publishBinderService(Context.ALARM\_SERVICE, mService);
71. }
73. **private** **native** **long** init();
75. // 把当前时区保存到内核
76. **void** setTimeZoneImpl(String tz) {
77. **if** (TextUtils.isEmpty(tz)) {
78. **return**;
79. }
81. // 获取要保存的时区
82. TimeZone zone = TimeZone.getTimeZone(tz);
83. // 写入时区时，避免同时写
84. **boolean** timeZoneWasChanged = **false**;
85. **synchronized** (**this**) {
86. // 再次获取要保存的时区
87. String current = SystemProperties.get(TIMEZONE\_PROPERTY);
88. // 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
89. **if** (current == **null** || !current.equals(zone.getID())) {
90. **if** (localLOGV) {
91. Slog.v(TAG, "timezone changed: " + current + ", new=" + zone.getID());
92. }
93. timeZoneWasChanged = **true**;
94. SystemProperties.set(TIMEZONE\_PROPERTY, zone.getID());
95. }
97. // Update the kernel timezone information
98. // Kernel tracks time offsets as 'minutes west of GMT'
99. // 更新内核时区信息
100. // 内核跟踪时间偏移为GMT以西的分钟数
101. **int** gmtOffset = zone.getOffset(System.currentTimeMillis());
102. // 保存时区到内核
103. setKernelTimezone(mNativeData, -(gmtOffset / 60000));
104. }
106. TimeZone.setDefault(**null**);
108. // 如果时区发生改变则发送时区改变的广播
109. **if** (timeZoneWasChanged) {
110. Intent intent = **new** Intent(Intent.ACTION\_TIMEZONE\_CHANGED);
111. intent.addFlags(Intent.FLAG\_RECEIVER\_REPLACE\_PENDING);
112. intent.putExtra("time-zone", zone.getID());
113. getContext().sendBroadcastAsUser(intent, UserHandle.ALL);
114. }
115. }
117. // 保存时区信息到内核中
118. **private** **native** **int** setKernelTimezone(**long** nativeData, **int** minuteswest);
120. // 时间/日期变化广播接收者
121. **class** ClockReceiver **extends** BroadcastReceiver {
122. **public** ClockReceiver() {
123. // 注册时间/日期变化的广播接收者
124. IntentFilter filter = **new** IntentFilter();
125. filter.addAction(Intent.ACTION\_TIME\_TICK);
126. filter.addAction(Intent.ACTION\_DATE\_CHANGED);
127. getContext().registerReceiver(**this**, filter);
128. }
130. @Override
131. **public** **void** onReceive(Context context, Intent intent) {
132. **if** (intent.getAction().equals(Intent.ACTION\_TIME\_TICK)) {
133. **if** (DEBUG\_BATCH) {
134. Slog.v(TAG, "Received TIME\_TICK alarm; rescheduling");
135. }
136. // 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
137. scheduleTimeTickEvent();
138. } **else** **if** (intent.getAction().equals(Intent.ACTION\_DATE\_CHANGED)) {
139. // 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
140. TimeZone zone = TimeZone.getTimeZone(SystemProperties.get(TIMEZONE\_PROPERTY));
141. **int** gmtOffset = zone.getOffset(System.currentTimeMillis());
142. // 日期变化时更新内核保存的时区信息
143. setKernelTimezone(mNativeData, -(gmtOffset / 60000));
144. // 调度日期变化事件
145. scheduleDateChangedEvent();
146. }
147. }
149. // 一分钟后给所有接收ACTION\_TIME\_TICK广播的接收者发送广播
150. **public** **void** scheduleTimeTickEvent() {
151. **final** **long** currentTime = System.currentTimeMillis();
152. **final** **long** nextTime = 60000 \* ((currentTime / 60000) + 1);
154. // 下一次执行事件的延迟时间
155. **final** **long** tickEventDelay = nextTime - currentTime;
157. **final** WorkSource workSource = **null**; // Let system take blame for time tick events.
158. // 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
159. setImpl(ELAPSED\_REALTIME, SystemClock.elapsedRealtime() + tickEventDelay, 0,
160. 0, mTimeTickSender, AlarmManager.FLAG\_STANDALONE, workSource, **null**,
161. Process.myUid());
162. }
164. // 调度日期变化事件
165. **public** **void** scheduleDateChangedEvent() {
166. Calendar calendar = Calendar.getInstance();
167. calendar.setTimeInMillis(System.currentTimeMillis());
168. calendar.set(Calendar.HOUR, 0);
169. calendar.set(Calendar.MINUTE, 0);
170. calendar.set(Calendar.SECOND, 0);
171. calendar.set(Calendar.MILLISECOND, 0);
172. calendar.add(Calendar.DAY\_OF\_MONTH, 1);
174. **final** WorkSource workSource = **null**; // Let system take blame for date change events.
175. // 设置mDateChangeSender
176. setImpl(RTC, calendar.getTimeInMillis(), 0, 0, mDateChangeSender,
177. AlarmManager.FLAG\_STANDALONE, workSource, **null**, Process.myUid());
178. }
179. }
181. // 修正参数并设置传递过来的PendingIntent
182. **void** setImpl(**int** type, **long** triggerAtTime, **long** windowLength, **long** interval,
183. PendingIntent operation, **int** flags, WorkSource workSource,
184. AlarmManager.AlarmClockInfo alarmClock, **int** callingUid) {
185. **if** (operation == **null**) {
186. Slog.w(TAG, "set/setRepeating ignored because there is no intent");
187. **return**;
188. }
190. /\*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\*/
192. // 检查时间窗长度，如果超过半天就转换成一小时。
193. **if** (windowLength > AlarmManager.INTERVAL\_HALF\_DAY) {
194. Slog.w(TAG, "Window length " + windowLength
195. + "ms suspiciously long; limiting to 1 hour");
196. windowLength = AlarmManager.INTERVAL\_HOUR;
197. }
199. // 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。
200. // 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
201. **final** **long** minInterval = mConstants.MIN\_INTERVAL;// 1分钟
202. **if** (interval > 0 && interval < minInterval) {
203. Slog.w(TAG, "Suspiciously short interval " + interval
204. + " millis; expanding to " + (minInterval/1000)
205. + " seconds");
206. interval = minInterval;
207. }
209. // 检查type的合法性
210. **if** (type < RTC\_WAKEUP || type > RTC\_POWEROFF\_WAKEUP) {
211. **throw** **new** IllegalArgumentException("Invalid alarm type " + type);
212. }
214. // 检查triggerAtTime的合法性
215. **if** (triggerAtTime < 0) {
216. **final** **long** what = Binder.getCallingPid();
217. Slog.w(TAG, "Invalid alarm trigger time! " + triggerAtTime + " from uid=" + callingUid
218. + " pid=" + what);
219. triggerAtTime = 0;
220. }
222. **final** **long** nowElapsed = SystemClock.elapsedRealtime();
223. // 把RTC格式的转换成elapsedRealtime格式的触发时间
224. **final** **long** nominalTrigger = convertToElapsed(triggerAtTime, type);
225. // 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
226. **final** **long** minTrigger = nowElapsed + mConstants.MIN\_FUTURITY;// 5秒
227. **final** **long** triggerElapsed = (nominalTrigger > minTrigger) ? nominalTrigger : minTrigger;
229. **final** **long** maxElapsed;
230. **if** (windowLength == AlarmManager.WINDOW\_EXACT) {
231. // 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
232. maxElapsed = triggerElapsed;
233. } **else** **if** (windowLength < 0) {
234. // 计算alarm的最晚触发时间
235. maxElapsed = maxTriggerTime(nowElapsed, triggerElapsed, interval);
236. // 根据最晚触发时间修正alarm的时间窗时间
237. windowLength = maxElapsed - triggerElapsed;
238. } **else** {
239. // 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
240. maxElapsed = triggerElapsed + windowLength;
241. }
243. **synchronized** (mLock) {
244. **if** (DEBUG\_BATCH) {
245. Slog.v(TAG, "set(" + operation + ") : type=" + type
246. + " triggerAtTime=" + triggerAtTime + " win=" + windowLength
247. + " tElapsed=" + triggerElapsed + " maxElapsed=" + maxElapsed
248. + " interval=" + interval + " flags=0x" + Integer.toHexString(flags));
249. }
250. // 根据修正后的各参数进一步设置alarm
251. setImplLocked(type, triggerAtTime, triggerElapsed, windowLength, maxElapsed,
252. interval, operation, flags, **true**, workSource, alarmClock, callingUid);
253. }
254. }
256. // 如果alarm的执行时间设置的是RTC则需要转换成开机时间
257. **static** **long** convertToElapsed(**long** when, **int** type) {
258. **final** **boolean** isRtc = (type == RTC || type == RTC\_WAKEUP
259. || type == RTC\_POWEROFF\_WAKEUP);
260. **if** (isRtc) {
261. when -= System.currentTimeMillis() - SystemClock.elapsedRealtime();
262. }
263. **return** when;
264. }
266. // 最小重复周期
267. **static** **final** **long** MIN\_FUZZABLE\_INTERVAL = 10000;
269. // 最晚触发时间
270. **static** **long** maxTriggerTime(**long** now, **long** triggerAtTime, **long** interval) {
271. **long** futurity = (interval == 0)
272. ? (triggerAtTime - now)
273. : interval;
274. // 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
275. **if** (futurity < MIN\_FUZZABLE\_INTERVAL) {
276. futurity = 0;
277. }
278. // 把0.75倍的下次触发时间+触发时间赋值给alarm的maxElapsed
279. **return** triggerAtTime + (**long**)(.75 \* futurity);
280. }
282. // 根据传递来的各alarm相关参数进一步设置alarm
283. **private** **void** setImplLocked(**int** type, **long** when, **long** whenElapsed, **long** windowLength,
284. **long** maxWhen, **long** interval, PendingIntent operation, **int** flags,
285. **boolean** doValidate, WorkSource workSource, AlarmManager.AlarmClockInfo alarmClock,
286. **int** uid) {
287. // 根据传递过来的参数重新创建一个alarm实例
288. Alarm a = **new** Alarm(type, when, whenElapsed, windowLength, maxWhen, interval,
289. operation, workSource, flags, alarmClock, uid);
290. // 根据operation参数从所在列表中移除该alarm
291. removeLocked(operation);
292. // 把新创建的alarm添加到相关列表
293. setImplLocked(a, **false**, doValidate);
294. }
296. // alarm批处理对象列表
297. **final** ArrayList<Batch> mAlarmBatches = **new** ArrayList<>();
298. // 延迟到空闲状态时执行的alarm列表
299. ArrayList<Alarm> mPendingWhileIdleAlarms = **new** ArrayList<>();
300. // 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
301. Alarm mPendingIdleUntil = **null**;
302. // 下一个请求退出空闲模式的alarm
303. Alarm mNextWakeFromIdle = **null**;
305. // 根据operation从各列表中查找alarm并移除
306. **private** **void** removeLocked(PendingIntent operation) {
307. **boolean** didRemove = **false**;
308. // 遍历mAlarmBatches查找目标alarm
309. **for** (**int** i = mAlarmBatches.size() - 1; i >= 0; i--) {
310. Batch b = mAlarmBatches.get(i);
311. ArrayList<Alarm> alarmList = b.alarms;
312. Alarm alarm = **null**;
313. **for** (**int** j = alarmList.size() - 1; j >= 0; j--) {
314. alarm = alarmList.get(j);
315. **if** (alarm.type == RTC\_POWEROFF\_WAKEUP && alarm.operation.equals(operation)) {
316. **long** alarmSeconds, alarmNanoseconds;
317. alarmSeconds = alarm.when / 1000;
318. alarmNanoseconds = (alarm.when % 1000) \* 1000 \* 1000;
319. // 遍历所有alarm，找到和目标operation相同的alarm，
320. // 如果该alarm的类型是Rtc wakeup的则清空在驱动中的相关设置
321. clear(mNativeData, alarm.type, alarmSeconds, alarmNanoseconds);
322. mNextRtcWakeup = 0;
323. }
324. }
325. // 把和目标operation相同的alarm从批处理中移除掉
326. didRemove |= b.remove(operation);
327. // 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
328. **if** (b.size() == 0) {
329. mAlarmBatches.remove(i);
330. }
331. }
332. // 遍历mPendingWhileIdleAlarms查找目标alarm并移除
333. **for** (**int** i = mPendingWhileIdleAlarms.size() - 1; i >= 0; i--) {
334. **if** (mPendingWhileIdleAlarms.get(i).operation.equals(operation)) {
335. // 从mPendingWhileIdleAlarms列表中移除alarm不用设置didRemove字段，
336. // 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。
337. mPendingWhileIdleAlarms.remove(i);
338. }
339. }
341. // 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
342. **if** (didRemove) {
343. **if** (DEBUG\_BATCH) {
344. Slog.v(TAG, "remove(operation) changed bounds; rebatching");
345. }
346. **boolean** restorePending = **false**;
347. // 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
348. **if** (mPendingIdleUntil != **null** && mPendingIdleUntil.operation.equals(operation)) {
349. mPendingIdleUntil = **null**;
350. restorePending = **true**;
351. }
352. // 查找目标alarm是否为mNextWakeFromIdle，如果是则赋null
353. **if** (mNextWakeFromIdle != **null** && mNextWakeFromIdle.operation.equals(operation)) {
354. mNextWakeFromIdle = **null**;
355. }
356. // 对所有的alarm进行重新批处理操作
357. rebatchAllAlarmsLocked(**true**);
358. **if** (restorePending) {
359. // 由于mPendingIdleUntil变量改变，所以需要重新存储空闲时延迟执行的alarm
360. restorePendingWhileIdleAlarmsLocked();
361. }
362. // 更新下一个alarm的alarmClock
363. updateNextAlarmClockLocked();
364. }
365. }
367. // 根据alarm的类型、触发时间等参数从驱动中移除掉该alarm
368. **private** **native** **void** clear(**long** nativeData, **int** type, **long** seconds, **long** nanoseconds);
370. // 对所有的alarm进行重新批处理操作
371. **void** rebatchAllAlarmsLocked(**boolean** doValidate) {
372. ArrayList<Batch> oldSet = (ArrayList<Batch>) mAlarmBatches.clone();
373. mAlarmBatches.clear();
374. Alarm oldPendingIdleUntil = mPendingIdleUntil;
375. **final** **long** nowElapsed = SystemClock.elapsedRealtime();
376. **final** **int** oldBatches = oldSet.size();
377. **for** (**int** batchNum = 0; batchNum < oldBatches; batchNum++) {
378. Batch batch = oldSet.get(batchNum);
379. **final** **int** N = batch.size();
380. **for** (**int** i = 0; i < N; i++) {
381. // 重新添加alarm到mAlarmBatches列表中
382. reAddAlarmLocked(batch.get(i), nowElapsed, doValidate);
383. }
384. }
385. **if** (oldPendingIdleUntil != **null** && oldPendingIdleUntil != mPendingIdleUntil) {
386. Slog.wtf(TAG, "Rebatching: idle until changed from " + oldPendingIdleUntil
387. + " to " + mPendingIdleUntil);
388. **if** (mPendingIdleUntil == **null**) {
389. // mPendingIdleUntil变化导致需要重新存储空闲时延迟执行的alarm
390. restorePendingWhileIdleAlarmsLocked();
391. }
392. }
393. // 重新设置几个核心alarm
394. rescheduleKernelAlarmsLocked();
395. // 更新下一个alarm的alarmClock
396. updateNextAlarmClockLocked();
397. }
399. // 重新添加alarm到mAlarmBatches列表中
400. **void** reAddAlarmLocked(Alarm a, **long** nowElapsed, **boolean** doValidate) {
401. a.when = a.origWhen;
402. **long** whenElapsed = convertToElapsed(a.when, a.type);
403. **final** **long** maxElapsed;
404. // windowLength：alarm执行的时间窗时间
405. **if** (a.windowLength == AlarmManager.WINDOW\_EXACT) {
406. // 精确执行时间
407. maxElapsed = whenElapsed;
408. } **else** {
409. // 非精确执行时间，即允许在时间窗误差内触发
410. maxElapsed = (a.windowLength > 0)
411. ? (whenElapsed + a.windowLength)
412. : maxTriggerTime(nowElapsed, whenElapsed, a.repeatInterval);
413. }
414. a.whenElapsed = whenElapsed;
415. a.maxWhenElapsed = maxElapsed;
416. // 重新添加alarm到各相关列表中
417. setImplLocked(a, **true**, doValidate);
418. }
420. Random mRandom;
421. // 下一个将要执行的alarm的开始执行时间是否可能改变
422. **private** **boolean** mNextAlarmClockMayChange;
424. // 把alarm添加各相关列表中
425. **private** **void** setImplLocked(Alarm a, **boolean** rebatching, **boolean** doValidate) {
426. // 判断alarm是否设置了FLAG\_IDLE\_UNTIL
427. **if** ((a.flags&AlarmManager.FLAG\_IDLE\_UNTIL) != 0) {
428. // 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。
429. // 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
430. **if** (mNextWakeFromIdle != **null** && a.whenElapsed > mNextWakeFromIdle.whenElapsed) {
431. // 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
432. a.when = a.whenElapsed = a.maxWhenElapsed = mNextWakeFromIdle.whenElapsed;
433. }
434. // 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。
435. // 这里增加模糊时间，让alarm比实际开始执行时间早一点执行。
436. **final** **long** nowElapsed = SystemClock.elapsedRealtime();
437. // 计算模糊时间
438. **final** **int** fuzz = fuzzForDuration(a.whenElapsed-nowElapsed);
439. **if** (fuzz > 0) {
440. **if** (mRandom == **null**) {
441. mRandom = **new** Random();
442. }
443. // 根据模糊时间取一个随机数
444. **final** **int** delta = mRandom.nextInt(fuzz);
445. // alarm的开始执行时间减去随机数，让alarm早执行一点
446. a.whenElapsed -= delta;
447. **if** (**false**) {
448. Slog.d(TAG, "Alarm when: " + a.whenElapsed);
449. Slog.d(TAG, "Delta until alarm: " + (a.whenElapsed-nowElapsed));
450. Slog.d(TAG, "Applied fuzz: " + fuzz);
451. Slog.d(TAG, "Final delta: " + delta);
452. Slog.d(TAG, "Final when: " + a.whenElapsed);
453. }
454. a.when = a.maxWhenElapsed = a.whenElapsed;
455. }
457. } **else** **if** (mPendingIdleUntil != **null**) {
458. // 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
459. // 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
460. **if** ((a.flags&(AlarmManager.FLAG\_ALLOW\_WHILE\_IDLE
461. | AlarmManager.FLAG\_ALLOW\_WHILE\_IDLE\_UNRESTRICTED
462. | AlarmManager.FLAG\_WAKE\_FROM\_IDLE))
463. == 0) {
464. mPendingWhileIdleAlarms.add(a);
465. **return**;
466. }
467. }
469. // 获取批处理对象在mAlarmBatches列表中的索引值
470. **int** whichBatch = ((a.flags&AlarmManager.FLAG\_STANDALONE) != 0)
471. ? -1 : attemptCoalesceLocked(a.whenElapsed, a.maxWhenElapsed);
472. **if** (whichBatch < 0) {
473. // 如果alarm是独立的，则它不能和其他alarm一起进行批处理操作
474. Batch batch = **new** Batch(a);
475. // 把批处理对象添加到mAlarmBatches列表中
476. addBatchLocked(mAlarmBatches, batch);
477. } **else** {
478. Batch batch = mAlarmBatches.get(whichBatch);
479. // 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
480. **if** (batch.add(a)) {
481. // 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
482. mAlarmBatches.remove(whichBatch);
483. // 重新把批处理对象添加到mAlarmBatches列表中的正确位置上
484. addBatchLocked(mAlarmBatches, batch);
485. }
486. }
488. **if** (a.alarmClock != **null**) {
489. mNextAlarmClockMayChange = **true**;
490. }
492. **boolean** needRebatch = **false**;
494. // 判断alarm是否设置了FLAG\_IDLE\_UNTIL
495. **if** ((a.flags&AlarmManager.FLAG\_IDLE\_UNTIL) != 0) {
496. // 由于该alarm设置了FLAG\_ALLOW\_WHILE\_IDLE,所以把该alarm赋值给mPendingIdleUntil变量
497. mPendingIdleUntil = a;
498. // 更新可以开始执行flag为ALLOW\_WHILE\_IDLE的alarm的最小时间间隔
499. mConstants.updateAllowWhileIdleMinTimeLocked();
500. // 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
501. needRebatch = **true**;
502. // 判断alarm是否设置了FLAG\_WAKE\_FROM\_IDLE
503. } **else** **if** ((a.flags&AlarmManager.FLAG\_WAKE\_FROM\_IDLE) != 0) {
504. // 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
505. **if** (mNextWakeFromIdle == **null** || mNextWakeFromIdle.whenElapsed > a.whenElapsed) {
506. // 则把a赋值给mNextWakeFromIdle
507. mNextWakeFromIdle = a;
508. // 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
509. **if** (mPendingIdleUntil != **null**) {
510. needRebatch = **true**;
511. }
512. }
513. }
515. **if** (!rebatching) {
516. **if** (needRebatch) {
517. // 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
518. rebatchAllAlarmsLocked(**false**);
519. }
521. // 重新设置几个核心alarm
522. rescheduleKernelAlarmsLocked();
523. // 更新下一个alarm的alarmClock
524. updateNextAlarmClockLocked();
525. }
526. }
528. // 计算模糊时间
529. **static** **int** fuzzForDuration(**long** duration) {
530. **if** (duration < 15\*60\*1000) {
531. // 如果距离开始执行alarm的时间duration小于15分钟，则fuzz就等于duration
532. **return** (**int**)duration;
533. } **else** **if** (duration < 90\*60\*1000) {
534. // 如果duration小于1.5小时，则令fuzz等于15分钟
535. **return** 15\*60\*1000;
536. } **else** {
537. // 否则，令fuzz等于30分钟
538. **return** 30\*60\*1000;
539. }
540. }
542. // 返回匹配的批处理对象的索引，如果不存在则返回-1
543. **int** attemptCoalesceLocked(**long** whenElapsed, **long** maxWhen) {
544. **final** **int** N = mAlarmBatches.size();
545. **for** (**int** i = 0; i < N; i++) {
546. Batch b = mAlarmBatches.get(i);
547. **if** ((b.flags&AlarmManager.FLAG\_STANDALONE) == 0 && b.canHold(whenElapsed, maxWhen)) {
548. **return** i;
549. }
550. }
551. **return** -1;
552. }
554. // 根据批处理对象的start时间从小到大排序
555. **static** **final** BatchTimeOrder sBatchOrder = **new** BatchTimeOrder();
557. // 如果批处理对象添加到列表的第一位则返回true
558. **static** **boolean** addBatchLocked(ArrayList<Batch> list, Batch newBatch) {
559. // 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
560. **int** index = Collections.binarySearch(list, newBatch, sBatchOrder);
561. **if** (index < 0) {
562. index = 0 - index - 1;
563. }
564. list.add(index, newBatch);
565. **return** (index == 0);
566. }
568. **static** **class** BatchTimeOrder **implements** Comparator<Batch> {
569. **public** **int** compare(Batch b1, Batch b2) {
570. **long** when1 = b1.start;
571. **long** when2 = b2.start;
572. **if** (when1 - when2 > 0) {
573. **return** 1;
574. }
575. **if** (when1 - when2 < 0) {
576. **return** -1;
577. }
578. **return** 0;
579. }
580. }
582. // 延迟执行的非wakeup的alarm列表
583. ArrayList<Alarm> mPendingNonWakeupAlarms = **new** ArrayList<>();
584. // 下一次非wakeup类型的alarm传递的时间
585. **long** mNextNonWakeupDeliveryTime;
586. // 重新设置几个核心alarm（几个单独命名的特定alarm）
587. **void** rescheduleKernelAlarmsLocked() {
588. // Schedule the next upcoming wakeup alarm.  If there is a deliverable batch
589. // prior to that which contains no wakeups, we schedule that as well.
590. // 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。
591. **long** nextNonWakeup = 0;
592. **if** (mAlarmBatches.size() > 0) {
593. // 获取第一个包含wakeup的批处理对象
594. **final** Batch firstWakeup = findFirstWakeupBatchLocked();
595. // 获取mAlarmBatches列表中的第一个批处理对象
596. **final** Batch firstBatch = mAlarmBatches.get(0);
597. // 获取第一个包含Rtc wakeup的批处理对象
598. **final** Batch firstRtcWakeup = findFirstRtcWakeupBatchLocked();
599. **if** (firstWakeup != **null** && mNextWakeup != firstWakeup.start) {
600. // 更新mNextWakeup
601. mNextWakeup = firstWakeup.start;
602. // 把alarm设置到驱动中
603. setLocked(ELAPSED\_REALTIME\_WAKEUP, firstWakeup.start);
604. }
605. **if** (firstRtcWakeup != **null** && mNextRtcWakeup != firstRtcWakeup.start) {
606. // 更新mNextRtcWakeup
607. mNextRtcWakeup = firstRtcWakeup.start;
608. // 根据alarm.whenElapsed获取alarm.when
609. **long** when = firstRtcWakeup.getWhenByElapsedTime(mNextRtcWakeup);
610. **if** (when != 0) {
611. setLocked(RTC\_POWEROFF\_WAKEUP, when);
612. }
613. }
614. **if** (firstBatch != firstWakeup) {
615. // 把mAlarmBatches列表中的第一个批处理对象的start赋值给nextNonWakeup
616. nextNonWakeup = firstBatch.start;
617. }
618. }
619. **if** (mPendingNonWakeupAlarms.size() > 0) {
620. **if** (nextNonWakeup == 0 || mNextNonWakeupDeliveryTime < nextNonWakeup) {
621. // 更新nextNonWakeup
622. nextNonWakeup = mNextNonWakeupDeliveryTime;
623. }
624. }
625. **if** (nextNonWakeup != 0 && mNextNonWakeup != nextNonWakeup) {
626. // 更新mNextNonWakeup
627. mNextNonWakeup = nextNonWakeup;
628. setLocked(ELAPSED\_REALTIME, nextNonWakeup);
629. }
630. }
632. // 查找第一个包含wakeup的批处理对象
633. **private** Batch findFirstWakeupBatchLocked() {
634. **final** **int** N = mAlarmBatches.size();
635. **for** (**int** i = 0; i < N; i++) {
636. Batch b = mAlarmBatches.get(i);
637. **if** (b.hasWakeups()) {
638. **return** b;
639. }
640. }
641. **return** **null**;
642. }
644. // 查找第一个包含Rtc wakeup的批处理对象
645. **private** Batch findFirstRtcWakeupBatchLocked() {
646. **final** **int** N = mAlarmBatches.size();
647. **for** (**int** i = 0; i < N; i++) {
648. Batch b = mAlarmBatches.get(i);
649. **long** intervalTime  = b.start - SystemClock.elapsedRealtime();
650. **if** (b.isRtcPowerOffWakeup()) {
651. **return** b;
652. }
653. }
654. **return** **null**;
655. }
657. // 把alarm设置到驱动中
658. **private** **void** setLocked(**int** type, **long** when) {
659. **if** (mNativeData != 0) {
660. // The kernel never triggers alarms with negative wakeup times
661. // so we ensure they are positive.
662. // 秒、纳秒
663. **long** alarmSeconds, alarmNanoseconds;
664. **if** (when < 0) {
665. alarmSeconds = 0;
666. alarmNanoseconds = 0;
667. } **else** {
668. alarmSeconds = when / 1000;
669. alarmNanoseconds = (when % 1000) \* 1000 \* 1000;
670. }
672. // 把alarm设置到驱动中
673. set(mNativeData, type, alarmSeconds, alarmNanoseconds);
674. } **else** {
675. Message msg = Message.obtain();
676. msg.what = ALARM\_EVENT;
678. mHandler.removeMessages(ALARM\_EVENT);
679. mHandler.sendMessageAtTime(msg, when);
680. }
681. }
683. // 设置alarm只跟类型、触发时间相关，跟alarm的其他参数无关
684. **private** **native** **void** set(**long** nativeData, **int** type, **long** seconds, **long** nanoseconds);
686. **private** **final** SparseArray<AlarmManager.AlarmClockInfo> mTmpSparseAlarmClockArray =
687. **new** SparseArray<>();
688. **private** **final** SparseArray<AlarmManager.AlarmClockInfo> mNextAlarmClockForUser =
689. **new** SparseArray<>();
691. /\*\*
692. \* 更新下一个alarm的alarmClock
693. \*/
694. **private** **void** updateNextAlarmClockLocked() {
695. // 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
696. **if** (!mNextAlarmClockMayChange) {
697. **return**;
698. }
699. mNextAlarmClockMayChange = **false**;
701. SparseArray<AlarmManager.AlarmClockInfo> nextForUser = mTmpSparseAlarmClockArray;
702. nextForUser.clear();
704. **final** **int** N = mAlarmBatches.size();
705. **for** (**int** i = 0; i < N; i++) {
706. ArrayList<Alarm> alarms = mAlarmBatches.get(i).alarms;
707. **final** **int** M = alarms.size();
709. **for** (**int** j = 0; j < M; j++) {
710. Alarm a = alarms.get(j);
711. **if** (a.alarmClock != **null**) {
712. **final** **int** userId = UserHandle.getUserId(a.uid);
714. **if** (DEBUG\_ALARM\_CLOCK) {
715. Log.v(TAG, "Found AlarmClockInfo at " +
716. formatNextAlarm(getContext(), a.alarmClock, userId) +
717. " for user " + userId);
718. }
720. // Alarms and batches are sorted by time, no need to compare times here.
721. **if** (nextForUser.get(userId) == **null**) {
722. // 把userId和alarmClock以键值对的形式添加到nextForUser中
723. nextForUser.put(userId, a.alarmClock);
724. }
725. }
726. }
727. }
729. // Update mNextAlarmForUser with new values.
730. **final** **int** NN = nextForUser.size();
731. **for** (**int** i = 0; i < NN; i++) {
732. AlarmManager.AlarmClockInfo newAlarm = nextForUser.valueAt(i);
733. **int** userId = nextForUser.keyAt(i);
734. AlarmManager.AlarmClockInfo currentAlarm = mNextAlarmClockForUser.get(userId);
735. **if** (!newAlarm.equals(currentAlarm)) {
736. // 更新mNextAlarmClockForUser
737. updateNextAlarmInfoForUserLocked(userId, newAlarm);
738. }
739. }
741. // Remove users without any alarm clocks scheduled.
742. **final** **int** NNN = mNextAlarmClockForUser.size();
743. **for** (**int** i = NNN - 1; i >= 0; i--) {
744. **int** userId = mNextAlarmClockForUser.keyAt(i);
745. **if** (nextForUser.get(userId) == **null**) {
746. // 更新mNextAlarmClockForUser
747. updateNextAlarmInfoForUserLocked(userId, **null**);
748. }
749. }
750. }
752. **private** **final** SparseBooleanArray mPendingSendNextAlarmClockChangedForUser =
753. **new** SparseBooleanArray();
755. // 更新mNextAlarmClockForUser
756. **private** **void** updateNextAlarmInfoForUserLocked(**int** userId,
757. AlarmManager.AlarmClockInfo alarmClock) {
758. **if** (alarmClock != **null**) {
759. **if** (DEBUG\_ALARM\_CLOCK) {
760. Log.v(TAG, "Next AlarmClockInfoForUser(" + userId + "): " +
761. formatNextAlarm(getContext(), alarmClock, userId));
762. }
763. mNextAlarmClockForUser.put(userId, alarmClock);
764. } **else** {
765. **if** (DEBUG\_ALARM\_CLOCK) {
766. Log.v(TAG, "Next AlarmClockInfoForUser(" + userId + "): None");
767. }
768. mNextAlarmClockForUser.remove(userId);
769. }
771. mPendingSendNextAlarmClockChangedForUser.put(userId, **true**);
772. mHandler.removeMessages(AlarmHandler.SEND\_NEXT\_ALARM\_CLOCK\_CHANGED);
773. mHandler.sendEmptyMessage(AlarmHandler.SEND\_NEXT\_ALARM\_CLOCK\_CHANGED);
774. }
776. // 重新存储延迟到空闲时执行的alarm
777. **void** restorePendingWhileIdleAlarmsLocked() {
778. **if** (mPendingWhileIdleAlarms.size() > 0) {
779. ArrayList<Alarm> alarms = mPendingWhileIdleAlarms;
780. mPendingWhileIdleAlarms = **new** ArrayList<>();
781. **final** **long** nowElapsed = SystemClock.elapsedRealtime();
782. **for** (**int** i=alarms.size() - 1; i >= 0; i--) {
783. Alarm a = alarms.get(i);
784. // 重新添加alarm到mAlarmBatches列表中
785. reAddAlarmLocked(a, nowElapsed, **false**);
786. }
787. }
789. // 更新可以开始执行flag为ALLOW\_WHILE\_IDLE的alarm的最小时间间隔
790. mConstants.updateAllowWhileIdleMinTimeLocked();
792. // Reschedule everything.
793. // 重新设置几个核心alarm
794. rescheduleKernelAlarmsLocked();
795. // 更新下一个alarm的alarmClock
796. updateNextAlarmClockLocked();
798. // And send a TIME\_TICK right now, since it is important to get the UI updated.
799. **try** {
800. // 执行和PendingIntent关联的操作
801. mTimeTickSender.send();
802. } **catch** (PendingIntent.CanceledException e) {
803. }
804. }
806. // 监听息屏/亮屏的广播接收者
807. **class** InteractiveStateReceiver **extends** BroadcastReceiver {
808. **public** InteractiveStateReceiver() {
809. IntentFilter filter = **new** IntentFilter();
810. filter.addAction(Intent.ACTION\_SCREEN\_OFF);
811. filter.addAction(Intent.ACTION\_SCREEN\_ON);
812. filter.setPriority(IntentFilter.SYSTEM\_HIGH\_PRIORITY);
813. getContext().registerReceiver(**this**, filter);
814. }
816. @Override
817. **public** **void** onReceive(Context context, Intent intent) {
818. **synchronized** (mLock) {
819. // 息屏/亮屏状态发生变化
820. interactiveStateChangedLocked(Intent.ACTION\_SCREEN\_ON.equals(intent.getAction()));
821. }
822. }
823. }
825. // 记录上次亮屏状态，亮屏为true
826. **boolean** mInteractive = **true**;
827. // 延迟执行的非wakeup的alarm的开始延迟的时间点
828. **long** mStartCurrentDelayTime;
829. // 延迟执行的非wakeup的alarm的总延迟时间
830. **long** mTotalDelayTime = 0;
831. // 延迟执行的非wakeup的alarm的最大一次延迟时间
832. **long** mMaxDelayTime = 0;
833. // 最长的息屏时间，后面打log用
834. **long** mNonInteractiveTime;
835. // 开始息屏的时间
836. **long** mNonInteractiveStartTime;
838. // 息屏/亮屏状态发生变化
839. **void** interactiveStateChangedLocked(**boolean** interactive) {
840. **if** (mInteractive != interactive) {
841. mInteractive = interactive;
842. **final** **long** nowELAPSED = SystemClock.elapsedRealtime();
843. // 如果当前是亮屏状态
844. **if** (interactive) {
845. // 判断延迟执行的非wakeup的alarm列表是否大于0
846. **if** (mPendingNonWakeupAlarms.size() > 0) {
847. // 计算本次延迟时间
848. **final** **long** thisDelayTime = nowELAPSED - mStartCurrentDelayTime;
849. // 累加到总延迟时间中
850. mTotalDelayTime += thisDelayTime;
851. // 更新最大一次延迟时间
852. **if** (mMaxDelayTime < thisDelayTime) {
853. mMaxDelayTime = thisDelayTime;
854. }
855. // 执行mPendingNonWakeupAlarms列表中的alarm
856. deliverAlarmsLocked(mPendingNonWakeupAlarms, nowELAPSED);
857. // 清空列表
858. mPendingNonWakeupAlarms.clear();
859. }
860. // 保存最长的息屏时间
861. **if** (mNonInteractiveStartTime > 0) {
862. **long** dur = nowELAPSED - mNonInteractiveStartTime;
863. **if** (dur > mNonInteractiveTime) {
864. mNonInteractiveTime = dur;
865. }
866. }
867. } **else** {
868. // 如果当前是息屏状态则开始息屏的时间
869. mNonInteractiveStartTime = nowELAPSED;
870. }
871. }
872. }
874. // 上次传递alarm的时间
875. **long** mLastAlarmDeliveryTime;
876. **static** **final** **boolean** RECORD\_ALARMS\_IN\_HISTORY = **true**;
877. **static** **final** Intent mBackgroundIntent = **new** Intent()
878. .addFlags(Intent.FLAG\_FROM\_BACKGROUND);
879. **int** mBroadcastRefCount = 0;
880. // 执行中的alarm的信息列表
881. ArrayList<InFlight> mInFlight = **new** ArrayList<>();
882. // 触发执行的alarm所在应用程序的uid列表
883. **private** **final** ArrayList<Integer> mTriggeredUids = **new** ArrayList<Integer>();
884. **final** ResultReceiver mResultReceiver = **new** ResultReceiver();
886. // 执行triggerList中的alarm
887. **void** deliverAlarmsLocked(ArrayList<Alarm> triggerList, **long** nowELAPSED) {
888. mLastAlarmDeliveryTime = nowELAPSED;
889. **for** (**int** i=0; i<triggerList.size(); i++) {
890. Alarm alarm = triggerList.get(i);
891. **final** **boolean** allowWhileIdle = (alarm.flags&AlarmManager.FLAG\_ALLOW\_WHILE\_IDLE) != 0;
892. **try** {
893. **if** (localLOGV) {
894. Slog.v(TAG, "sending alarm " + alarm);
895. }
896. **if** (RECORD\_ALARMS\_IN\_HISTORY) {
897. **if** (alarm.workSource != **null** && alarm.workSource.size() > 0) {
898. **for** (**int** wi=0; wi<alarm.workSource.size(); wi++) {
899. // 通过Binder对象调用AMS的noteAlarmStart方法
900. ActivityManagerNative.noteAlarmStart(
901. alarm.operation, alarm.workSource.get(wi), alarm.tag);
902. }
903. } **else** {
904. ActivityManagerNative.noteAlarmStart(
905. alarm.operation, -1, alarm.tag);
906. }
907. }
908. // 执行和PendingIntent关联的操作
909. alarm.operation.send(getContext(), 0,
910. mBackgroundIntent.putExtra(
911. Intent.EXTRA\_ALARM\_COUNT, alarm.count),
912. mResultReceiver, mHandler, **null**, allowWhileIdle ? mIdleOptions : **null**);
914. // we have an active broadcast so stay awake.
915. // 获取wakelock
916. **if** (mBroadcastRefCount == 0 || !mWakeLock.isHeld()) {
917. setWakelockWorkSource(alarm.operation, alarm.workSource,
918. alarm.type, alarm.tag, **true**);
919. mWakeLock.acquire();
920. }
921. **final** InFlight inflight = **new** InFlight(AlarmManagerService.**this**,
922. alarm.operation,
923. alarm.workSource,
924. alarm.type, alarm.tag,
925. nowELAPSED, alarm.uid);
926. mInFlight.add(inflight);
927. mBroadcastRefCount++;
928. mTriggeredUids.add(**new** Integer(alarm.uid));
929. // 释放wakelock
930. **if** (checkReleaseWakeLock()) {
931. **if** (mWakeLock.isHeld()) {
932. mWakeLock.release();
933. **if** (localLOGV) Slog.v(TAG, "AM WakeLock Released Internally deliverAlarms");
934. }
935. }
937. **if** (allowWhileIdle) {
938. // Record the last time this uid handled an ALLOW\_WHILE\_IDLE alarm.
939. mLastAllowWhileIdleDispatch.put(alarm.uid, nowELAPSED);
940. }
942. **final** BroadcastStats bs = inflight.mBroadcastStats;
943. bs.count++;
944. **if** (bs.nesting == 0) {
945. bs.nesting = 1;
946. bs.startTime = nowELAPSED;
947. } **else** {
948. bs.nesting++;
949. }
950. **final** FilterStats fs = inflight.mFilterStats;
951. fs.count++;
952. **if** (fs.nesting == 0) {
953. fs.nesting = 1;
954. fs.startTime = nowELAPSED;
955. } **else** {
956. fs.nesting++;
957. }
958. // 判断是否为WAKEUP类型的alarm
959. **if** (alarm.type == ELAPSED\_REALTIME\_WAKEUP
960. || alarm.type == RTC\_WAKEUP
961. || alarm.type == RTC\_POWEROFF\_WAKEUP) {
962. bs.numWakeup++;
963. fs.numWakeup++;
964. **if** (alarm.workSource != **null** && alarm.workSource.size() > 0) {
965. **for** (**int** wi=0; wi<alarm.workSource.size(); wi++) {
966. // 通过Binder对象调用AMS中的noteWakeupAlarm方法
967. ActivityManagerNative.noteWakeupAlarm(
968. alarm.operation, alarm.workSource.get(wi),
969. alarm.workSource.getName(wi), alarm.tag);
970. }
971. } **else** {
972. ActivityManagerNative.noteWakeupAlarm(
973. alarm.operation, -1, **null**, alarm.tag);
974. }
975. }
976. } **catch** (PendingIntent.CanceledException e) {
977. **if** (alarm.repeatInterval > 0) {
978. // This IntentSender is no longer valid, but this
979. // is a repeating alarm, so toss the hoser.
980. removeImpl(alarm.operation);
981. }
982. } **catch** (RuntimeException e) {
983. Slog.w(TAG, "Failure sending alarm.", e);
984. }
985. }
986. }
988. // 等待执行alarm所在应用程序的uid列表
989. **private** **final** ArrayList<Integer> mBlockedUids = **new** ArrayList<Integer>();
991. // 判断是否可以释放wakelock
992. **boolean** checkReleaseWakeLock() {
993. **if** (mTriggeredUids.size() == 0 || mBlockedUids.size() == 0)
994. **return** **false**;
996. **int** uid;
997. **for** (**int** i = 0; i <  mTriggeredUids.size(); i++) {
998. uid = mTriggeredUids.get(i);
999. // 只要等待执行列表中有未处理的alarm就返回false
1000. **if** (!mBlockedUids.contains(uid)) {
1001. **return** **false**;
1002. }
1003. }
1004. **return** **true**;
1005. }
1007. // 监听应用卸载和SD卡不可用的广播，判断是否移除alarm
1008. **class** UninstallReceiver **extends** BroadcastReceiver {
1009. **public** UninstallReceiver() {
1010. IntentFilter filter = **new** IntentFilter();
1011. filter.addAction(Intent.ACTION\_PACKAGE\_REMOVED);
1012. filter.addAction(Intent.ACTION\_PACKAGE\_RESTARTED);
1013. filter.addAction(Intent.ACTION\_QUERY\_PACKAGE\_RESTART);
1014. filter.addDataScheme("package");
1015. getContext().registerReceiverAsUser(**this**, UserHandle.ALL, filter, **null**, **null**);
1016. IntentFilter sdFilter = **new** IntentFilter();
1017. sdFilter.addAction(Intent.ACTION\_EXTERNAL\_APPLICATIONS\_UNAVAILABLE);
1018. sdFilter.addAction(Intent.ACTION\_USER\_STOPPED);
1019. sdFilter.addAction(Intent.ACTION\_UID\_REMOVED);
1020. getContext().registerReceiverAsUser(**this**, UserHandle.ALL, sdFilter, **null**, **null**);
1021. }
1023. @Override
1024. **public** **void** onReceive(Context context, Intent intent) {
1025. **synchronized** (mLock) {
1026. String action = intent.getAction();
1027. String pkgList[] = **null**;
1028. **int** userHandle = intent.getIntExtra(Intent.EXTRA\_USER\_HANDLE, -1);
1030. **if** (Intent.ACTION\_QUERY\_PACKAGE\_RESTART.equals(action)) {
1031. pkgList = intent.getStringArrayExtra(Intent.EXTRA\_PACKAGES);
1032. **for** (String packageName : pkgList) {
1033. **if** (lookForPackageLocked(packageName, userHandle)) {
1034. setResultCode(Activity.RESULT\_OK);
1035. **return**;
1036. }
1037. }
1038. **return**;
1039. } **else** **if** (Intent.ACTION\_EXTERNAL\_APPLICATIONS\_UNAVAILABLE.equals(action)) {
1040. pkgList = intent.getStringArrayExtra(Intent.EXTRA\_CHANGED\_PACKAGE\_LIST);
1041. } **else** **if** (Intent.ACTION\_USER\_STOPPED.equals(action)) {
1042. **if** (userHandle >= 0) {
1043. removeUserLocked(userHandle);
1044. }
1045. } **else** **if** (Intent.ACTION\_UID\_REMOVED.equals(action)) {
1046. **int** uid = intent.getIntExtra(Intent.EXTRA\_UID, -1);
1047. **if** (uid >= 0) {
1048. mLastAllowWhileIdleDispatch.delete(uid);
1049. }
1050. } **else** {
1051. **if** (Intent.ACTION\_PACKAGE\_REMOVED.equals(action)
1052. && intent.getBooleanExtra(Intent.EXTRA\_REPLACING, **false**)) {
1053. // This package is being updated; don't kill its alarms.
1054. **return**;
1055. }
1056. Uri data = intent.getData();
1057. **if** (data != **null**) {
1058. String pkg = data.getSchemeSpecificPart();
1059. **if** (pkg != **null**) {
1060. pkgList = **new** String[]{pkg};
1061. }
1062. }
1063. }
1064. **if** (pkgList != **null** && (pkgList.length > 0)) {
1065. **for** (String pkg : pkgList) {
1066. removeLocked(pkg, userHandle);
1067. removePackagePrio(pkg, userHandle);
1068. **for** (**int** i=mBroadcastStats.size()-1; i>=0; i--) {
1069. ArrayMap<String, BroadcastStats> uidStats = mBroadcastStats.valueAt(i);
1070. BroadcastStats stat = uidStats.get(pkg);
1071. **if** (stat != **null** && UserHandle.getUserId(stat.mUid) == userHandle &&
1072. stat.mPackageName.equals(pkg)) {
1073. **if** (uidStats.remove(pkg) != **null**) {
1074. **if** (uidStats.size() <= 0) {
1075. mBroadcastStats.removeAt(i);
1076. }
1077. }
1078. }
1079. }
1080. }
1081. }
1082. }
1083. }
1084. }
1086. **static** **final** **int** TIME\_CHANGED\_MASK = 1 << 16;
1087. // 上一次时间变化的时钟时间
1088. **long** mLastTimeChangeClockTime;
1089. // 上一次时间变化的开机时间
1090. **long** mLastTimeChangeRealtime;
1091. **static** **final** **boolean** WAKEUP\_STATS = **false**;
1093. **private** **class** AlarmThread **extends** Thread
1094. {
1095. **public** AlarmThread()
1096. {
1097. **super**("AlarmManager");
1098. }
1100. **public** **void** run()
1101. {
1102. ArrayList<Alarm> triggerList = **new** ArrayList<Alarm>();
1104. // 死循环
1105. **while** (**true**)
1106. {
1107. // 等待驱动触发alarm的执行
1108. **int** result = waitForAlarm(mNativeData);
1110. triggerList.clear();
1112. **final** **long** nowRTC = System.currentTimeMillis();
1113. **final** **long** nowELAPSED = SystemClock.elapsedRealtime();
1115. **if** ((result & TIME\_CHANGED\_MASK) != 0) {
1116. // 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。
1117. **final** **long** lastTimeChangeClockTime;
1118. **final** **long** expectedClockTime;
1119. **synchronized** (mLock) {
1120. lastTimeChangeClockTime = mLastTimeChangeClockTime;
1121. expectedClockTime = lastTimeChangeClockTime
1122. + (nowELAPSED - mLastTimeChangeRealtime);
1123. }
1124. **if** (lastTimeChangeClockTime == 0 || nowRTC < (expectedClockTime-500)
1125. || nowRTC > (expectedClockTime+500)) {
1126. // The change is by at least +/- 500 ms (or this is the first change),
1127. // let's do it!
1128. **if** (DEBUG\_BATCH) {
1129. Slog.v(TAG, "Time changed notification from kernel; rebatching");
1130. }
1131. // 由于时间变化，所以要重新批处理所有的alarm
1132. removeImpl(mTimeTickSender);
1133. rebatchAllAlarms();
1134. mClockReceiver.scheduleTimeTickEvent();
1135. **synchronized** (mLock) {
1136. mNumTimeChanged++;
1137. mLastTimeChangeClockTime = nowRTC;
1138. mLastTimeChangeRealtime = nowELAPSED;
1139. }
1140. // 发送时间变化的广播
1141. Intent intent = **new** Intent(Intent.ACTION\_TIME\_CHANGED);
1142. intent.addFlags(Intent.FLAG\_RECEIVER\_REPLACE\_PENDING
1143. | Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY\_BEFORE\_BOOT);
1144. getContext().sendBroadcastAsUser(intent, UserHandle.ALL);
1146. // The world has changed on us, so we need to re-evaluate alarms
1147. // regardless of whether the kernel has told us one went off.
1148. result |= IS\_WAKEUP\_MASK;
1149. }
1150. }
1152. **if** (result != TIME\_CHANGED\_MASK) {
1153. // If this was anything besides just a time change, then figure what if
1154. // anything to do about alarms.
1155. **synchronized** (mLock) {
1156. **if** (localLOGV) Slog.v(
1157. TAG, "Checking for alarms... rtc=" + nowRTC
1158. + ", elapsed=" + nowELAPSED);
1160. // 触发开始执行alarm，如果有wakeup类型的alarm则返回true
1161. **boolean** hasWakeup = triggerAlarmsLocked(triggerList, nowELAPSED, nowRTC);
1162. **if** (!hasWakeup && checkAllowNonWakeupDelayLocked(nowELAPSED)) {
1163. // 如果是非wakeup类型的alarm且处于息屏状态，可以延迟执行alarm
1165. **if** (mPendingNonWakeupAlarms.size() == 0) {
1166. // 记录延迟执行的非wakeup的alarm的开始延迟的时间点
1167. mStartCurrentDelayTime = nowELAPSED;
1168. // 计算下一次非wakeup类型的alarm传递的时间
1169. mNextNonWakeupDeliveryTime = nowELAPSED
1170. + ((currentNonWakeupFuzzLocked(nowELAPSED)\*3)/2);
1171. }
1173. // 把所有的alarm都添加到延迟执行列表中
1174. mPendingNonWakeupAlarms.addAll(triggerList);
1175. mNumDelayedAlarms += triggerList.size();
1176. rescheduleKernelAlarmsLocked();
1177. updateNextAlarmClockLocked();
1178. } **else** {
1179. // 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。
1180. rescheduleKernelAlarmsLocked();
1181. updateNextAlarmClockLocked();
1182. **if** (mPendingNonWakeupAlarms.size() > 0) {
1183. // 计算延迟执行alarm的传递优先级
1184. calculateDeliveryPriorities(mPendingNonWakeupAlarms);
1185. triggerList.addAll(mPendingNonWakeupAlarms);
1186. Collections.sort(triggerList, mAlarmDispatchComparator);
1187. **final** **long** thisDelayTime = nowELAPSED - mStartCurrentDelayTime;
1188. mTotalDelayTime += thisDelayTime;
1189. **if** (mMaxDelayTime < thisDelayTime) {
1190. mMaxDelayTime = thisDelayTime;
1191. }
1192. mPendingNonWakeupAlarms.clear();
1194. }
1195. // 执行alarm
1196. deliverAlarmsLocked(triggerList, nowELAPSED);
1197. }
1198. }
1199. }
1200. }
1201. }
1202. }
1204. **void** removeImpl(PendingIntent operation) {
1205. **if** (operation == **null**) {
1206. **return**;
1207. }
1209. **synchronized** (mLock) {
1210. // 移除掉alarm的PendingIntent
1211. removeLocked(operation);
1212. }
1213. }
1215. /\*\*
1216. \* key是uid，value是alarm的上一次执行时间
1217. \*/
1218. **final** SparseLongArray mLastAllowWhileIdleDispatch = **new** SparseLongArray();
1219. // 可以开始执行flag为ALLOW\_WHILE\_IDLE的alarm的最小时间间隔
1220. **long** mAllowWhileIdleMinTime;
1222. **int** mCurrentSeq = 0;

1225. // 执行alarm
1226. **boolean** triggerAlarmsLocked(ArrayList<Alarm> triggerList, **final** **long** nowELAPSED,
1227. **final** **long** nowRTC) {
1228. **boolean** hasWakeup = **false**;
1229. **while** (mAlarmBatches.size() > 0) {
1230. // 从列表中取出第一个批处理对象
1231. Batch batch = mAlarmBatches.get(0);
1232. // 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
1233. **if** (batch.start > nowELAPSED) {
1234. // Everything else is scheduled for the future
1235. **break**;
1236. }
1238. // 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
1239. mAlarmBatches.remove(0);
1241. // 获取批处理对象中Alarm的数量
1242. **final** **int** N = batch.size();
1243. **for** (**int** i = 0; i < N; i++) {
1244. Alarm alarm = batch.get(i);
1246. // 判断alarm是否设置了空闲时允许执行任务的flag
1247. **if** ((alarm.flags&AlarmManager.FLAG\_ALLOW\_WHILE\_IDLE) != 0) {
1248. // 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。
1250. // 根据uid获取alarm的上一次开始执行时间
1251. **long** lastTime = mLastAllowWhileIdleDispatch.get(alarm.uid, 0);
1252. // 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
1253. **long** minTime = lastTime + mAllowWhileIdleMinTime;
1254. // 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
1255. **if** (nowELAPSED < minTime) {
1256. // 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。
1258. // 把下一次的最早开始执行时间赋值给alarm的whenElapsed变量
1259. alarm.whenElapsed = minTime;
1260. // 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
1261. **if** (alarm.maxWhenElapsed < minTime) {
1262. alarm.maxWhenElapsed = minTime;
1263. }
1264. // 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
1265. setImplLocked(alarm, **true**, **false**);
1266. **continue**;
1267. }
1268. }
1270. // 设置alarm的执行次数为1
1271. alarm.count = 1;
1272. // 如果alarm不需要重新调度，则就把该alarm添加到triggerList列表中
1273. triggerList.add(alarm);
1274. **if** ((alarm.flags&AlarmManager.FLAG\_WAKE\_FROM\_IDLE) != 0) {
1275. EventLogTags.writeDeviceIdleWakeFromIdle(mPendingIdleUntil != **null** ? 1 : 0,
1276. alarm.tag);
1277. }
1278. // 判断该alarm是否是延迟到空闲时开始执行的alarm
1279. **if** (mPendingIdleUntil == alarm) {
1280. // 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
1281. mPendingIdleUntil = **null**;
1282. // 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
1283. rebatchAllAlarmsLocked(**false**);
1284. // 由于mPendingIdleUntil变量改变，所以需要重新存储空闲时延迟执行的alarm
1285. restorePendingWhileIdleAlarmsLocked();
1286. }
1287. // 判断该alarm是否是下一个请求退出空闲模式的alarm
1288. **if** (mNextWakeFromIdle == alarm) {
1289. // 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
1290. mNextWakeFromIdle = **null**;
1291. // 对所有的alarm进行重新批处理操作
1292. rebatchAllAlarmsLocked(**false**);
1293. }
1295. **if** (alarm.repeatInterval > 0) {
1296. // 计算重复执行的alarm从开始执行到现在的执行次数
1297. alarm.count += (nowELAPSED - alarm.whenElapsed) / alarm.repeatInterval;
1299. // 计算alarm的下一次开始执行时间
1300. **final** **long** delta = alarm.count \* alarm.repeatInterval;
1301. **final** **long** nextElapsed = alarm.whenElapsed + delta;
1302. // 根据新得到的alarm参数，设置下次触发的alarm
1303. setImplLocked(alarm.type, alarm.when + delta, nextElapsed, alarm.windowLength,
1304. maxTriggerTime(nowELAPSED, nextElapsed, alarm.repeatInterval),
1305. alarm.repeatInterval, alarm.operation, alarm.flags, **true**,
1306. alarm.workSource, alarm.alarmClock, alarm.uid);
1307. }
1309. **if** (alarm.wakeup) {
1310. hasWakeup = **true**;
1311. }
1313. // We removed an alarm clock. Let the caller recompute the next alarm clock.
1314. **if** (alarm.alarmClock != **null**) {
1315. mNextAlarmClockMayChange = **true**;
1316. }
1317. }
1318. }
1320. // This is a new alarm delivery set; bump the sequence number to indicate that
1321. // all apps' alarm delivery classes should be recalculated.
1322. // 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
1323. mCurrentSeq++;
1324. // 重新计算alarm的传递优先级
1325. calculateDeliveryPriorities(triggerList);
1326. // 根据优先级排序
1327. Collections.sort(triggerList, mAlarmDispatchComparator);
1329. **if** (localLOGV) {
1330. **for** (**int** i=0; i<triggerList.size(); i++) {
1331. Slog.v(TAG, "Triggering alarm #" + i + ": " + triggerList.get(i));
1332. }
1333. }
1335. **return** hasWakeup;
1336. }
1338. // alarm传递优先级定义
1339. **static** **final** **int** PRIO\_TICK = 0;
1340. **static** **final** **int** PRIO\_WAKEUP = 1;
1341. **static** **final** **int** PRIO\_NORMAL = 2;
1343. // 计算alarm的传递优先级
1344. **void** calculateDeliveryPriorities(ArrayList<Alarm> alarms) {
1345. **final** **int** N = alarms.size();
1346. **for** (**int** i = 0; i < N; i++) {
1347. Alarm a = alarms.get(i);
1349. **final** **int** alarmPrio;
1350. **if** (Intent.ACTION\_TIME\_TICK.equals(a.operation.getIntent().getAction())) {
1351. alarmPrio = PRIO\_TICK;
1352. } **else** **if** (a.wakeup) {
1353. alarmPrio = PRIO\_WAKEUP;
1354. } **else** {
1355. alarmPrio = PRIO\_NORMAL;
1356. }
1358. PriorityClass packagePrio = a.priorityClass;
1359. **if** (packagePrio == **null**) {
1360. packagePrio = a.priorityClass = **new** PriorityClass(); // lowest prio & stale sequence
1361. }
1362. a.priorityClass = packagePrio;
1364. **if** (packagePrio.seq != mCurrentSeq) {
1365. // first alarm we've seen in the current delivery generation from this package
1366. packagePrio.priority = alarmPrio;
1367. packagePrio.seq = mCurrentSeq;
1368. } **else** {
1369. // Multiple alarms from this package being delivered in this generation;
1370. // bump the package's delivery class if it's warranted.
1371. // TICK < WAKEUP < NORMAL
1372. **if** (alarmPrio < packagePrio.priority) {
1373. packagePrio.priority = alarmPrio;
1374. }
1375. }
1376. }
1377. }
1379. **final** **class** PriorityClass {
1380. **int** seq;
1381. **int** priority;
1383. PriorityClass() {
1384. seq = mCurrentSeq - 1;
1385. priority = PRIO\_NORMAL;
1386. }
1387. }
1389. // 根据优先级排序，优先级相同的则根据开始执行时间排序
1390. **final** Comparator<Alarm> mAlarmDispatchComparator = **new** Comparator<Alarm>() {
1391. @Override
1392. **public** **int** compare(Alarm lhs, Alarm rhs) {
1393. // priority class trumps everything.  TICK < WAKEUP < NORMAL
1394. **if** (lhs.priorityClass.priority < rhs.priorityClass.priority) {
1395. **return** -1;
1396. } **else** **if** (lhs.priorityClass.priority > rhs.priorityClass.priority) {
1397. **return** 1;
1398. }
1400. // within each class, sort by nominal delivery time
1401. **if** (lhs.whenElapsed < rhs.whenElapsed) {
1402. **return** -1;
1403. } **else** **if** (lhs.whenElapsed > rhs.whenElapsed) {
1404. **return** 1;
1405. }
1407. // same priority class + same target delivery time
1408. **return** 0;
1409. }
1410. };
1412. **private** **native** **int** waitForAlarm(**long** nativeData);
1414. // 检查是否允许非wakeup类型的alarm延迟执行
1415. **boolean** checkAllowNonWakeupDelayLocked(**long** nowELAPSED) {
1417. /\*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\*/
1419. // 如果当前是亮屏状态，则返回false
1420. **if** (mInteractive) {
1421. **return** **false**;
1422. }
1423. **if** (mLastAlarmDeliveryTime <= 0) {
1424. **return** **false**;
1425. }
1426. **if** (mPendingNonWakeupAlarms.size() > 0 && mNextNonWakeupDeliveryTime < nowELAPSED) {
1427. // This is just a little paranoia, if somehow we have pending non-wakeup alarms
1428. // and the next delivery time is in the past, then just deliver them all.  This
1429. // avoids bugs where we get stuck in a loop trying to poll for alarms.
1430. **return** **false**;
1431. }
1432. **long** timeSinceLast = nowELAPSED - mLastAlarmDeliveryTime;
1433. // 判断上次alarm传递的时间是否小于延迟时间
1434. **return** timeSinceLast <= currentNonWakeupFuzzLocked(nowELAPSED);
1435. }
1437. // 根据息屏状态持续的时间决定延迟的时间
1438. **long** currentNonWakeupFuzzLocked(**long** nowELAPSED) {
1439. // 息屏状态持续时间=当前时间-开始息屏的时间
1440. **long** timeSinceOn = nowELAPSED - mNonInteractiveStartTime;
1441. **if** (timeSinceOn < 5\*60\*1000) {
1442. // If the screen has been off for 5 minutes, only delay by at most two minutes.
1443. **return** 2\*60\*1000;
1444. } **else** **if** (timeSinceOn < 30\*60\*1000) {
1445. // If the screen has been off for 30 minutes, only delay by at most 15 minutes.
1446. **return** 15\*60\*1000;
1447. } **else** {
1448. // Otherwise, we will delay by at most an hour.
1449. **return** 60\*60\*1000;
1450. }
1451. }
1453. **private** **final** IAlarmManager mService;
1455. **private** **final** IBinder mService = **new** IAlarmManager.Stub() {
1456. @Override
1457. **public** **void** set(**int** type, **long** triggerAtTime, **long** windowLength, **long** interval, **int** flags,
1458. PendingIntent operation, WorkSource workSource,
1459. AlarmManager.AlarmClockInfo alarmClock) {
1460. **final** **int** callingUid = Binder.getCallingUid();
1461. **if** (workSource != **null**) {
1462. getContext().enforcePermission(
1463. android.Manifest.permission.UPDATE\_DEVICE\_STATS,
1464. Binder.getCallingPid(), callingUid, "AlarmManager.set");
1465. }
1467. // No incoming callers can request either WAKE\_FROM\_IDLE or
1468. // ALLOW\_WHILE\_IDLE\_UNRESTRICTED -- we will apply those later as appropriate.
1469. flags &= ~(AlarmManager.FLAG\_WAKE\_FROM\_IDLE
1470. | AlarmManager.FLAG\_ALLOW\_WHILE\_IDLE\_UNRESTRICTED);
1472. // Only the system can use FLAG\_IDLE\_UNTIL -- this is used to tell the alarm
1473. // manager when to come out of idle mode, which is only for DeviceIdleController.
1474. **if** (callingUid != Process.SYSTEM\_UID) {
1475. flags &= ~AlarmManager.FLAG\_IDLE\_UNTIL;
1476. }
1478. // If the caller is a core system component, and not calling to do work on behalf
1479. // of someone else, then always set ALLOW\_WHILE\_IDLE\_UNRESTRICTED.  This means we
1480. // will allow these alarms to go off as normal even while idle, with no timing
1481. // restrictions.
1482. **if** (callingUid < Process.FIRST\_APPLICATION\_UID && workSource == **null**) {
1483. flags |= AlarmManager.FLAG\_ALLOW\_WHILE\_IDLE\_UNRESTRICTED;
1484. }
1486. // If this is an exact time alarm, then it can't be batched with other alarms.
1487. **if** (windowLength == AlarmManager.WINDOW\_EXACT) {
1488. flags |= AlarmManager.FLAG\_STANDALONE;
1489. }
1491. // If this alarm is for an alarm clock, then it must be standalone and we will
1492. // use it to wake early from idle if needed.
1493. **if** (alarmClock != **null**) {
1494. flags |= AlarmManager.FLAG\_WAKE\_FROM\_IDLE | AlarmManager.FLAG\_STANDALONE;
1495. }
1497. // 调用AlarmManagerService类的setImpl方法设置alarm
1498. setImpl(type, triggerAtTime, windowLength, interval, operation,
1499. flags, workSource, alarmClock, callingUid);
1500. }
1502. @Override
1503. **public** **boolean** setTime(**long** millis) {
1504. getContext().enforceCallingOrSelfPermission(
1505. "android.permission.SET\_TIME",
1506. "setTime");
1508. **if** (mNativeData == 0) {
1509. Slog.w(TAG, "Not setting time since no alarm driver is available.");
1510. **return** **false**;
1511. }
1513. **synchronized** (mLock) {
1514. **return** setKernelTime(mNativeData, millis) == 0;
1515. }
1516. }
1518. @Override
1519. **public** **void** setTimeZone(String tz) {
1520. getContext().enforceCallingOrSelfPermission(
1521. "android.permission.SET\_TIME\_ZONE",
1522. "setTimeZone");
1524. **final** **long** oldId = Binder.clearCallingIdentity();
1525. **try** {
1526. setTimeZoneImpl(tz);
1527. } **finally** {
1528. Binder.restoreCallingIdentity(oldId);
1529. }
1530. }
1532. @Override
1533. **public** **void** remove(PendingIntent operation) {
1534. removeImpl(operation);
1536. }
1538. @Override
1539. **public** **long** getNextWakeFromIdleTime() {
1540. **return** getNextWakeFromIdleTimeImpl();
1541. }
1543. @Override
1544. **public** AlarmManager.AlarmClockInfo getNextAlarmClock(**int** userId) {
1545. userId = ActivityManager.handleIncomingUser(Binder.getCallingPid(),
1546. Binder.getCallingUid(), userId, **false** /\* allowAll \*/, **false** /\* requireFull \*/,
1547. "getNextAlarmClock", **null**);
1549. **return** getNextAlarmClockImpl(userId);
1550. }
1552. @Override
1553. **protected** **void** dump(FileDescriptor fd, PrintWriter pw, String[] args) {
1554. . . .
1555. }
1557. @Override
1558. /\* updates the blocked uids, so if a wake lock is acquired to only fire
1559. \* alarm for it, it can be released.
1560. \*/
1561. **public** **void** updateBlockedUids(**int** uid, **boolean** isBlocked) {
1563. **if** (localLOGV) Slog.v(TAG, "UpdateBlockedUids: uid = " + uid +
1564. " isBlocked = " + isBlocked);
1566. **if** (Binder.getCallingUid() != Process.SYSTEM\_UID) {
1567. **if** (localLOGV) Slog.v(TAG, "UpdateBlockedUids is not allowed");
1568. **return**;
1569. }
1571. **synchronized**(mLock) {
1572. **if**(isBlocked) {
1573. mBlockedUids.add(**new** Integer(uid));
1574. **if** (checkReleaseWakeLock()) {
1575. /\* all the uids for which the alarms are triggered
1576. \* are either blocked or have called onSendFinished.
1577. \*/
1578. **if** (mWakeLock.isHeld()) {
1579. mWakeLock.release();
1580. **if** (localLOGV)
1581. Slog.v(TAG, "AM WakeLock Released Internally in updateBlockedUids");
1582. }
1583. }
1584. } **else** {
1585. mBlockedUids.clear();
1586. }
1587. }
1588. }
1589. };
1591. SystemService类中的publishBinderService方法：
1592. /\*\*
1593. \* 发布服务，以便其他服务或应用程序能调用该服务。
1594. \*/
1595. **protected** **final** **void** publishBinderService(String name, IBinder service) {
1596. publishBinderService(name, service, **false**);
1597. }
1599. /\*\*
1600. \* 发布服务，以便其他服务或应用程序能调用该服务。
1601. \*/
1602. **protected** **final** **void** publishBinderService(String name, IBinder service,
1603. **boolean** allowIsolated) {
1604. // 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
1605. // getSystemService时就是根据name获取到该服务的
1606. ServiceManager.addService(name, service, allowIsolated);
1607. }

下面看下AlarmManagerService类的构造方法中初始化的Handler的实现：

**[java]** [view plain](http://blog.csdn.net/zhangyongfeiyong/article/details/52224300) [copy](http://blog.csdn.net/zhangyongfeiyong/article/details/52224300)

1. **final** Object mLock = **new** Object();
3. **private** **class** AlarmHandler **extends** Handler {
4. **public** **static** **final** **int** ALARM\_EVENT = 1;
5. **public** **static** **final** **int** MINUTE\_CHANGE\_EVENT = 2;
6. **public** **static** **final** **int** DATE\_CHANGE\_EVENT = 3;
7. **public** **static** **final** **int** SEND\_NEXT\_ALARM\_CLOCK\_CHANGED = 4;
9. **public** AlarmHandler() {
10. }
12. **public** **void** handleMessage(Message msg) {
13. **if** (msg.what == ALARM\_EVENT) {
14. ArrayList<Alarm> triggerList = **new** ArrayList<Alarm>();
15. **synchronized** (mLock) {
16. **final** **long** nowRTC = System.currentTimeMillis();
17. **final** **long** nowELAPSED = SystemClock.elapsedRealtime();
18. // 触发开始执行alarm
19. triggerAlarmsLocked(triggerList, nowELAPSED, nowRTC);
20. // 更新下一个alarm的alarmClock
21. updateNextAlarmClockLocked();
22. }
24. // now trigger the alarms without the lock held
25. **for** (**int** i=0; i<triggerList.size(); i++) {
26. Alarm alarm = triggerList.get(i);
27. **try** {
28. // 执行和PendingIntent关联的操作
29. alarm.operation.send();
30. } **catch** (PendingIntent.CanceledException e) {
31. **if** (alarm.repeatInterval > 0) {
32. // 移除掉alarm的PendingIntent
33. removeImpl(alarm.operation);
34. }
35. }
36. }
37. } **else** **if** (msg.what == SEND\_NEXT\_ALARM\_CLOCK\_CHANGED) {
38. // 处理下一个alarm的开始执行时间改变的逻辑
39. sendNextAlarmClockChanged();
40. }
41. }
42. }
44. **private** **final** SparseArray<AlarmManager.AlarmClockInfo> mHandlerSparseAlarmClockArray =
45. **new** SparseArray<>();
46. **private** **static** **final** Intent NEXT\_ALARM\_CLOCK\_CHANGED\_INTENT = **new** Intent(
47. AlarmManager.ACTION\_NEXT\_ALARM\_CLOCK\_CHANGED);
49. // 处理下一个alarm的开始执行时间改变的逻辑
50. **private** **void** sendNextAlarmClockChanged() {
51. SparseArray<AlarmManager.AlarmClockInfo> pendingUsers = mHandlerSparseAlarmClockArray;
52. pendingUsers.clear();
54. **synchronized** (mLock) {
55. **final** **int** N  = mPendingSendNextAlarmClockChangedForUser.size();
56. // 循环把mPendingSendNextAlarmClockChangedForUser中的值添加到pendingUsers中
57. **for** (**int** i = 0; i < N; i++) {
58. **int** userId = mPendingSendNextAlarmClockChangedForUser.keyAt(i);
59. pendingUsers.append(userId, mNextAlarmClockForUser.get(userId));
60. }
61. mPendingSendNextAlarmClockChangedForUser.clear();
62. }
64. **final** **int** N = pendingUsers.size();
65. **for** (**int** i = 0; i < N; i++) {
66. **int** userId = pendingUsers.keyAt(i);
67. AlarmManager.AlarmClockInfo alarmClock = pendingUsers.valueAt(i);
68. Settings.System.putStringForUser(getContext().getContentResolver(),
69. Settings.System.NEXT\_ALARM\_FORMATTED,
70. // 格式化时间
71. formatNextAlarm(getContext(), alarmClock, userId),
72. userId);
74. // 发送广播，接收广播的是状态栏
75. getContext().sendBroadcastAsUser(NEXT\_ALARM\_CLOCK\_CHANGED\_INTENT,
76. **new** UserHandle(userId));
77. }
78. }
80. /\*\*
81. \* Formats an alarm like platform/packages/apps/DeskClock used to.
82. \*/
83. **private** **static** String formatNextAlarm(**final** Context context, AlarmManager.AlarmClockInfo info,
84. **int** userId) {
85. String skeleton = DateFormat.is24HourFormat(context, userId) ? "EHm" : "Ehma";
86. String pattern = DateFormat.getBestDateTimePattern(Locale.getDefault(), skeleton);
87. **return** (info == **null**) ? "" :
88. DateFormat.format(pattern, info.getTriggerTime()).toString();
89. }

下面分析下使用AlarmManagerService的流程：

应用通过getSystemService方法获取系统服务时，会调用到ContextImpl类的getSystemService方法：

**[java]** [view plain](http://blog.csdn.net/zhangyongfeiyong/article/details/52224300) [copy](http://blog.csdn.net/zhangyongfeiyong/article/details/52224300)

1. @Override
2. **public** Object getSystemService(String name) {
3. **return** SystemServiceRegistry.getSystemService(**this**, name);
4. }

SystemServiceRegistry类中的getSystemService方法：

**[java]** [view plain](http://blog.csdn.net/zhangyongfeiyong/article/details/52224300) [copy](http://blog.csdn.net/zhangyongfeiyong/article/details/52224300)

1. /\*\*
2. \* Gets a system service from a given context.
3. \*/
4. **public** **static** Object getSystemService(ContextImpl ctx, String name) {
5. // 根据服务名从map中取出ServiceFetcher
6. ServiceFetcher<?> fetcher = SYSTEM\_SERVICE\_FETCHERS.get(name);
7. // 如果fetcher不为null，则从fetcher中根据ctx取出对应的服务并返回
8. **return** fetcher != **null** ? fetcher.getService(ctx) : **null**;
9. }

但是在SystemServiceRegistry类的静态代码块中注册服务时，就是把上面发布到ServiceManager中的Binder代理对象返回给createService方法：

**[java]** [view plain](http://blog.csdn.net/zhangyongfeiyong/article/details/52224300) [copy](http://blog.csdn.net/zhangyongfeiyong/article/details/52224300)

1. **static** {
2. . . .
3. registerService(Context.ALARM\_SERVICE, AlarmManager.**class**,
4. **new** CachedServiceFetcher<AlarmManager>() {
5. @Override
6. **public** AlarmManager createService(ContextImpl ctx) {
7. IBinder b = ServiceManager.getService(Context.ALARM\_SERVICE);
8. IAlarmManager service = IAlarmManager.Stub.asInterface(b);
9. **return** **new** AlarmManager(service, ctx);
10. }});
11. }
13. **private** **static** <T> **void** registerService(String serviceName, Class<T> serviceClass,
14. ServiceFetcher<T> serviceFetcher) {
15. SYSTEM\_SERVICE\_NAMES.put(serviceClass, serviceName);
16. SYSTEM\_SERVICE\_FETCHERS.put(serviceName, serviceFetcher);
17. }
19. **static** **abstract** **interface** ServiceFetcher<T> {
20. T getService(ContextImpl ctx);
21. }
23. **static** **abstract** **class** StaticServiceFetcher<T> **implements** ServiceFetcher<T> {
24. **private** T mCachedInstance;
26. @Override
27. **public** **final** T getService(ContextImpl unused) {
28. // 单例懒加载模式，具体的创建由子类实现
29. **synchronized** (StaticServiceFetcher.**this**) {
30. **if** (mCachedInstance == **null**) {
31. mCachedInstance = createService();
32. }
33. **return** mCachedInstance;
34. }
35. }
37. **public** **abstract** T createService();
38. }

看下AlarmManager类：

**[java]** [view plain](http://blog.csdn.net/zhangyongfeiyong/article/details/52224300) [copy](http://blog.csdn.net/zhangyongfeiyong/article/details/52224300)

1. **private** **final** **boolean** mAlwaysExact;
2. **private** **final** **int** mTargetSdkVersion;
4. AlarmManager(IAlarmManager service, Context ctx) {
5. mService = service;
7. mTargetSdkVersion = ctx.getApplicationInfo().targetSdkVersion;
8. // API=19之前的alarm都是精确执行的
9. mAlwaysExact = (mTargetSdkVersion < Build.VERSION\_CODES.KITKAT);
10. }

也就是说，其他应用通过getSystemService(Context.ALARM\_SERVICE)方法获取到的AlarmManager对象中的Binder对象其实是AlarmManagerService类中的IAlarmManager代理对象。

下面看下其他应用调用AlarmManager类的setExact方法的流程：

**[java]** [view plain](http://blog.csdn.net/zhangyongfeiyong/article/details/52224300) [copy](http://blog.csdn.net/zhangyongfeiyong/article/details/52224300)

1. **public** **void** setExact(**int** type, **long** triggerAtMillis, PendingIntent operation) {
2. setImpl(type, triggerAtMillis, WINDOW\_EXACT, 0, 0, operation, **null**, **null**);
3. }
5. **private** **void** setImpl(**int** type, **long** triggerAtMillis, **long** windowMillis, **long** intervalMillis,
6. **int** flags, PendingIntent operation, WorkSource workSource, AlarmClockInfo alarmClock) {
7. **if** (triggerAtMillis < 0) {
8. /\* NOTYET
9. if (mAlwaysExact) {
10. // Fatal error for KLP+ apps to use negative trigger times
11. throw new IllegalArgumentException("Invalid alarm trigger time "
12. + triggerAtMillis);
13. }
14. \*/
15. triggerAtMillis = 0;
16. }
18. **try** {
19. // 调用Binder代理对象的set方法
20. mService.set(type, triggerAtMillis, windowMillis, intervalMillis, flags, operation,
21. workSource, alarmClock);
22. } **catch** (RemoteException ex) {
23. }
24. }

由上面的分析知，Binder代理对象调用的是AlarmManagerService类的setImpl方法设置alarm的。