原 AlarmManager研究

*4*人收藏此文章, 我要收藏 發表於2個月前(2013-08-02 20:17) ,已有**144**次閱讀，共**2**個評論

目錄：[-]

AlarmManager研究

侯亮

1.概述

在Android系統中，鬧鐘和喚醒功能都是由Alarm Manager Service控制並管理的。我們所熟悉的RTC鬧鐘以及定時器都和它有莫大的關係。為了便於稱呼，我常常也把這個service簡稱為ALMS。

另外，ALMS還提供了一個AlarmManager輔助類。在實際的代碼中，應用程序一般都是通過這個輔助類來和ALMS打交道的。就代碼而言，輔助類只不過是把一些邏輯語義傳遞給ALMS服務端而已，具體怎麼做則完全要看ALMS的實現代碼了。

ALMS的實現代碼並不算太複雜，主要只是在管理“邏輯鬧鐘”。它把邏輯鬧鐘分成幾個大類，分別記錄在不同的列表中。然後ALMS會在一個專門的線程中循環等待鬧鐘的激發，一旦時機到了，就“回調”邏輯鬧鐘對應的動作。

以上只是一些概要性的介紹，下面我們來看具體的技術細節。

2.AlarmManager

前文我們已經說過，ALMS只是服務端的東西。它必須向外提供具體的接口，才能被外界使用。在Android平台中，ALMS的外部接口為IAlarmManager。其定義位於frameworks\base\core\java\android\app\IAlarmManager.aidl腳本中，定義截選如下：

```
interface IAlarmManager {
     void set( int type, long triggerAtTime, in PendingIntent operation);
     void setRepeating( int type, long triggerAtTime, long interval, in PendingIntent operation);
     void setInexactRepeating( int type, long triggerAtTime, long interval, in PendingIntent operation) ;
     void setTime( long millis);
     void setTimeZone(String zone);
     void remove( in PendingIntent operation);
}
```

在一般情況下，service的使用者會通過Service Manager Service接口，先拿到它感興趣的service對應的代理I接口，然後再調用I接口的成員函數向service發出請求。所以按理說，我們也應該先拿到一個IAlarmManager接口，然後再使用它。可是，對Alarm Manager Service來說，情況略有不同，其最常見的調用方式如下：

```
manager = (AlarmManager)context.getSystemService(Context.ALARM_SERVICE);
```

其中，getSystemService()返回的不再是IAlarmManager接口，而是AlarmManager對象。

我們參考AlarmManager.java文件，可以看到AlarmManager類中聚合了一個IAlarmManager接口，

```
private final IAlarmManager mService;
```

也就是說在執行實際動作時，AlarmManager只不過是把外界的請求轉發給內部聚合的IAlarmManager接口而已。

2.1 AlarmManager的成員函數

AlarmManager的成員函數有：

```
AlarmManager(IAlarmManager service)
 public  void set( int type, long triggerAtTime, PendingIntent operation)
 public  void setRepeating( int type, long triggerAtTime, long interval,
PendingIntent operation)
public  void setInexactRepeating( int type, long triggerAtTime, long interval,
                                  PendingIntent operation)
public  void cancel(PendingIntent operation)
 public  void setTime( long millis)
 public  void setTimeZone(String timeZone)
```

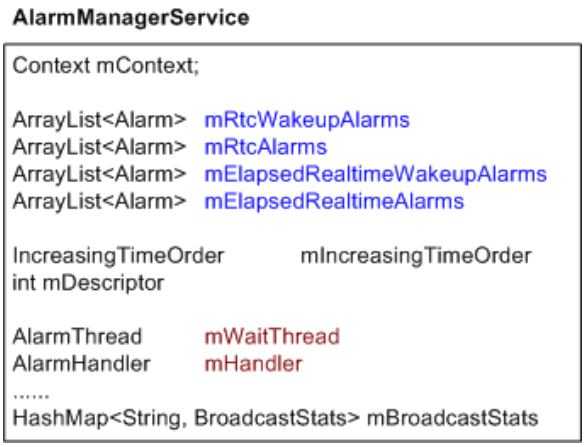
即1個構造函數，6個功能函數。基本上完全和IAlarmManager的成員函數一一對應。

另外，AlarmManager類中會以不同的公共常量來表示多種不同的邏輯鬧鐘，在Android 4.0的原生代碼中有4種邏輯鬧鐘：
1）RTC_WAKEUP
2）RTC
3）ELAPSED_REALTIME_WAKEUP
4）ELAPSED_REALTIME

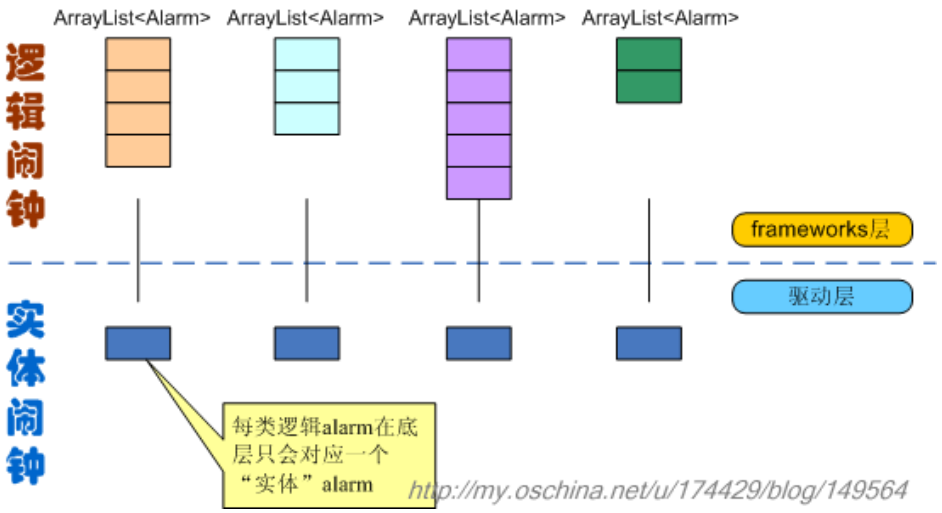
應用側通過調用AlarmManager對象的成員函數，可以把語義傳遞到AlarmManagerService，並由它進行實際的處理。

3.AlarmManagerService

ALMS的重頭戲在AlarmManagerService中，這個類繼承於IAlarmManager.Stub，所以是個binder實體。它包含的重要成員如下：



其中，mRtcWakeupAlarms等4個ArrayList<Alarm>數組分別對應著前文所說的4種“邏輯鬧鐘”。為了便於理解，我們可以想像在底層有4個“實體鬧鐘”，注意，是4個，不是4類。上面每一類“邏輯鬧鐘”都會對應一個“實體鬧鐘”，而邏輯鬧鐘則可以有若干個，它們被存儲在ArrayList中，示意圖如下：



當然，這裡所說的“實體鬧鐘”只是個概念而已，其具體實現和底層驅動有關，在frameworks層不必過多關心。

Frameworks層應該關心的是那幾個ArrayList<Alarm>。這裡的Alarm對應著邏輯鬧鐘。

3.1 邏輯鬧鐘

Alarm是AlarmManagerService的一個內嵌類Alarm，定義截選如下：

```
private  static  class Alarm {
     public  int type;
     public  int count;
     public  long when;
     public  long repeatInterval;
     public PendingIntent operation;
     public  int uid;
     public  int pid;
    . . . . . .
```

其中記錄了邏輯鬧鐘的一些關鍵信息。

- type域：記錄著邏輯鬧鐘的鬧鐘類型，比如RTC_WAKEUP、ELAPSED_REALTIME_WAKEUP等；
- count域：是個輔助域，它和repeatInterval域一起工作。當repeatInterval大於0時，這個域可被用於計算下一次重複激發alarm的時間，詳細情況見後文；
- when域：記錄鬧鐘的激發時間。這個域和type域相關，詳細情況見後文；
- repeatInterval域：表示重複激發鬧鐘的時間間隔，如果鬧鐘只需激發一次，則此域為0，如果鬧鐘需要重複激發，此域為以毫秒為單位的時間間隔；
- operation域：記錄鬧鐘激發時應該執行的動作，詳細情況見後文；
- uid域：記錄設置鬧鐘的進程的uid；
- pid域：記錄設置鬧鐘的進程的pid。

總體來說還是比較簡單的，我們先補充說明一下其中的count域。這個域是針對重複性鬧鐘的一個輔助域。重複性鬧鐘的實現機理是，如果當前時刻已經超過鬧鐘的激發時刻，那麼ALMS會先從邏輯鬧鐘數組中摘取下Alarm節點，並執行鬧鐘對應的邏輯動作，然後進一步比較“當前時刻”和“ Alarm理應激發的理想時刻”之間的時間跨度，從而計算出Alarm的“下一次理應激發的理想時刻”，並將這個激發時間記入Alarm節點，接著將該節點重新排入邏輯鬧鐘列表。這一點和普通Alarm不太一樣，普通Alarm節點摘下後就不再還回邏輯鬧鐘列表了。

“當前時刻”和“理應激發時刻”之間的時間跨度會隨實際的運作情況而變動。我們分兩步來說明“下一次理應激發時刻”的計算公式：
1） count = (時間跨度/ repeatInterval) + 1 ；
2） “下一次理應激發時刻” = “上一次理應激發時刻” + count * repeatInterval ；

我們畫一張示意圖，其中綠色的可激發時刻表示“上一次理應激發時刻”，我們假定“當前時刻”分別為now_1處或now_2處，可以看到會計算出不同的“下一次理應激發時刻”，這裡用桔紅色表示。

可以看到，如果當前時刻為now_1，那麼它和“上一次理應激發時刻”之間的“時間跨度”是小於一個repeatInterval的，所以count數為1。而如果當前時刻為now_2，那麼“時間跨度”與repeatInterval的商取整後為2，所以count數為3。另外，圖中那兩個虛線箭頭對應的可激發時刻，只是用來做刻度的東西。

3.2 主要行為

接下來我們來看ALMS中的主要行為，這些行為和AlarmManager輔助類提供的成員函數相對應。

3.2.1 設置alarm

外界能接觸的設置alarm的函數是set()：

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

type：表示要設置的alarm類型。如前文所述，有4個alarm類型。
triggerAtTime：表示alarm“理應激發”的時間。
operation：指明了alarm鬧鈴激發時需要執行的動作，比如執行某種廣播通告。

設置alarm的動作會牽扯到一個發起者。簡單地說，發起者會向Alarm Manager Service發出一個設置alarm的請求，而且在請求裡註明了到時間後需要執行的動作。由於“待執行的動作”一般都不會馬上執行，所以要表達成PendingIntent的形式。（PendingIntent的詳情可參考其他文章）

另外，triggerAtTime參數的意義也會隨type參數的不同而不同。簡單地說，如果type是和RTC相關的話，那麼triggerAtTime的值應該是標準時間，即從1970 年1 月1 日午夜開始所經過的毫秒數。而如果type是其他類型的話，那麼triggerAtTime的值應該是從本次開機開始算起的毫秒數。

3.2.2 重複性alarm

另一個設置alarm的函數是setRepeating()：

```
public  void setRepeating( int type, long triggerAtTime, long interval,PendingIntent operation)
```

其參數基本上和set()函數差不多，只是多了一個“時間間隔”參數。事實上，在Alarm Manager Service一側，set()函數內部也是在調用setRepeating()的，只不過會把interval設成了0。

setRepeating()的實現函數如下：

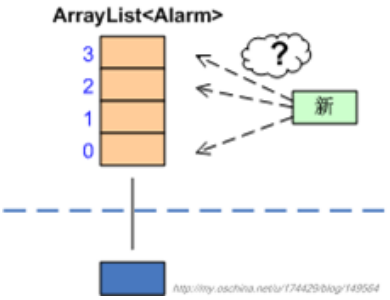
```
public  void setRepeating( int type, long triggerAtTime, long interval,
                        PendingIntent operation)
{
    if (operation == null ) {
        Slog.w(TAG, "set/setRepeating ignored because there is no intent" );
         return ;
    }
    synchronized (mLock) {
        Alarm alarm = 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);
        }
    }
}
```

代碼很簡單，會創建一個邏輯鬧鐘Alarm，而後調用addAlarmLocked()將邏輯鬧鐘添加到內部邏輯鬧鐘數組的某個合適位置。

```
private  int addAlarmLocked(Alarm alarm) {
    ArrayList<Alarm> alarmList = getAlarmList(alarm.type);

    int index = Collections.binarySearch(alarmList, alarm, mIncreasingTimeOrder);
     if (index < 0) {
        index = 0 - index - 1;
    }
    if (localLOGV) Slog.v(TAG, "Adding alarm " + alarm + " at " + index);
    alarmList.add(index, alarm);
    . . . . . .
    return index;
}
```

邏輯鬧鐘列表是依據alarm的激發時間進行排序的，越早激發的alarm，越靠近第0位。所以，addAlarmLocked()在添加新邏輯鬧鐘時，需要先用二分查找法快速找到列表中合適的位置，然後再把Alarm對象插入此處。



如果所插入的位置正好是第0位，就說明此時新插入的這個邏輯鬧鐘將會是本類alarm中最先被激發的alarm，而正如我們前文所述，每一類邏輯鬧鐘會對應同一個“實體鬧鐘”，此處我們在第0位設置了新的激發時間，明確表示我們以前對“實體鬧鐘”設置的激發時間已經不准確了，所以setRepeating()中必須重新調整一下“實體鬧鐘”的激發時間，於是有了下面的句子：

```
if (index == 0) {
    setLocked(alarm);
}
```

setLocked()內部會調用native函數set()：

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

重新設置“實體鬧鐘”的激發時間。這個函數內部會調用ioctl()和底層打交道。具體代碼可參考frameworks/base/services/jni/com_android_server_AlarmManagerService.cpp文件：

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

我們知道，PendingIntent只是frameworks一層的概念，和底層驅動是沒有關係的。所以向底層設置alarm時只需要type信息以及激發時間信息就可以了。

3.2.3 取消alarm

用戶端是調用AlarmManager對象的cancel()函數來取消alarm的。這個函數內部其實是調用IAlarmManager的remove()函數。所以我們只來看AlarmManagerService的remove()就可以了。

```
public  void remove(PendingIntent operation)
{
    if (operation == null ) {
         return ;
    }
    synchronized (mLock) {
        removeLocked(operation);
    }
}
```

注意，在取消alarm時，是以一個PendingIntent對像作為參數的。這個PendingIntent對象正是當初設置alarm時，所傳入的那個operation參數。我們不能隨便創建一個新的PendingIntent對象來調用remove()函數，否則remove()是不會起作用的。PendingIntent的運作細節不在本文論述範圍之內，此處我們只需粗淺地知道，PendingIntent對像在AMS（Activity Manager Service）端會對應一個PendingIntentRecord實體，而ALMS在遍歷邏輯鬧鐘列表時，是根據是否指代相同PendingIntentRecord實體來判斷PendingIntent的相符情況的。如果我們隨便創建一個PendingIntent對象並傳入remove()函數的話，那麼在ALMS端勢必找不到相符的PendingIntent對象，所以remove()必然無效。

remove()中調用的removeLocked()如下：

```
public  void removeLocked(PendingIntent operation)
{
    removeLocked(mRtcWakeupAlarms, operation);
    removeLocked(mRtcAlarms, operation);
    removeLocked(mElapsedRealtimeWakeupAlarms, operation);
    removeLocked(mElapsedRealtimeAlarms, operation);
}
```

簡單地說就是，把4個邏輯鬧鐘數組都遍歷一遍，刪除其中所有和operation相符的Alarm節點。removeLocked()的實現代碼如下：

```
private  void removeLocked(ArrayList<Alarm> alarmList,
                           PendingIntent operation)
{
    if (alarmList.size() <= 0) {
         return ;
    }

    // iterator over the list removing any it where the intent match
    Iterator<Alarm> it = alarmList.iterator();

    while (it.hasNext()) {
        Alarm alarm = it.next();
        if (alarm.operation.equals(operation)) {
            it.remove();
        }
    }
}
```

請注意，所謂的取消alarm，只是刪除了對應的邏輯Alarm節點而已，並不會和底層驅動再打什麼交道。也就是說，是不存在針對底層“實體鬧鐘”的刪除動作的。所以，底層“實體鬧鐘”在到時之時，還是會被“激發”出來的，只不過此時在frameworks層，會因為找不到符合要求的“邏輯鬧鐘”而不做進一步的激發動作。

3.2.4 設置系統時間和時區

AlarmManager還提供設置系統時間的功能，設置者需要具有android.permission.SET_TIME權限。

```
public  void setTime( long millis)
{
    mContext.enforceCallingOrSelfPermission( "android.permission.SET_TIME" , "setTime" );
    SystemClock.setCurrentTimeMillis(millis);
}
```

另外，還具有設置時區的功能：

```
public  void setTimeZone(String tz)
```

相應地，設置者需要具有android.permission.SET_TIME_ZONE權限。

3.3 運作細節

3.3.1 AlarmThread和Alarm的激發

AlarmManagerService內部是如何感知底層激發alarm的呢？首先，AlarmManagerService有一個表示線程的mWaitThread成員：

```
private final AlarmThread mWaitThread = new AlarmThread();
```

在AlarmManagerService構造之初，就會啟動這個專門的“等待線程”。

```
public AlarmManagerService(Context context)
{
    mContext = context;
    mDescriptor = init();
    . . . . . .
    . . . . . .
    if (mDescriptor != -1)
    {
        mWaitThread.start();    //啟動線程！
    }
    else
    {
        Slog.w(TAG, "Failed to open alarm driver. Falling back to a handler." );
    }
}
```

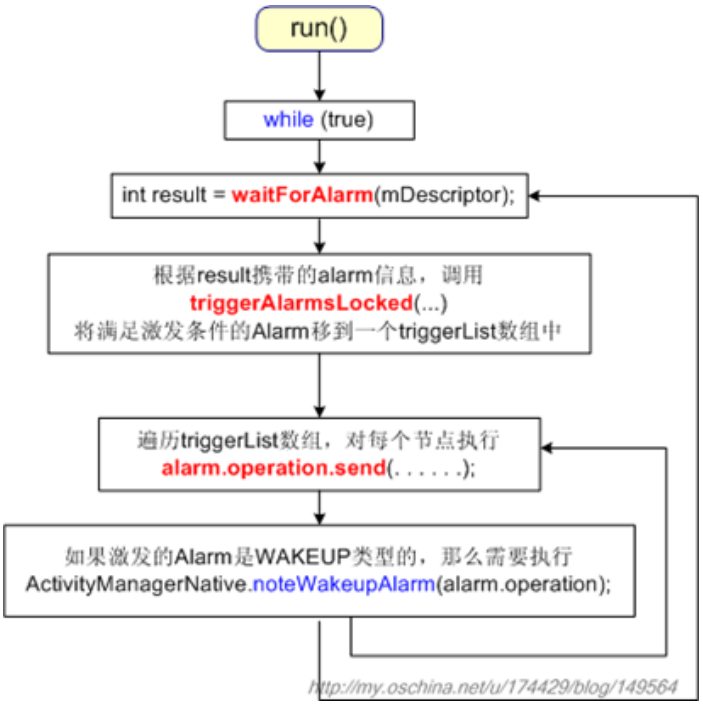
AlarmManagerService的構造函數一開始就會調用一個init()函數，該函數是個native函數，它的內部會打開alarm驅動，並返回驅動文件句柄。只要能夠順利打開alarm驅動，ALMS就可以走到mWaitThread.start()一句，於是“等待線程”就啟動了。

3.3.1.1 AlarmThread中的run()

AlarmThread本身是AlarmManagerService中一個繼承於Thread的內嵌類：

```
private  class AlarmThread extends Thread
```

其最核心的run()函數的主要動作流程圖如下：



我們分別來闡述上圖中的關鍵步驟。

3.3.1.2 waitForAlarm()

首先，從上文的流程圖中可以看到，AlarmThread線程是在一個while(true)循環裡不斷調用waitForAlarm()函數來等待底層alarm激發動作的。waitForAlarm()是一個native函數：

```
private native int waitForAlarm( int fd);
```

其對應的C++層函數是android_server_AlarmManagerService_waitForAlarm()：

【com_android_server_AlarmManagerService.cpp】

```
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)
    {
        ALOGE( "Unable to wait on alarm: %s\n" , strerror(errno));
         return 0;
    }

    return result;
}
```

當AlarmThread調用到ioctl()一句時，線程會阻塞住，直到底層激發alarm。而且所激發的alarm的類型會記錄到ioctl()的返回值中。這個返回值對外界來說非常重要，外界用它來判斷該遍歷哪個邏輯鬧鐘列表。

3.3.1.3 triggerAlarmsLocked()

一旦等到底層驅動的激發動作，AlarmThread會開始遍歷相應的邏輯鬧鐘列表：

```
ArrayList<Alarm> triggerList = new ArrayList<Alarm>();
. . . . . .
final long nowRTC = System.currentTimeMillis();
```

```
final long nowELAPSED = SystemClock.elapsedRealtime();
. . . . . .
if ((result & RTC_WAKEUP_MASK) != 0)
    triggerAlarmsLocked(mRtcWakeupAlarms, triggerList, nowRTC);
if ((result & RTC_MASK) != 0)
    triggerAlarmsLocked(mRtcAlarms, triggerList, nowRTC);
if ((result & ELAPSED_REALTIME_WAKEUP_MASK) != 0)
    triggerAlarmsLocked(mElapsedRealtimeWakeupAlarms, triggerList, nowELAPSED);
if ((result & ELAPSED_REALTIME_MASK) != 0)
    triggerAlarmsLocked(mElapsedRealtimeAlarms, triggerList, nowELAPSED);
```

可以看到，AlarmThread先創建了一個臨時的數組列表triggerList，然後根據result的值對相應的alarm數組列表調用triggerAlarmsLocked()，一旦發現alarm數組列表中有某個alarm符合激發條件，就把它移到triggerList中。這樣，4條alarm數組列表中需要激發的alarm就匯總到triggerList數組列表中了。

接下來，只需遍歷一遍triggerList就可以了：

```
Iterator<Alarm> it = triggerList.iterator();
 while (it.hasNext())
{
    Alarm alarm = it.next();
    . . . . . .
    alarm.operation.send(mContext, 0,
                         mBackgroundIntent.putExtra(Intent.EXTRA_ALARM_COUNT, alarm.count),
                         mResultReceiver, mHandler);

    // we have an active broadcast so stay awake.
    if (mBroadcastRefCount == 0) {
        setWakelockWorkSource(alarm.operation);
        mWakeLock.acquire();
    }
    mInFlight.add(alarm.operation);
    mBroadcastRefCount++;
    mTriggeredUids.add( new Integer(alarm.uid));
    BroadcastStats bs = getStatsLocked(alarm.operation);
    if (bs.nesting == 0) {
        bs.startTime = nowELAPSED;
    } else {
        bs.nesting++;
    }
    if (alarm.type == AlarmManager.ELAPSED_REALTIME_WAKEUP
        || alarm.type == AlarmManager.RTC_WAKEUP) {
        bs.numWakeup++;
        ActivityManagerNative.noteWakeupAlarm(alarm.operation);
    }
}
```

在上面的while循環中，每遍歷到一個Alarm對象，就執行它的alarm.operation.send()函數。我們知道，**alarm中記錄的operation就是當初設置它時傳來的那個PendingIntent對象**，現在開始執行PendingIntent的send()操作啦。

PendingIntent的send()函數代碼是：

```
public  void send(Context context, int code, Intent intent,
                  OnFinished onFinished, Handler handler) throws CanceledException
{
    send(context, code, intent, onFinished, handler, null );
}
```

調用了下面的send()函數：

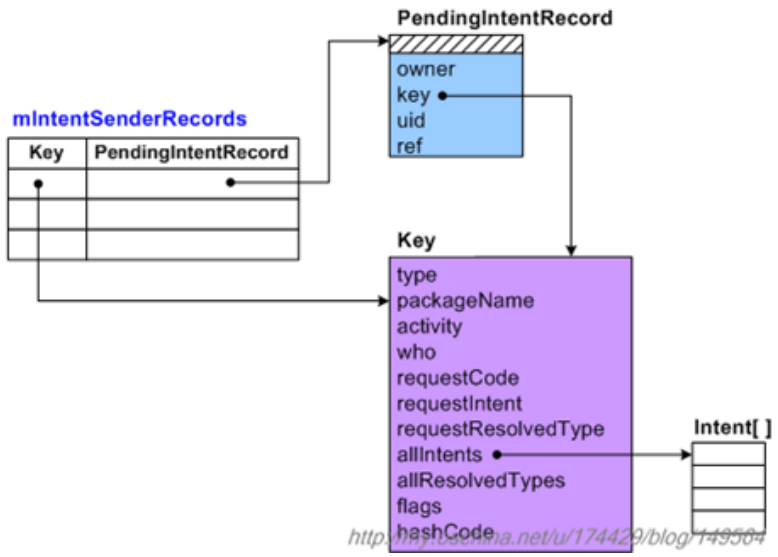
```
public  void send(Context context, int code, Intent intent,
                OnFinished onFinished, Handler handler, String requiredPermission)
                throws CanceledException
{
    try
    {
        String resolvedType = intent != null
                            ? intent.resolveTypeIfNeeded(context.getContentResolver())
                            : null ;
         int res = mTarget.send(code, intent, resolvedType,
                             onFinished != null
```

```
                                ? new FinishedDispatcher( this , onFinished, handler)
                                : null ,
                                requiredPermission);
            if (res < 0)
            {
                throw  new CanceledException();
            }
        }
        catch (RemoteException e)
        {
            throw  new CanceledException(e);
        }
    }
```

mTarget是個IPendingIntent代理接口，它對應AMS（Activity Manager Service）中的某個PendingIntentRecord實體。需要說明的是，PendingIntent的重要信息都是在AMS的PendingIntentRecord以及PendingIntentRecord.Key對像中管理的。AMS中有一張哈希表專門用於記錄所有可用的PendingIntentRecord對象。

相較起來，在創建PendingIntent對象時傳入的intent數組，其重要性並不太明顯。這種intent數組主要用於一次性啟動多個activity，如果你只是希望啟動一個activity或一個service，那麼這個intent的內容有可能在最終執行PendingIntent的send()動作時，被新傳入的intent內容替換掉。

AMS中關於PendingIntentRecord哈希表的示意圖如下：



AMS是整個Android平台中最複雜的一個核心service了，所以我們不在這裡做過多的闡述，有興趣的讀者可以參考其他相關文檔。

3.3.1.4 進一步處理“喚醒鬧鐘”

在AlarmThread.run()函數中while循環的最後，會進一步判斷，當前激發的alarm是不是“喚醒鬧鐘”。如果鬧鐘類型為RTC_WAKEUP或ELAPSED_REALTIME_WAKEUP，那它就屬於“喚醒鬧鐘”，此時需要通知一下AMS：

```
if (alarm.type == AlarmManager.ELAPSED_REALTIME_WAKEUP
    || alarm.type == AlarmManager.RTC_WAKEUP)
{
    bs.numWakeup++;
    ActivityManagerNative.noteWakeupAlarm(alarm.operation);
}
```

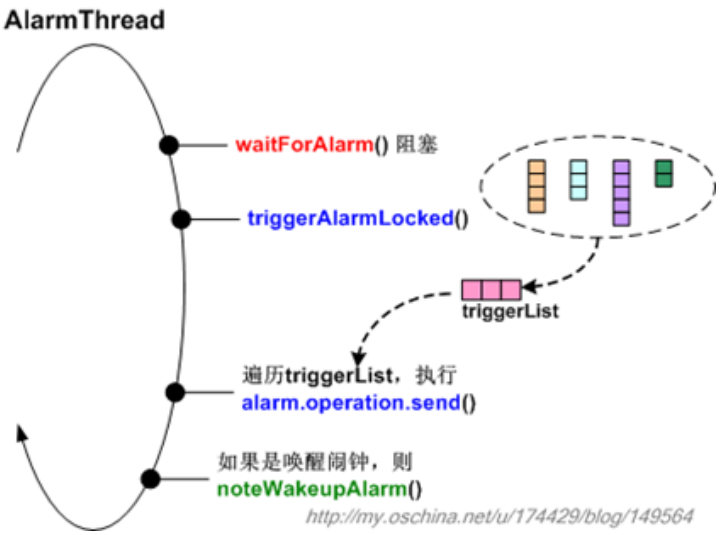
這兩種alarm就是我們常說的0型和2型鬧鐘，它們和我們手機的續航時間息息相關。

AMS裡的noteWakeupAlarm()比較簡單，只是在調用BatteryStatsService服務的相關動作，但是卻會導致機器的喚醒：

```
public  void noteWakeupAlarm(IIntentSender sender)
```

```
{
    if (!(sender instanceof PendingIntentRecord))
    {
        return ;
    }

    BatteryStatsImpl stats = mBatteryStatsService.getActiveStatistics();
    synchronized (stats)
    {
        if (mBatteryStatsService.isOnBattery())
        {
            mBatteryStatsService.enforceCallingPermission();
            PendingIntentRecord rec = (PendingIntentRecord)sender;
            int MY_UID = Binder.getCallingUid();
             int uid = rec.uid == MY_UID ? Process.SYSTEM_UID : rec.uid;
            BatteryStatsImpl.Uid.Pkg pkg = stats.getPackageStatsLocked(uid, rec.key.packageName);
            pkg.incWakeupsLocked();
        }
    }
}
```

好了，說了這麼多，我們還是畫一張AlarmThread示意圖作為總結：



3.3.2 說說AlarmManagerService中的mBroadcastRefCount

下面我們說說AlarmManagerService中的mBroadcastRefCount，之所以要說它，僅僅是因為我在修改AlarmManagerService代碼的時候，吃過它的虧。

我們先回顧一下處理triggerList列表的代碼，如下：

```
Iterator<Alarm> it = triggerList.iterator();
 while (it.hasNext())
{
    Alarm alarm = it.next();
    . . . . . .
    alarm.operation.send(mContext, 0,
                         mBackgroundIntent.putExtra(Intent.EXTRA_ALARM_COUNT, alarm.count),
                         mResultReceiver, mHandler);

    // we have an active broadcast so stay awake.
    if (mBroadcastRefCount == 0) {
        setWakelockWorkSource(alarm.operation);
        mWakeLock.acquire();
    }
    mInFlight.add(alarm.operation);
    mBroadcastRefCount++;
    . . . . . .
    . . . . . .
}
```

可以看到，在AlarmThread.run()中，只要triggerList中含有可激發的alarm，mBroadcastRefCount就會執行加一操作。一開始mBroadcastRefCount的值為0，所以會進入上面那句if語句，進而調用mWakeLock.acquire()。

後來我才知道，這個mBroadcastRefCount變量，是決定何時釋放mWakeLock的計數器。AlarmThread的意思很明確，只要還有處於激發狀態的邏輯鬧鐘，機器就不能完全睡眠。那麼釋放這個mWakeLock的地方又在哪裡呢？答案就在alarm.operation.send()一句的mResultReceiver參數中。

mResultReceiver是AlarmManagerService的私有成員變量：

```
private final ResultReceiver mResultReceiver = newResultReceiver();
```

類型為ResultReceiver，這個類實現了PendingIntent.OnFinished接口：

```
class ResultReceiver implements PendingIntent.OnFinished
```

當send()動作完成後，框架會間接回調這個對象的onSendFinished()成員函數。

```
public  void onSendFinished(PendingIntent pi, Intent intent, int resultCode,
                            String resultData, Bundle resultExtras)
{
    . . . . . .
    . . . . . .
    if (mBlockedUids.contains( new Integer(uid)))
    {
        mBlockedUids.remove( new Integer(uid));
    }
    else
    {
        if (mBroadcastRefCount > 0)
        {
            mInFlight.removeFirst();
            mBroadcastRefCount--;

            if (mBroadcastRefCount == 0)
            {
                mWakeLock.release();
            }
            . . . . . .
        }
        . . . . . .
    }
    . . . . . .
}
```

也就是說每當處理完一個alarm的send()動作，mBroadcastRefCount就會減一，一旦減為0，就釋放mWakeLock。

我一開始沒有足夠重視這個mBroadcastRefCount，所以把alarm.operation.send()語句包在了一條if語句中，也就是說在某種情況下，程序會跳過alarm.operation.send()一句，直接執行下面的語句。然而此時的mBroadcastRefCount還在堅定不移地加一，這直接導致mBroadcastRefCount再也減不到0了，於是mWakeLock也永遠不會釋放了。令人頭痛的是，這個mWakeLock雖然不讓手機深睡眠下去，卻也不會點亮屏幕，所以這個bug潛藏了好久才被找到。還真是應了我說的那句話：“魔鬼總藏在細節中。”

關鍵字： Android Alarm Manager

- 紅茶一杯話Binder（初始篇）»

共有2 條網友評論

-  1樓：李永明發表於2013-08-04 13:40 回复此評論
  謝謝分享
-  2樓：李永明發表於2013-08-04 13:42 回复此評論
  流程及架構圖畫得非常好，應該花了不少心思！

發表評論

文明上網，理性發言

回到頁首 | 回到評論列表

關閉相關文章閱讀

- 2012/12/29 Android Alarm manager定時鬧鐘開發...
- 2012/04/06 Android SDK Manager更新問題...
- 2013/08/28 解決Android Studio和Android SD...
- 2013/03/20 alarm_handler模塊
- 2013/01/26 Notification Manager...