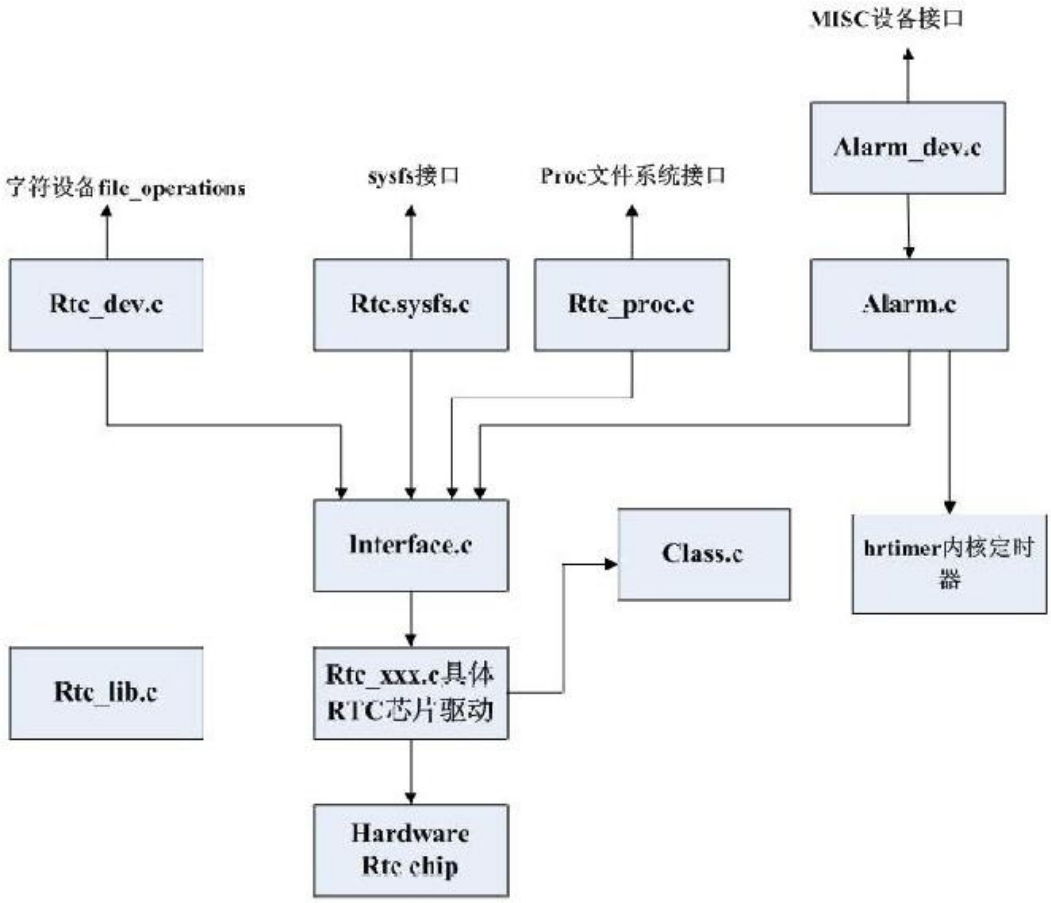
第十一章 Android 内核驱动——Alarm

11.1 基本原理

Alarm 闹钟是 android 系统中在标准 RTC 驱动上开发的一个新的驱动，提供了一个定时器用于把设备从睡眠状态唤醒，当然因为它是依赖 RTC 驱动的，所以它同时还可以为系统提供一个掉电下还能运行的实时时钟。

当系统断电时，主板上的 rtc 芯片将继续维持系统的时间，这样保证再次开机后系统的时间不会错误。当系统开始时，内核从 RTC 中读取时间来初始化系统时间，关机时便又将系统时间写回到 rtc 中，关机阶段将有主板上另外的电池来供应 rtc 计时。Android 中的 Alarm 在设备处于睡眠模式时仍保持活跃，它可以设置来唤醒设备。



上图为 android 系统中 alarm 和 rtc 驱动的框架。Alarm 依赖于 rtc 驱动框架，但它不是一个 rtc 驱动，主要还是实现定时闹钟的功能。相关源代码在 kernel/drivers/rtc/alarm.c 和 drivers/rtc/alarm_dev.c。

其中 alarm.c 文件实现的是所有 alarm 设备的通用性操作，它创建了一个设备 class，而

alarm_dev.c 则创建具体的 alarm 设备，注册到该设备 class 中。alarm.c 还实现了与 interface.c 的接口，即建立了与具体 rtc 驱动和 rtc 芯片的联系。alarm_dev.c 在 alarm.c 基础包装了一层，主要是实现了标准的 miscdevice 接口，提供给应用层调用。

可以这样概括：alarm.c 实现的是机制和框架，alarm_dev.c 则是实现符合这个框架的设备驱动，alarm_dev.c 相当于在底层硬件 rtc 闹钟功能的基础上虚拟了多个软件闹钟。

11.2 关键数据结构

- **alarm**

  定义在 include/linux/android_alarm.h 中。

```
struct alarm {
     struct rb_node           node;
     enum android_alarm_type  type;
     ktime_t                  softexpires; //最早的到期时间
     ktime_t                  expires; //绝对到期时间
     void (*function)(struct alarm *); //当到期时系统回调该函数
};
```

  这个结构体代表 alarm 设备，所有的 alarm 设备按照它们过期时间的先后被组织成一个红黑树，alarm.node 即红黑树的节点，alarm 设备通过这个变量插入红黑树。alarm.type 是类型，android 中一共定义了如下 5 种类型，在现在的系统中每种类型只有一个设备。

```
enum android_alarm_type {
     /* return code bit numbers or set alarm arg */
     ANDROID_ALARM_RTC_WAKEUP,
     ANDROID_ALARM_RTC,
     ANDROID_ALARM_ELAPSED_REALTIME_WAKEUP,
     ANDROID_ALARM_ELAPSED_REALTIME,
     ANDROID_ALARM_SYSTEMTIME,
     ANDROID_ALARM_TYPE_COUNT,
     /* return code bit numbers */
     /* ANDROID_ALARM_TIME_CHANGE = 16 */
};
```

- **alarm_queue**

```
struct alarm_queue {
     struct rb_root    alarms; //红黑树的根
     struct rb_node   *first; //指向第一个 alarm device,即最早到时的
     struct hrtimer    timer; //内核定时器,android 利用它来确定 alarm 过期时间
     ktime_t           delta; //是一个计算 elasped realtime 的修正值
     bool              stopped;
     ktime_t           stopped_time;
};
```

  这个结构体用于将前面的 struct alarm 表示的设备组织成红黑树。它是基于内核定时器来实现 alarm 的到期闹铃的。

11.3 关键代码分析

- **alarm_dev.c**

该文件依赖于 alarm.c 提供的框架，实现了与应用层交互的功能，具体说就是暴露出 miscdevice 的设备接口。Alarm_dev.c 定义了几个全局变量：

每种类型一个 alarm 设备，android 目前创建了 5 个 alarm 设备。

```
static struct alarm alarms[ANDROID_ALARM_TYPE_COUNT];
```

wake lock 锁，当加锁时，阻止系统进 suspend 状态。

```
static struct wake_lock alarm_wake_lock;
```

标志位，alarm 设备是否被打开。

```
static int alarm_opened;
```

标志位，alarm 设备是否就绪。所谓就绪是指该 alarm 设备的闹铃时间到达，但原本等待在该 alarm 设备上的进程还未唤醒，一旦唤醒，该标志清零。

```
static uint32_t alarm_pending;
```

标志位，表示 alarm 设备是否 enabled，表示该设备设置了闹铃时间（并且闹铃时间还未到），一旦闹铃时间到了，该标志清零。

```
static uint32_t alarm_enabled;
```

标志位，表示原先等待该 alarm 的进程被唤醒了（它们等待的 alarm 到时了）。

```
static uint32_t wait_pending;
```

该文件提供的主要函数有:

1，模块初始化和 exit 函数：alarm_dev_init 和 alarm_dev_exit

2，模块 miscdevice 标准接口函数：alarm_open、alarm_release 和 alarm_ioctl

3， alarm 定时时间到时候的回调函数：alarm_triggered

alarm_dev_init 初始化函数调用 misc_register 注册一个 miscdevice。

```
static int __init alarm_dev_init(void){
    int err;
    int i;

    err = misc_register(&alarm_device);
    if (err)
        return err;

    for (i = 0; i < ANDROID_ALARM_TYPE_COUNT; i++)
        alarm_init(&alarms[i], i, alarm_triggered);
    wake_lock_init(&alarm_wake_lock, WAKE_LOCK_SUSPEND, "alarm");

    return 0;
}
```

该设备称为 alarm_device，定义如下:

```
static struct miscdevice alarm_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = "alarm",
    .fops = &alarm_fops,
};
```

对应的 file operations 为 alarm_fops，定义为:

```
static const struct file_operations alarm_fops = {
    .owner = THIS_MODULE,
    .unlocked_ioctl = alarm_ioctl,
    .open = alarm_open,
    .release = alarm_release,
};
```

然后为每个 alarm device 调用 alarm_init 初始化，这个函数代码在 alarm.c 中，如下：

```
void alarm_init(struct alarm *alarm,
                enum android_alarm_type type,
                void (*function)(struct alarm *)){
    RB_CLEAR_NODE(&alarm->node);
    alarm->type = type;
    alarm->function = function;
    pr_alarm(FLOW, "created alarm, type %d, func %pF\n", type, function);
}
```

就是初始化 alarm 结构体，设置其回调函数为 alarm_triggered。最后调用 wake_lock_init 初始化 alarm_wake_lock，它是 suspend 型的。alarm_triggered 是回调函数，当定时闹铃的时间到了，alarm_timer_triggered 函数会调用该函数（详细请看 alarm.c 的 alarm_timer_triggered 函数）。

```
static void alarm_triggered(struct alarm *alarm){
    unsigned long flags;
    uint32_t alarm_type_mask = 1U << alarm->type;

    pr_alarm(INT, "alarm_triggered type %d\n", alarm->type);
    spin_lock_irqsave(&alarm_slock, flags);
    if (alarm_enabled & alarm_type_mask) {
        wake_lock_timeout(&alarm_wake_lock, 5 * HZ);
        alarm_enabled &= ~alarm_type_mask;
        alarm_pending |= alarm_type_mask;
        wake_up(&alarm_wait_queue);
    }
    spin_unlock_irqrestore(&alarm_slock, flags);
}
```

这个函数里调用 wake_lock_timeout 对全局 alarm_wake_lock（超时锁，超时时间是 5 秒）加锁，禁止对应的 alarm 设备。唤醒所有等待在该 alarm 设备上的进程。这时，如果 AP 层呼叫 ioctl(fd， ANDROID_ALARM_WAIT)，会返回表示等到 alarm 的返回值（这个会在 AlarmManagerSevice.java 中细述）。

alarm_ioctl 定义了以下命令:

ANDROID_ALARM_CLEAR	清除 alarm，即 deactivate 这个 alarm
ANDROID_ALARM_SET_OLD	设置 alarm 闹铃时间
ANDROID_ALARM_SET	同上
ANDROID_ALARM_SET_AND_WAIT_OLD	设置 alarm 闹铃时间并等待这个 alarm
ANDROID_ALARM_SET_AND_WAIT	同上
ANDROID_ALARM_WAIT	等待 alarm
ANDROID_ALARM_SET_RTC	设置 RTC 时间
ANDROID_ALARM_GET_TIME	读取 alarm 时间，根据 alarm 类型又分四种情况

ANDROID_ALARM_RTC_WAKEUP ANDROID_ALARM_RTC	读取系统当前时间，即从1970-1-1-00：00：00至今过去多少秒，精确到us
ANDROID_ALARM_ELAPSED_REALTIME_WAKEUP ANDROID_ALARM_ELAPSED_REALTIME	读取 elapsed time
ANDROID_ALARM_SYSTEMTIME ANDROID_ALARM_TYPE_COUNT	读取 system time，代码注释说是所谓的 monotonic clock，就是 ANDROID_ALARM_RTC 的时间做了个修正

- alarm.c

该文件完成主要功能有:

- 创建一个 alarm class，所有 alarm 设备都属于这个类；
- 注册了 platform driver，提供 suspend 和 resume 支持；
- 实现了一系列函数，包括 alarm_init，alarm_start_range，alarm_cancel，alarm_timer_triggered 函数等。

Alarm.c 的初始化函数 alarm_driver_init 如下：

```
static int __init alarm_driver_init(void){
    int err;
    int i;

    for (i = 0; i < ANDROID_ALARM_SYSTEMTIME; i++) {
        hrtimer_init(&alarms[i].timer, CLOCK_REALTIME, HRTIMER_MODE_ABS);
        alarms[i].timer.function = alarm_timer_triggered;
    }
    hrtimer_init(&alarms[ANDROID_ALARM_SYSTEMTIME].timer,
            CLOCK_MONOTONIC, HRTIMER_MODE_ABS);
    alarms[ANDROID_ALARM_SYSTEMTIME].timer.function = alarm_timer_triggered;

    err = platform_driver_register(&alarm_driver);
    if (err < 0)
        goto err1;
    wake_lock_init(&alarm_rtc_wake_lock, WAKE_LOCK_SUSPEND, "alarm_rtc");
    rtc_alarm_interface.class = rtc_class;
    err = class_interface_register(&rtc_alarm_interface);
    if (err < 0)
        goto err2;

    return 0;

err2:
    wake_lock_destroy(&alarm_rtc_wake_lock);
    platform_driver_unregister(&alarm_driver);
err1:
    return err;
}
```

该函数初始化 5 个 alarm device 相关联的 hrtimer 定时器，设置 hrtimer 定时器的回调函数为 alarm_timer_triggered 函数，再注册一个 plateform driver 和 class interface。 如果设置了闹铃时间，则内核通过 hrtimer 定时器来跟踪是否到时间，到时后会触发调用 hrtimer 的处理函数 alarm_timer_triggered。alarm_timer_triggered 的 code 如下：

```
static enum hrtimer_restart alarm_timer_triggered(struct hrtimer *timer){
    struct alarm_queue *base;
    struct alarm *alarm;
    unsigned long flags;
    ktime_t now;

    spin_lock_irqsave(&alarm_slock, flags);

    base = container_of(timer, struct alarm_queue, timer);
    now = base->stopped ? base->stopped_time : hrtimer_cb_get_time(timer);
    now = ktime_sub(now, base->delta);

    pr_alarm(INT, "alarm_timer_triggered type %d at %lld\n",
        base - alarms, ktime_to_ns(now));

    while (base->first) {
        alarm = container_of(base->first, struct alarm, node);
        if (alarm->softexpires.tv64 > now.tv64) {
            pr_alarm(FLOW, "don't call alarm, %pF, %lld (s %lld)\n",
                alarm->function, ktime_to_ns(alarm->expires),
                ktime_to_ns(alarm->softexpires));
            break;
        }
        base->first = rb_next(&alarm->node);
        rb_erase(&alarm->node, &base->alarms);
        RB_CLEAR_NODE(&alarm->node);
        pr_alarm(CALL, "call alarm, type %d, func %pF, %lld (s %lld)\n",
            alarm->type, alarm->function,
            ktime_to_ns(alarm->expires),
            ktime_to_ns(alarm->softexpires));
        spin_unlock_irqrestore(&alarm_slock, flags);
        alarm->function(alarm);
        spin_lock_irqsave(&alarm_slock, flags);
    }
    if (!base->first)
        pr_alarm(FLOW, "no more alarms of type %d\n", base - alarms);
    update_timer_locked(base, true);
    spin_unlock_irqrestore(&alarm_slock, flags);
    return HRTIMER_NORESTART;
}
```

它会轮询红黑树中的所有 alarm 节点，符合条件的节点会执行 alarm.function(alarm)，指向 alarm_dev.c 的 alarm_triggered 函数。因为我们在执行 alarm_dev.c 的 alarm_init 时，把每个 alarm 节点的 function 设置成了 alarm_triggered。

请注意这两个函数的区别，alarm_triggered 和 alarm_timer_triggered。前者是 rtc 芯片的 alarm 中断的回调函数，后者是 android alarm_queue->timer 到时的回调函数。

上面说到，alarm_driver_init 注册了一个类接口

```
class_interface_register(&rtc_alarm_interface)
```

rtc_alarm_interface 的 code 如下：

```
static struct class_interface rtc_alarm_interface = {
    .add_dev = &rtc_alarm_add_device,
    .remove_dev = &rtc_alarm_remove_device,
};
```

在 rtc_alarm_add_device 中，注册了一个 rtc 中断 rtc_irq_register(rtc，&alarm_rtc_task)

```
static int rtc_alarm_add_device(struct device *dev,
                                struct class_interface *class_intf){
    int err;
    struct rtc_device *rtc = to_rtc_device(dev);

    mutex_lock(&alarm_setrtc_mutex);

    if (alarm_rtc_dev) {
        err = -EBUSY;
        goto err1;
    }

    alarm_platform_dev = platform_device_register_simple("alarm", -1, NULL, 0);
    if (IS_ERR(alarm_platform_dev)) {
        err = PTR_ERR(alarm_platform_dev);
        goto err2;
    }
    err = rtc_irq_register(rtc, &alarm_rtc_task);
    if (err)
        goto err3;
    alarm_rtc_dev = rtc;
    pr_alarm(INIT_STATUS, "using rtc device, %s, for alarms", rtc->name);
    mutex_unlock(&alarm_setrtc_mutex);

    return 0;

err3:
    platform_device_unregister(alarm_platform_dev);
err2:
err1:
    mutex_unlock(&alarm_setrtc_mutex);
    return err;
}
```

中断的回调函数为 alarm_triggered_func：

```
static struct rtc_task alarm_rtc_task = {
    .func = alarm_triggered_func
};

static void alarm_triggered_func(void *p){
    struct rtc_device *rtc = alarm_rtc_dev;
    if (!(rtc->irq_data & RTC_AF))
        return;
    pr_alarm(INT, "rtc alarm triggered\n");
    wake_lock_timeout(&alarm_rtc_wake_lock, 1 * HZ);
}
```

当硬件 rtc chip 的 alarm 中断发生时，系统会调用 alarm_triggered_func 函数。alarm_triggered_func 的功能很简单，wake_lock_timeout 锁住 alarm_rtc_wake_lock 1 秒。因为这时，alarm 会进入 alarm_resume，lock 住 alarm_rtc_wake_lock 以防止 alarm 在此时进入 suspend。

- AlarmManager.java，该文件提供的接口主要有:
  - 设置闹钟

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

  - 设置周期闹钟。

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

  - 取消闹钟

```
public void cancel(PendingIntent operation);
```

上面 3 个函数分别会呼叫到 AlarmManagerSevice.java 以下三个函数：

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

AlarmManagerSevice.java 通过 JNI 机制可以呼叫 com_android_server_AlarmManagerService.cpp 透出的几个接口。

- AlarmManagerSevice.java 有关接口的 code 如下：

```
private native int init();
private native void close(int fd);
private native void set(int fd, int type, long seconds, long nanoseconds);
private native int waitForAlarm(int fd);
private native int setKernelTimezone(int fd, int minuteswest);
```

- com_android_server_AlarmManagerService.cpp 有关接口的对应 code 如下：

```
static JNINativeMethod sMethods[] = {
  /* name, signature, funcPtr */
{"init", "()I", (void*)android_server_AlarmManagerService_init},
{"close", "(I)V", (void*)android_server_AlarmManagerService_close},
{"set", "(IIJJ)V", (void*)android_server_AlarmManagerService_set},
{"waitForAlarm", "(I)I", (void*)android_server_AlarmManagerService_waitForAlarm},
{"setKernelTimezone", "(II)I", (void*)android_server_AlarmManagerService_setKernelTimezone},
};
```

当 AP 呼叫 AlarmManager.java 的 set 或 setRepeating 函数时，最终会呼叫 com_android_server_AlarmManagerService.cpp 的

```
static void android_server_AlarmManagerService_set (JNIEnv* env, jobject obj,
                        jint fd, jint type, jlong seconds, jlong nanoseconds)
```

在此函数中，会执行

```
ioctl(fd, ANDROID_ALARM_SET(type), &ts);
```

然后会呼叫到 alarm-dev.c 中 alarm_ioctl 中，接着 alarm-dev.c 会往它的红黑树中增加一个 alarm 节点。

在 AlarmManagerService 开始的时候，会启动一个 AlarmThread。在这个 AlarmThread 中有一个 while 循环去执行 waitForAlarm 这个动作，这个函数最终通过 JNI 机制呼叫到 com_android_server_AlarmManagerService.cpp 的

```
static jint android_server_AlarmManagerService_waitForAlarm(JNIEnv* env,
                                                        jobject obj, jint fd){
#if HAVE_ANDROID_OS
    int result = 0;

    do
    {
        result = ioctl(fd, ANDROID_ALARM_WAIT);
    } while (result < 0 && errno == EINTR);

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

```
    return result;
#endif
}
```

从 code 中可以看到，实际上它是在不断地执行 ioctl (fd，ANDROID_ALARM_WAIT)，上面说到，当闹钟到期时，alarm.c 中的 alarm_timer_triggered 函数会调用 alarm_triggered，这时，AP 层在呼叫 ioctl (fd，ANDROID_ALARM_WAIT)时，会返回表示等到 alarm 的返回值。

所以当闹钟到期时，AlarmThread 的 waitForAlarm 会返回一个值。接着通过执行 triggerAlarmsLocked，把几种类型的闹钟列表中符合要求的 alarm 添加到 triggerList 中，然后用 alarm.operation.send 发送消息，调起小闹钟程序。

AlarmThread 的 code 如下：

```
private class AlarmThread extends Thread {
    public AlarmThread() {
        super("AlarmManager");
    }

    public void run() {
        while (true)
        {
            int result = waitForAlarm(mDescriptor);

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

            if ((result & TIME_CHANGED_MASK) != 0) {
                remove(mTimeTickSender);
                mClockReceiver.scheduleTimeTickEvent();
                Intent intent = new Intent(Intent.ACTION_TIME_CHANGED);
                intent.addFlags(Intent.FLAG_RECEIVER_REPLACE_PENDING);
                mContext.sendBroadcast(intent);
            }

            synchronized (mLock) {
                final long nowRTC = System.currentTimeMillis();
                final long nowELAPSED = SystemClock.elapsedRealtime();
                if (localLOGV) Slog.v(
                    TAG, "Checking for alarms... rtc=" + nowRTC
                    + ", elapsed=" + nowELAPSED);

                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);

                // now trigger the alarms
                Iterator<Alarm> it = triggerList.iterator();
                while (it.hasNext()) {
                    Alarm alarm = it.next();
                    try {
                        if (localLOGV) Slog.v(TAG, "sending alarm " + alarm);
                        alarm.operation.send(mContext, 0,
```

```
                            mBackgroundIntent.putExtra(
                                Intent.EXTRA_ALARM_COUNT, alarm.count),
                            mResultReceiver, mHandler);

                        // we have an active broadcast so stay awake.
                        if (mBroadcastRefCount == 0) {
                            mWakeLock.acquire();
                        }
                        mBroadcastRefCount++;

                        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);
                        }
                    } catch (PendingIntent.CanceledException e) {
                        if (alarm.repeatInterval > 0) {
                            remove(alarm.operation);
                        }
                    } catch (RuntimeException e) {
                        Slog.w(TAG, "Failure sending alarm.", e);
                    }
                }
            }
        }
    }
}
```

11.4 接口

Android 中，Alarm 的操作通过 AlarmManager 来处理，AlarmManager 系统服务的具体实现在:frameworks/base/services/java/com/android/server/AlarmManagerServic.java 文件中。应用程序中可以通过 getSystemService 获得其系统服务，如下所示：

```
AlarmManager alarms = (AlarmManager)getSystemService(Context.ALARM_SERVICE);
```

为了创建一个新的 Alarm，使用 set 方法并指定一个 Alarm 类型、触发时间和在 Alarm 触发时要调用的 Intent。如果你设定的 Alarm 发生在过去，那么它将立即触发。

这里有 4 种 Alarm 类型。你的选择将决定你在 set 方法中传递的时间值代表什么，是特定的时间或者是时间流逝：

- RTC_WAKEUP：在指定的时刻（设置 Alarm 的时候），唤醒设备来触发 Intent。
- RTC：在一个显式的时间触发 Intent，但不唤醒设备。
- ELAPSED_REALTIME：从设备启动后，如果流逝的时间达到总时间，那么触发 Intent，但不唤醒设备。流逝的时间包括设备睡眠的任何时间。注意一点的是，时间流逝的计算点是自从它最后一次启动算起。
- ELAPSED_REALTIME_WAKEUP：从设备启动后，达到流逝的总时间后，如果需要将唤醒设备并触发 Intent。

这 4 种 Alarm 类型详情请参考 frameworks/base/core/java/android/app/AlarmManager.java。

11.5 实例

最后，请看一个 Alarm 的实例：

1、 建立一个 AlarmReceiver 继承入 BroadcastReceiver，并在 AndroidManifest.xml 声明

```
Public static class AlarmReceiver extends BroadcastReceiver {
        @Override
        Public void onReceive (Context context, Intent intent) {
            Toast.makeText(context, "时间到", Toast.LENGTH_LONG).show();
        }
}
```

2、 建立 Intent 和 PendingIntent，来调用目标组件。

```
Intent it = new Intent(this, AlarmReceiver.class);
PendingIntent pi = PendingIntent.getBroadcast(this, 0, intent, 0);
```

3、 设置闹钟

获取闹钟管理的实例：

```
AlarmManager am = (AlarmManager)getSystemService(Context.ALARM_SERVICE);
```

设置单次闹钟：

```
am.set(AlarmManager.RTC_WAKEUP, System.currentTimeMillis() + (5*1000), pi);
```

设置周期闹钟：

```
am.setRepeating(AlarmManager.RTC_WAKEUP,   System.currentTimeMillis()   +
(10*1000), (24*60*60*1000), pi);
```