Android (/tags/#Android) frameworks (/tags/#frameworks) AlarmManager (/tags/#AlarmManager)

AlarmManagerService之设置alarm流程

从代码角度理解alarm的设置流程

Posted by Cheson on February 14, 2017

1. 代码和平台版本

Android Version： 6.0 Platform： MTK MT6735/6737

2. 设置Alarm的demo代码

首先看下用set接口来设置一个alarm的流程，set接口设置的是不精准alarm，它的触发时间会被各种修改和优化.先来看下一个设置alarm的代码例子：

```
mAlarmManager = (AlarmManager)getSystemService(Context.ALARM_SERVICE);
intentSensing = new Intent(ACTION_ALARM_TRIGGER)
                .setPackage("com.cheson.alarmtest")
                .setFlags(Intent.FLAG_RECEIVER_REGISTERED_ONLY);
mAlarmIntent = PendingIntent.getBroadcast(getApplicationContext(), 0, intentSensing, 0);
```

```
private void scheduleAlarmLocked(long delay) {
        //Log.d(TAG, "scheduleAlarmLocked");
        cancelAlarmLocked();
        long nextTime = SystemClock.elapsedRealtime() + delay;
        mAlarmManager.set(AlarmManager.ELAPSED_REALTIME_WAKEUP,
                nextTime, mAlarmIntent);
    }
```

这里调用了AlarmManager的set接口，传入了三个参数，第一个为alarm的类型（alarm的类型有四种：RTC_WAKEUP、RTC、ELAPSED_REALTIME_WAKEUP和ELAPSED_REALTIME，可以参见AlarmManager中的定义）；第二个参数为触发时间；第三个参数已一个PendingIntent类型的变量，其含义是挂起的intent，作用是定义了一个alarm触发时的行为，行为可以是发广播，启动service和activity。

3. AlarmManager

来看下AlarmManager中对这个set接口的实现

```
public void set(int type, long triggerAtMillis, PendingIntent operation) {
    setImpl(type, triggerAtMillis, legacyExactLength(), 0, 0, operation, null, null);
}
```

可以看到是直接又调用了setImpl方法

```
private void setImpl(int type, long triggerAtMillis, long windowMillis, long intervalMillis, int flags, PendingIntent operation, Work
        if (triggerAtMillis < 0) {
            /* NOTYET
            if (mAlwaysExact) {
                // Fatal error for KLP+ apps to use negative trigger times
                throw new IllegalArgumentException("Invalid alarm trigger time "
                        + triggerAtMillis);
            }
            */
            triggerAtMillis = 0;
        }

        try {
            mService.set(type, triggerAtMillis, windowMillis, intervalMillis, flags, operation,
                    workSource, alarmClock);
        } catch (RemoteException ex) {
        }
    }
```

这里需要传入八个参数，第一个就是alarm的类型；第二个是触发时间；第三个参数标示是否精准闹钟（set接口设置的为非精准闹钟），这里是通过legacyExactLength()方法来返回的，而方法内是通过SDK的等级来判断，小于KK的就返回WINDOW_EXACT也就是精准闹钟，否则返回WINDOW_HEURISTIC非精准闹钟，设计的目的是在KK版本后降低功耗；第四个参数intervalMillis为重复闹钟的间隔时间，这里是一次性Alarm所以直接传入的是0；第五个参数flags是一个标志，这里定义了四种，FLAG_STANDALONE = 1«0（独立的alarm，不允许被调剂到一批alarm中统一触发），FLAG_WAKE_FROM_IDLE = 1«1（idle模式下也能唤醒设备），FLAG_ALLOW_WHILE_IDLE = 1«2（idle模式下也可以执行，但不会使设备退出idle模式），FLAG_ALLOW_WHILE_IDLE_UNRESTRICTED = 1«3（和前者功能类似，但没有限制安排的频率）；第六个参数就是PendingIntent；第七个参数为WorkSource，这里为null；第八个参数为AlarmClockInfo，这里为null。而后是通过binder将条码调用跳到了AlarmManagerService中去实现。

4. AlarmManagerService

4.1 set

代码通过binder调用到了AlarmManagerService中的set方法，这里都是在对flag进行处理加工:

- 不允许所有的调用者使用WAKE_FROM_IDLE和ALLOW_WHILE_IDLE_UNRESTRICTED

```
// No incoming callers can request either WAKE_FROM_IDLE or
// ALLOW_WHILE_IDLE_UNRESTRICTED -- we will apply those later as appropriate.
flags &= ~(AlarmManager.FLAG_WAKE_FROM_IDLE
    | AlarmManager.FLAG_ALLOW_WHILE_IDLE_UNRESTRICTED);
```

- 如果不是System进程，则不允许使用FLAG_IDLE_UNTIL

```
// Only the system can use FLAG_IDLE_UNTIL -- this is used to tell the alarm
// manager when to come out of idle mode, which is only for DeviceIdleController.
if (callingUid != Process.SYSTEM_UID) {
    flags &= ~AlarmManager.FLAG_IDLE_UNTIL;
}
```

- 如果是system核心模块并且workSource为null，则添加FLAG_ALLOW_WHILE_IDLE_UNRESTRICTED标记，允许在idle模式下执行

```
// If the caller is a core system component, and not calling to do work on behalf
// of someone else, then always set ALLOW_WHILE_IDLE_UNRESTRICTED.  This means we
// will allow these alarms to go off as normal even while idle, with no timing
// restrictions.
if (callingUid < Process.FIRST_APPLICATION_UID && workSource == null) {
    flags |= AlarmManager.FLAG_ALLOW_WHILE_IDLE_UNRESTRICTED;
}
```

- 如果设置的是精准alarm，则添加FLAG_STANDALONE标记，使其不允许被重新安排

```
// If this is an exact time alarm, then it can't be batched with other alarms.
if (windowLength == AlarmManager.WINDOW_EXACT) {
    flags |= AlarmManager.FLAG_STANDALONE;
}
```

- 如果该alarm是个闹钟，则添加FLAG_WAKE_FROM_IDLE和FLAG_STANDALONE标记，使其为精准闹钟，并且可以在idle时唤醒

```
// If this alarm is for an alarm clock, then it must be standalone and we will
// use it to wake early from idle if needed.
if (alarmClock != null) {
    flags |= AlarmManager.FLAG_WAKE_FROM_IDLE | AlarmManager.FLAG_STANDALONE;
}
```

之后再调用setImpl进行下一步处理，参数部分增加了binder调用者的uid。

```
setImpl(type, triggerAtTime, windowLength, interval, operation,
    flags, workSource, alarmClock, callingUid);
```

4.2 setImpl

来看下setImpl方法里都做了些什么

```
// 无操作alarm，无意义直接return
if (operation == null) {
    Slog.w(TAG, "set/setRepeating ignored because there is no intent");
        return;
}
```

```
// MTK的快速关机流程，忽略alarm
// /M:add for IPO,when shut down,do not set alarm to driver ,@{
if (mIPOShutdown && (mNativeData == -1)) {
    Slog.w(TAG, "IPO Shutdown so drop the alarm");
    return;
}
// /@}
```

```
// 检查设置的window length若超过半天则重设为1小时
// Sanity check the window length.  This will catch people mistakenly
// trying to pass an end-of-window timestamp rather than a duration.
if (windowLength > AlarmManager.INTERVAL_HALF_DAY) {
   Slog.w(TAG, "Window length " + windowLength
        + "ms suspiciously long; limiting to 1 hour");
    windowLength = AlarmManager.INTERVAL_HOUR;
}
```

```
// Sanity check the recurrence interval.  This will catch people who supply
// seconds when the API expects milliseconds.
final long minInterval = mConstants.MIN_INTERVAL;//重复闹钟最小间隔，默认1分钟
if (interval > 0 && interval < minInterval) {//小于最小间隔时，重设为最小间隔
    Slog.w(TAG, "Suspiciously short interval " + interval
        + " millis; expanding to " + (minInterval/1000)
        + " seconds");
    interval = minInterval;
}
```

```
if (triggerAtTime < 0) {//触发时间无效
    final long what = Binder.getCallingPid();
    Slog.w(TAG, "Invalid alarm trigger time! " + triggerAtTime + " from uid=" + callingUid
        + " pid=" + what);
    triggerAtTime = 0;
}
```

进行过一些前期处理之后，接下是对触发时间的处理

```
// 对触发时间的预处理
final long nowElapsed = SystemClock.elapsedRealtime();//当前系统开机时间
// RTC类型alarm时，计算nominal时间
final long nominalTrigger = convertToElapsed(triggerAtTime, type);
// Try to prevent spamming by making sure we aren't firing alarms in the immediate future
// 确保alarm不会立即触发，防止频繁的垃圾alarm（安全考虑）
final long minTrigger = nowElapsed + mConstants.MIN_FUTURITY;
final long triggerElapsed = (nominalTrigger > minTrigger) ? nominalTrigger : minTrigger;
```

触发时间计算完成之后紧接着计算了alarm的最大延时触发，也就是说非精准alarm可以在最大延时范围内触发，这个就是Android对Alarm功耗相关的一个处理方法。

```
long maxElapsed;
if (mSupportAlarmGrouping && (mAmPlus != null)) {
    // MTK平台开启背景省电功能时mmSupportAlarmGrouping为true
    // M: BG powerSaving feature
    // MTK封装了AlarmManagerPlus，没有源码.
    // 开启MTK的alarm group功能后，alarm的触发时间会被统一延长，增加5分钟给maxElapsed
    // 从log中看当maxElapsed算出来为正数时，延长的时间确实为5分钟，什么时候计算为正什么时候为负？
    maxElapsed = mAmPlus.getMaxTriggerTime(type, triggerElapsed, windowLength, interval, operation, mAlarmMode, true);
    if (maxElapsed < 0) {
        maxElapsed = 0 - maxElapsed;
        mNeedGrouping = false;
    } else {
        mNeedGrouping = true;
        //isStandalone = false; //ALPS02190343
    }
} else if (windowLength == AlarmManager.WINDOW_EXACT) {
    maxElapsed = triggerElapsed;//精准闹钟不延时
} else if (windowLength < 0) {
    // google原始流程中对非精准闹钟对延时触发时间的处理
    // 如果是一次性闹钟，触发时间减去当前时间小于10s的,maxElapsed即返回triggerElapsed
    // 如果是重复闹钟，间隔时间小于10s的，maxElapsed即返回triggerElapsed
    // 否则maxElapsed的计算方法为triggerAtTime + (long)(.75 * futurity)
    // 这里的futurity为一次性闹钟的触发时间减去当前时间或者重复闹钟的间隔时间
    maxElapsed = maxTriggerTime(nowElapsed, triggerElapsed, interval);
    // Fix this window in place, so that as time approaches we don't collapse it.
    windowLength = maxElapsed - triggerElapsed;
} else {
    maxElapsed = triggerElapsed + windowLength;
}
```

一系列的数据处理完成之后setImpl完成了使命，调用setImplLocked来完成最后的工作

```
synchronized (mLock) {
  if (false) {
    Slog.v(TAG, "APP set(" + operation + ") : type=" + type
        + " triggerAtTime=" + triggerAtTime + " win=" + windowLength
        + " tElapsed=" + triggerElapsed + " maxElapsed=" + maxElapsed
        + " interval=" + interval + " flags=0x" + Integer.toHexString(flags));
    }
    setImplLocked(type, triggerAtTime, triggerElapsed, windowLength, maxElapsed,
        interval, operation, flags, true, workSource,
        alarmClock, callingUid, mNeedGrouping);
}
```

4.3 setImplLocked

首先会有一段中转处理的代码，新建了一个Alarm的对象，将属性值都封装到这个对象中，然后通过重载调用了三个参数的setImplLocked方法。

```
private void setImplLocked(int type, long when, long whenElapsed, long windowLength,
        long maxWhen, long interval, PendingIntent operation, int flags,
        boolean doValidate, WorkSource workSource, AlarmManager.AlarmClockInfo alarmClock,
        int uid, boolean mNeedGrouping) {
    Alarm a = new Alarm(type, when, whenElapsed, windowLength, maxWhen, interval,
            operation, workSource, flags, alarmClock, uid, mNeedGrouping);
    removeLocked(operation);
    setImplLocked(a, false, doValidate);
}
```

跳过前面一些对特殊类型alarm的处理代码，来看一般的alarm处理，做的最重要的一件事就是将alarm分批(batch)，核心的代码如下：

```
// 计算alarm的批次
int whichBatch = 0;
if (mSupportAlarmGrouping && (mAmPlus != null)) {// MTK alarm group流程
    // M using a.needGrouping for check condition
    // a.needGrouping is false -> run default flow
    // a.needGrouping is true  -> run find batch flow
    if (a.needGrouping == false) {// 如果alarm不需要group
        // 如果是STANDALONE类型的alarm，whichBatch则为-1，否则计算whichBatch
        whichBatch = ((a.flags & AlarmManager.FLAG_STANDALONE) != 0)
            ? -1 : attemptCoalesceLocked(a.whenElapsed, a.maxWhenElapsed);
    } else {
        // 这边代码写的冗余了，其实不需要对needGrouping进行判断，在非STANDALONE类型alarm时
        // 都要对whichBatch进行计算
        whichBatch = attemptCoalesceLocked(a.whenElapsed, a.maxWhenElapsed);
    }
} else {
    // google原始流程，逻辑同上
    // 其实这部分代码都可以合并，可能是MTK为了区别处理吧
    Slog.d(TAG, "default path for whichBatch");
    whichBatch = ((a.flags & AlarmManager.FLAG_STANDALONE) != 0)
        ? -1 : attemptCoalesceLocked(a.whenElapsed, a.maxWhenElapsed);
}
```

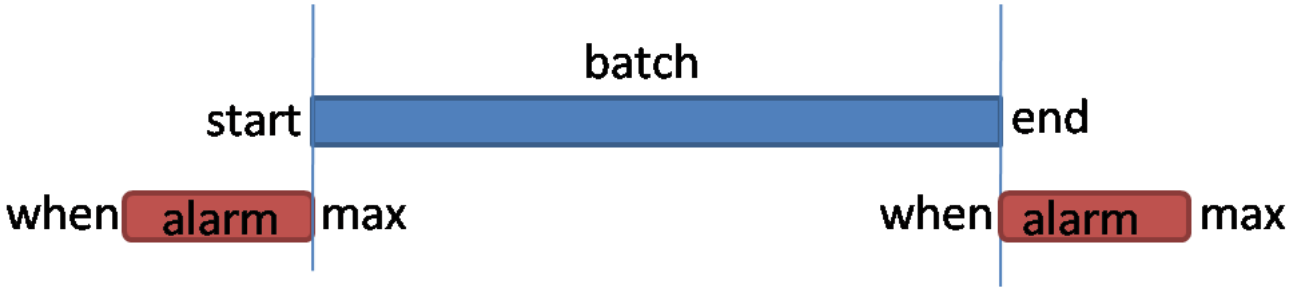
这里多次出现最核心的逻辑就是通过attemptCoalesceLocked方法来计算所属的批次，这个函数的代码如下：

```
// Return the index of the matching batch, or -1 if none found.
int attemptCoalesceLocked(long whenElapsed, long maxWhen) {
    final int N = mAlarmBatches.size();// 有多个alarm的batch
    for (int i = 0; i < N; i++) {
        Batch b = mAlarmBatches.get(i);
        if (mSupportAlarmGrouping && (mAmPlus != null)) {
            // M mark b.flags for check condition
            if (b.canHold(whenElapsed, maxWhen)) {// 当一个batch能容纳这个alarm时
                return i;
            }
        } else {
            if ((b.flags & AlarmManager.FLAG_STANDALONE) == 0
                    && b.canHold(whenElapsed, maxWhen)) {
                // google原始代码里没有多加这一层判断
                // MTK为了修复什么bug？
                if (b.canHold(whenElapsed, maxWhen)) {
                    return i;
                }
            }
        }
    }
    return -2;//MTK为什么改成-2了？
}
```

发现MTK把代码改的有点莫名其妙，先不管，来看重点，就是如何判断batch是否能容纳这个alarm代码逻辑是这样的：

```
boolean canHold(long whenElapsed, long maxWhen) {
    return (newEnd >= whenElapsed) && (start <= maxWhen);
}
```

是判断batch的起始时间是否包含了alarm的触发时间和前面计算得到的最大延迟时间，用一张图来描述如下：



在这两个alarm的区间范围内的alarm就可以算在这个batch的范围之内，也就是说alarm的触发时间要在batch的结束时间之前，并且最大延迟时间要能够得上batch的开始时间就满足条件了。然后开始安排alarm到batch中，或新建或插入。

```
// 开始分批
Slog.d(TAG, " whichBatch = " + whichBatch);
if (whichBatch < 0) {// 为这个alarm新建一个batch
    Batch batch = new Batch(a);
    // 将batch添加到列表中
    addBatchLocked(mAlarmBatches, batch);
} else {
    Batch batch = mAlarmBatches.get(whichBatch);
    Slog.d(TAG, " alarm = " + a + " add to " + batch);
    if (batch.add(a)) {// 添加alarm到batch中，返回值为start的值是否改变
        // The start time of this batch advanced, so batch ordering may
        // have just been broken.  Move it to where it now belongs.
        // 这个batch的开始时间增加了，需要重排它在列表中的次序
        mAlarmBatches.remove(whichBatch);
        addBatchLocked(mAlarmBatches, batch);
    }
}
```

一个常规的alarm添加流程就这样。

5. dumpsys alarm

用adb shell dumpsys alarm命令可以看到系统alarm的信息，包括了有多少个batch，每个batch的start和end时间，batch中包括了几个alarm，还有top10的alarm信息等。详细的可以参考这篇资料：dumpsys alarm 格式解读 (http://blog.csdn.net/memoryjs/article/details/48709183)

6. 功耗优化思路

进一步做Alarm相关功耗优化时可以有两个思路：

1. 对不精准alarm，可以在原先的基础上再增加maxElapsed的时间，如此可以增大alarm被分到batch中去概率
2. 一个batch中拥有IM应用的alarm时，增大这个batch的end time，如此可以增加这个batch容纳alarm的能力。

PREVIOUS
ANDROID中ALARMMANAGER使用指南 **(/2017/02/14/ALARMMANAGER-GUIDE/)**

NEXT
ANDROID性能优化之BOOT PERFORMANCE **(/2017/02/16/BOOT_PERFORMANCE/)**

FEATURED TAGS (/tags/)

前端 (/tags/#前端) Android (/tags/#Android) frameworks (/tags/#frameworks) AlarmManager (/tags/#AlarmManager) Performance (/tags/#Performance) systrace (/tags/#systrace) PowerManager (/tags/#PowerManager) Wakelock (/tags/#Wakelock) Guitar (/tags/#Guitar) 民谣 (/tags/#民谣) 赵雷 (/tags/#赵雷) Doze (/tags/#Doze) Android Performance Patterns (/tags/#Android Performance Patterns)

FRIENDS

待遇见志同道合的你 (https://github.com) 小明 (http://www.betterming.cn)

(https://twitter.com/chendongqi) (https://www.zhihu.com/people/chendongqi)
(http://weibo.com/chendongqi) (https://www.facebook.com/chendongqi)
(https://github.com/chendongqi) (https://www.linkedin.com/in/firstname-lastname-idxxxx)