**Android Framework 框架系列之 AlarmManagerService(一)**

本篇文章主要介绍 Android 开发中的**AlarmManagerService**部分知识点，通过阅读本篇文章，您将收获以下内容:

1.AlarmManager的使用  
2.AlarmManagerService初始化

本文转自网络地址如下：

http://www.robinheztto.com/2017/03/10/android-alarm-1/

Android系统通过AlarmManager向应用提供定时/闹钟服务，以使应用在其生命周期之外可执行基于特定时间的操作，本篇将具体分析AlarmManager的使用及AlarmManagerService服务的初始化。

相关源码位于以下文件中:

frameworks/base/core/java/android/app/AlarmManager.java

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

frameworks/base/services/core/jni/com\_android\_server\_AlarmManagerService.cpp

**AlarmManager的使用**

1.获取AlarmManager:  
Alarm相关的服务接口定义在AlarmManager中，与其他系统服务一样，通过Context获取AlarmManager。

Context.getSystemService(Context.ALARM\_SERVICE);

1. Alarm的类型:

* AlarmManager.RTC\_WAKEUP  
  使用系统绝对时间(当前系统时间，System.currentTimeMillis())，系统休眠状态也将唤醒系统。
* AlarmManager.RTC  
  使用系统绝对时间(当前系统时间，System.currentTimeMillis())，系统休眠状态下不可用。
* AlarmManager.ELAPSED\_REALTIME\_WAKEUP  
  使用系统相对时间(相对系统启动时间，SystemClock.elapsedRealtime())，系统休眠状态也将唤醒系统。
* AlarmManager.ELAPSED\_REALTIME  
  使用系统相对时间(相对系统启动时间，SystemClock.elapsedRealtime())，系统休眠状态下不可用  
  RTC/RTC\_WAKEUP和ELAPSED\_REALTIME/ELAPSED\_REALTIME\_WAKEUP最大的差别就是RTC受time zone/locale的影响，可以通过修改手机时间触发闹钟事件，ELAPSED\_REALTIME/ELAPSED\_REALTIME\_WAKEUP要通过真实时间的流逝，即使在休眠状态时间也会被计算。

**WAKEUP**类型的Alarm会唤醒系统，休眠状态下会增加系统的功耗，所以在使用中应尽量避免使用该种类型的Alarm。

1. Alarm的Flag:

* FLAG\_STANDALONE  
  指定stand-alone精准alarm，该alarm不会被batch，设置WINDOW\_EXACT的alarm会指定此flag。
* FLAG\_WAKE\_FROM\_IDLE  
  指定alarm即使在idle模式也将唤醒系统，如alarm clock。
* FLAG\_ALLOW\_WHILE\_IDLE  
  针对Doze模式，alarm即使在系统idle状态下也会执行，但是不会使系统退出idle mode，只有特殊alarm才需要标记该Flag。
* FLAG\_ALLOW\_WHILE\_IDLE\_UNRESTRICTED  
  针对Doze模式，alarm即使在系统idle状态下也会执行而且没有时间限制，但是不会使系统退出idle mode，只有特殊alarm才需要标记该Flag。
* FLAG\_IDLE\_UNTIL  
  只有调用AlarmManager.setIdleUntil()接口才可能设置该flag，用来使系统进入idle mode直到marker alarm被执行，执行marker alarm时系统会退出idle mode(设置后进入DozeIdle状态让Alarm系统挂起，直到这个Alarm到期)。

4.Alarm的set:

非精准Alarm，其window被指定为WINDOW\_HEURISTIC：

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

public void set(int type, long triggerAtMillis, String tag, OnAlarmListener listener,Handler targetHandler) {}

public void setRepeating(int type, long triggerAtMillis,long intervalMillis, PendingIntent operation) {}

public void setInexactRepeating(int type, long triggerAtMillis,long intervalMillis, PendingIntent operation) {}// Doze模式下

public void setAndAllowWhileIdle(int type, long triggerAtMillis, PendingIntent operation) {}

精准Alarm，其window被标记为WINDOW\_EXACT

public void setWindow(int type, long windowStartMillis, long windowLengthMillis,PendingIntent operation) {}

public void setWindow(int type, long windowStartMillis, long windowLengthMillis, String tag, OnAlarmListener listener, Handler targetHandler) {}

public void setExact(int type, long triggerAtMillis, PendingIntent operation) {}

public void setExact(int type, long triggerAtMillis, String tag, OnAlarmListener listener, Handler targetHandler) {}

public void setAlarmClock(AlarmClockInfo info, PendingIntent operation) {}// Doze模式下

public void setIdleUntil(int type, long triggerAtMillis, String tag, OnAlarmListener listener,Handler targetHandler) {}

public void setExactAndAllowWhileIdle(int type, long triggerAtMillis, PendingIntent operation) {}

AlarmManager中的set方法最终都是调用setImpl，下面是setImpl的具体实现。  
frameworks/base/core/java/android/app/AlarmManager.java

private void setImpl(@AlarmType int type, long triggerAtMillis, long windowMillis,

long intervalMillis, int flags, PendingIntent operation, final OnAlarmListener listener,

String listenerTag, Handler targetHandler, WorkSource workSource,

AlarmClockInfo alarmClock) {

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;

}

// OnAlarmListener封装到ListenerWrapper，并添加到sWrappers管理

ListenerWrapper recipientWrapper = null;

if (listener != null) {

synchronized (AlarmManager.class) {

if (sWrappers == null) {

sWrappers = new ArrayMap<OnAlarmListener, ListenerWrapper>();

}

recipientWrapper = sWrappers.get(listener);

// no existing wrapper => build a new one

if (recipientWrapper == null) {

recipientWrapper = new ListenerWrapper(listener);

sWrappers.put(listener, recipientWrapper);

}

}

final Handler handler = (targetHandler != null) ? targetHandler : mMainThreadHandler;

recipientWrapper.setHandler(handler);

}

// 调用AlarmManagerService

try {

mService.set(mPackageName, type, triggerAtMillis, windowMillis, intervalMillis, flags,

operation, recipientWrapper, listenerTag, workSource, alarmClock);

} catch (RemoteException ex) {

throw ex.rethrowFromSystemServer();

}

}

**AlarmManagerService初始化**

public AlarmManagerService(Context context) {

super(context);

mConstants = new Constants(mHandler);

}

下面先看Constants类的具体实现，主要负责Alarm相关的常量的读取及更新。

private final class Constants extends ContentObserver {

// Key names stored in the settings value.

private static final String KEY\_MIN\_FUTURITY = "min\_futurity";

private static final String KEY\_MIN\_INTERVAL = "min\_interval";

private static final String KEY\_ALLOW\_WHILE\_IDLE\_SHORT\_TIME = "allow\_while\_idle\_short\_time";

private static final String KEY\_ALLOW\_WHILE\_IDLE\_LONG\_TIME = "allow\_while\_idle\_long\_time";

private static final String KEY\_ALLOW\_WHILE\_IDLE\_WHITELIST\_DURATION

= "allow\_while\_idle\_whitelist\_duration";

private static final String KEY\_LISTENER\_TIMEOUT = "listener\_timeout";

private static final long DEFAULT\_MIN\_FUTURITY = 5 \* 1000;

private static final long DEFAULT\_MIN\_INTERVAL = 60 \* 1000;

private static final long DEFAULT\_ALLOW\_WHILE\_IDLE\_SHORT\_TIME = DEFAULT\_MIN\_FUTURITY;

private static final long DEFAULT\_ALLOW\_WHILE\_IDLE\_LONG\_TIME = 9\*60\*1000;

private static final long DEFAULT\_ALLOW\_WHILE\_IDLE\_WHITELIST\_DURATION = 10\*1000;

private static final long DEFAULT\_LISTENER\_TIMEOUT = 5 \* 1000;

// Minimum futurity of a new alarm

public long MIN\_FUTURITY = DEFAULT\_MIN\_FUTURITY;

// Minimum alarm recurrence interval

public long MIN\_INTERVAL = DEFAULT\_MIN\_INTERVAL;

// Minimum time between ALLOW\_WHILE\_IDLE alarms when system is not idle.

public long ALLOW\_WHILE\_IDLE\_SHORT\_TIME = DEFAULT\_ALLOW\_WHILE\_IDLE\_SHORT\_TIME;

// Minimum time between ALLOW\_WHILE\_IDLE alarms when system is idling.

public long ALLOW\_WHILE\_IDLE\_LONG\_TIME = DEFAULT\_ALLOW\_WHILE\_IDLE\_LONG\_TIME;

// BroadcastOptions.setTemporaryAppWhitelistDuration() to use for FLAG\_ALLOW\_WHILE\_IDLE.

public long ALLOW\_WHILE\_IDLE\_WHITELIST\_DURATION

= DEFAULT\_ALLOW\_WHILE\_IDLE\_WHITELIST\_DURATION;

// Direct alarm listener callback timeout

public long LISTENER\_TIMEOUT = DEFAULT\_LISTENER\_TIMEOUT;

private ContentResolver mResolver;

private final KeyValueListParser mParser = new KeyValueListParser(',');

private long mLastAllowWhileIdleWhitelistDuration = -1;

public Constants(Handler handler) {

super(handler);

updateAllowWhileIdleMinTimeLocked();

updateAllowWhileIdleWhitelistDurationLocked();

}

public void start(ContentResolver resolver) {

mResolver = resolver;

mResolver.registerContentObserver(Settings.Global.getUriFor(

Settings.Global.ALARM\_MANAGER\_CONSTANTS), false, this);

updateConstants();

}

public void updateAllowWhileIdleMinTimeLocked() {

mAllowWhileIdleMinTime = mPendingIdleUntil != null

? ALLOW\_WHILE\_IDLE\_LONG\_TIME : ALLOW\_WHILE\_IDLE\_SHORT\_TIME;

}

public void updateAllowWhileIdleWhitelistDurationLocked() {

if (mLastAllowWhileIdleWhitelistDuration != ALLOW\_WHILE\_IDLE\_WHITELIST\_DURATION) {

mLastAllowWhileIdleWhitelistDuration = ALLOW\_WHILE\_IDLE\_WHITELIST\_DURATION;

BroadcastOptions opts = BroadcastOptions.makeBasic();

opts.setTemporaryAppWhitelistDuration(ALLOW\_WHILE\_IDLE\_WHITELIST\_DURATION);

mIdleOptions = opts.toBundle();

}

}

@Override

public void onChange(boolean selfChange, Uri uri) {

updateConstants();

}

private void updateConstants() {

synchronized (mLock) {

try {

mParser.setString(Settings.Global.getString(mResolver,

Settings.Global.ALARM\_MANAGER\_CONSTANTS));

} catch (IllegalArgumentException e) {

// Failed to parse the settings string, log this and move on

// with defaults.

Slog.e(TAG, "Bad alarm manager settings", e);

}

MIN\_FUTURITY = mParser.getLong(KEY\_MIN\_FUTURITY, DEFAULT\_MIN\_FUTURITY);

MIN\_INTERVAL = mParser.getLong(KEY\_MIN\_INTERVAL, DEFAULT\_MIN\_INTERVAL);

ALLOW\_WHILE\_IDLE\_SHORT\_TIME = mParser.getLong(KEY\_ALLOW\_WHILE\_IDLE\_SHORT\_TIME,

DEFAULT\_ALLOW\_WHILE\_IDLE\_SHORT\_TIME);

ALLOW\_WHILE\_IDLE\_LONG\_TIME = mParser.getLong(KEY\_ALLOW\_WHILE\_IDLE\_LONG\_TIME,

DEFAULT\_ALLOW\_WHILE\_IDLE\_LONG\_TIME);

ALLOW\_WHILE\_IDLE\_WHITELIST\_DURATION = mParser.getLong(

KEY\_ALLOW\_WHILE\_IDLE\_WHITELIST\_DURATION,

DEFAULT\_ALLOW\_WHILE\_IDLE\_WHITELIST\_DURATION);

LISTENER\_TIMEOUT = mParser.getLong(KEY\_LISTENER\_TIMEOUT,

DEFAULT\_LISTENER\_TIMEOUT);

updateAllowWhileIdleMinTimeLocked();

updateAllowWhileIdleWhitelistDurationLocked();

}

}

void dump(PrintWriter pw) {

pw.println(" Settings:");

pw.print(" "); pw.print(KEY\_MIN\_FUTURITY); pw.print("=");

TimeUtils.formatDuration(MIN\_FUTURITY, pw);

pw.println();

pw.print(" "); pw.print(KEY\_MIN\_INTERVAL); pw.print("=");

TimeUtils.formatDuration(MIN\_INTERVAL, pw);

pw.println();

pw.print(" "); pw.print(KEY\_LISTENER\_TIMEOUT); pw.print("=");

TimeUtils.formatDuration(LISTENER\_TIMEOUT, pw);

pw.println();

pw.print(" "); pw.print(KEY\_ALLOW\_WHILE\_IDLE\_SHORT\_TIME); pw.print("=");

TimeUtils.formatDuration(ALLOW\_WHILE\_IDLE\_SHORT\_TIME, pw);

pw.println();

pw.print(" "); pw.print(KEY\_ALLOW\_WHILE\_IDLE\_LONG\_TIME); pw.print("=");

TimeUtils.formatDuration(ALLOW\_WHILE\_IDLE\_LONG\_TIME, pw);

pw.println();

pw.print(" "); pw.print(KEY\_ALLOW\_WHILE\_IDLE\_WHITELIST\_DURATION); pw.print("=");

TimeUtils.formatDuration(ALLOW\_WHILE\_IDLE\_WHITELIST\_DURATION, pw);

pw.println();

}

}

AlarmManagerService实例化后即调用onStart()方法。

@Override

public void onStart() {

// native层初始化

mNativeData = init();

mNextWakeup = mNextNonWakeup = 0;

// We have to set current TimeZone info to kernel

// because kernel doesn't keep this after reboot

setTimeZoneImpl(SystemProperties.get(TIMEZONE\_PROPERTY));

/// M:add for PPL feature ,@{

initPpl();

///@}

/// M: For handling non-wakeup alarms while WFD is connected

registerWFDStatusChangeReciever();

///@}

/// M: added for BG powerSaving feature @{

initAlarmGrouping();

///@}

// Also sure that we're booting with a halfway sensible current time

if (mNativeData != 0) {

final long systemBuildTime = Environment.getRootDirectory().lastModified();

if (System.currentTimeMillis() < systemBuildTime) {

Slog.i(TAG, "Current time only " + System.currentTimeMillis()

+ ", advancing to build time " + systemBuildTime);

setKernelTime(mNativeData, systemBuildTime);

}

}

// Determine SysUI's uid

final PackageManager packMan = getContext().getPackageManager();

try {

PermissionInfo sysUiPerm = packMan.getPermissionInfo(SYSTEM\_UI\_SELF\_PERMISSION, 0);

ApplicationInfo sysUi = packMan.getApplicationInfo(sysUiPerm.packageName, 0);

if ((sysUi.privateFlags&ApplicationInfo.PRIVATE\_FLAG\_PRIVILEGED) != 0) {

mSystemUiUid = sysUi.uid;

} else {

Slog.e(TAG, "SysUI permission " + SYSTEM\_UI\_SELF\_PERMISSION

+ " defined by non-privileged app " + sysUi.packageName

+ " - ignoring");

}

} catch (NameNotFoundException e) {

}

if (mSystemUiUid <= 0) {

Slog.wtf(TAG, "SysUI package not found!");

}

PowerManager pm = (PowerManager) getContext().getSystemService(Context.POWER\_SERVICE);

mWakeLock = pm.newWakeLock(PowerManager.PARTIAL\_WAKE\_LOCK, "\*alarm\*");

mTimeTickSender = PendingIntent.getBroadcastAsUser(getContext(), 0,

new Intent(Intent.ACTION\_TIME\_TICK).addFlags(

Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY

| Intent.FLAG\_RECEIVER\_FOREGROUND

| Intent.FLAG\_RECEIVER\_VISIBLE\_TO\_INSTANT\_APPS), 0,

UserHandle.ALL);

Intent intent = new Intent(Intent.ACTION\_DATE\_CHANGED);

intent.addFlags(Intent.FLAG\_RECEIVER\_REPLACE\_PENDING

| Intent.FLAG\_RECEIVER\_VISIBLE\_TO\_INSTANT\_APPS);

mDateChangeSender = PendingIntent.getBroadcastAsUser(getContext(), 0, intent,

Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY\_BEFORE\_BOOT, UserHandle.ALL);

// now that we have initied the driver schedule the alarm

mClockReceiver = new ClockReceiver();

mClockReceiver.scheduleTimeTickEvent();

mClockReceiver.scheduleDateChangedEvent();

mInteractiveStateReceiver = new InteractiveStateReceiver();

mUninstallReceiver = new UninstallReceiver();

if (mNativeData != 0) {

AlarmThread waitThread = new AlarmThread();

waitThread.start();

} else {

Slog.w(TAG, "Failed to open alarm driver. Falling back to a handler.");

}

try {

ActivityManager.getService().registerUidObserver(new UidObserver(),

ActivityManager.UID\_OBSERVER\_IDLE, ActivityManager.PROCESS\_STATE\_UNKNOWN, null);

} catch (RemoteException e) {

// ignored; both services live in system\_server

}

publishBinderService(Context.ALARM\_SERVICE, mService);

publishLocalService(LocalService.class, new LocalService());

}

在onStart()后，SYSTEM\_SERVICES\_READY时onBootPhase()将被回调。

@Override

public void onBootPhase(int phase) {

if (phase == PHASE\_SYSTEM\_SERVICES\_READY) {

mConstants.start(getContext().getContentResolver());

mAppOps = (AppOpsManager) getContext().getSystemService(Context.APP\_OPS\_SERVICE);

mLocalDeviceIdleController

= LocalServices.getService(DeviceIdleController.LocalService.class);

}

}

回到onStart()，分析native层init的调用，下面看native init()的实现。

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

private native long init();

frameworks/base/services/core/jni/com\_android\_server\_AlarmManagerService.cpp

static const JNINativeMethod sMethods[] = {

/\* name, signature, funcPtr \*/

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

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

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

{"clear", "(JIJJ)V", (void\*)android\_server\_AlarmManagerService\_clear},

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

{"setKernelTime", "(JJ)I", (void\*)android\_server\_AlarmManagerService\_setKernelTime},

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

};

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

{

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

sMethods, NELEM(sMethods));

}

register\_android\_server\_AlarmManagerService中注册了native方法，init()即调用android\_server\_AlarmManagerService\_init。

frameworks/base/services/core/jni/com\_android\_server\_AlarmManagerService.cpp

static jlong android\_server\_AlarmManagerService\_init(JNIEnv\*, jobject)

{

// 初始化/dev/alarm

jlong ret = init\_alarm\_driver();

if (ret) {

return ret;

}

// 如果初始化/dev/alarm不成功，则进入timerfd初始化，现一般采用timerfd方式采用

return init\_timerfd();

}

Native Alarm初始化采用了二种方案，AlarmDriver与timerfd，当alarm\_driver失败时则使用timerfd，现在基本使用的是timerfd。如下，AlarmImpl是Native层Alarm操作的统一接口，AlarmImplAlarmDriver与AlarmImplTimerFd是AlarmDriver与timerfd二种不同方式的具体实现。

class AlarmImpl

{

public:

AlarmImpl(int \*fds, size\_t n\_fds);

virtual ~AlarmImpl();

virtual int set(int type, struct timespec \*ts) = 0;

virtual int clear(int type, struct timespec \*ts) = 0;

virtual int setTime(struct timeval \*tv) = 0;

virtual int waitForAlarm() = 0;

protected:

int \*fds;

size\_t n\_fds;

};

class AlarmImplAlarmDriver : public AlarmImpl

{

public:

AlarmImplAlarmDriver(int fd) : AlarmImpl(&fd, 1) { }

int set(int type, struct timespec \*ts);

int clear(int type, struct timespec \*ts);

int setTime(struct timeval \*tv);

int waitForAlarm();

};

class AlarmImplTimerFd : public AlarmImpl

{

public:

AlarmImplTimerFd(int fds[N\_ANDROID\_TIMERFDS], int epollfd, int rtc\_id) :

AlarmImpl(fds, N\_ANDROID\_TIMERFDS), epollfd(epollfd), rtc\_id(rtc\_id) { }

~AlarmImplTimerFd();

int set(int type, struct timespec \*ts);

int clear(int type, struct timespec \*ts);

int setTime(struct timeval \*tv);

int waitForAlarm();

private:

int epollfd;

int rtc\_id;

};

下面先看AlarmDriver的方式。

frameworks/base/services/core/jni/com\_android\_server\_AlarmManagerService.cpp

static jlong init\_alarm\_driver()

{

// 打开/dev/alarm,失败则退出

int fd = open("/dev/alarm", O\_RDWR);

if (fd < 0) {

ALOGV("opening alarm driver failed: %s", strerror(errno));

return 0;

}

// 根据fd创建AlarmImplAlarmDriver对象

AlarmImpl \*ret = new AlarmImplAlarmDriver(fd);

return reinterpret\_cast<jlong>(ret);

}

int AlarmImplAlarmDriver::set(int type, struct timespec \*ts)

{

return ioctl(fds[0], ANDROID\_ALARM\_SET(type), ts);

}

int AlarmImplAlarmDriver::clear(int type, struct timespec \*ts)

{

return ioctl(fds[0], ANDROID\_ALARM\_CLEAR(type), ts);

}

int AlarmImplAlarmDriver::setTime(struct timeval \*tv)

{

struct timespec ts;

int res;

ts.tv\_sec = tv->tv\_sec;

ts.tv\_nsec = tv->tv\_usec \* 1000;

res = ioctl(fds[0], ANDROID\_ALARM\_SET\_RTC, &ts);

if (res < 0)

ALOGV("ANDROID\_ALARM\_SET\_RTC ioctl failed: %s\n", strerror(errno));

return res;

}

int AlarmImplAlarmDriver::waitForAlarm()

{

return ioctl(fds[0], ANDROID\_ALARM\_WAIT);

}

AlarmImplAlarmDriver中主要通过ioctl来实现Alarm的操作。当init\_alarm\_driver打开/dev/alarm失败时，选择timerfd实现Alarm的操作。

frameworks/base/services/core/jni/com\_android\_server\_AlarmManagerService.cpp

static const size\_t N\_ANDROID\_TIMERFDS = ANDROID\_ALARM\_TYPE\_COUNT + 1;

static const clockid\_t android\_alarm\_to\_clockid[N\_ANDROID\_TIMERFDS] = {

CLOCK\_REALTIME\_ALARM,

CLOCK\_REALTIME,

CLOCK\_BOOTTIME\_ALARM,

CLOCK\_BOOTTIME,

CLOCK\_MONOTONIC,

CLOCK\_POWEROFF\_ALARM,

CLOCK\_REALTIME,

};

static jlong init\_timerfd()

{

int epollfd;

int fds[N\_ANDROID\_TIMERFDS];

// 创建epoll句柄，监听N\_ANDROID\_TIMERFDS个文件描述符

epollfd = epoll\_create(N\_ANDROID\_TIMERFDS);

if (epollfd < 0) {

ALOGV("epoll\_create(%zu) failed: %s", N\_ANDROID\_TIMERFDS,

strerror(errno));

return 0;

}

for (size\_t i = 0; i < N\_ANDROID\_TIMERFDS; i++) {

// 创建定时器文件

fds[i] = timerfd\_create(android\_alarm\_to\_clockid[i], 0);

if (fds[i] < 0) {

ALOGV("timerfd\_create(%u) failed: %s", android\_alarm\_to\_clockid[i],

strerror(errno));

close(epollfd);

for (size\_t j = 0; j < i; j++) {

close(fds[j]);

}

return 0;

}

}

// 根据fds创建AlarmImplTimerFd对象，AlarmImplTimerFd也继承于AlarmImpl

AlarmImpl \*ret = new AlarmImplTimerFd(fds, epollfd, wall\_clock\_rtc());

for (size\_t i = 0; i < N\_ANDROID\_TIMERFDS; i++) {

epoll\_event event;

event.events = EPOLLIN | EPOLLWAKEUP;

event.data.u32 = i;

// 将创建的定时器文件列表加入到epoll监听中

int err = epoll\_ctl(epollfd, EPOLL\_CTL\_ADD, fds[i], &event);

if (err < 0) {

ALOGV("epoll\_ctl(EPOLL\_CTL\_ADD) failed: %s", strerror(errno));

delete ret;

return 0;

}

}

struct itimerspec spec;

memset(&spec, 0, sizeof(spec));

int err = timerfd\_settime(fds[ANDROID\_ALARM\_TYPE\_COUNT],

TFD\_TIMER\_ABSTIME | TFD\_TIMER\_CANCEL\_ON\_SET, &spec, NULL);

if (err < 0) {

ALOGV("timerfd\_settime() failed: %s", strerror(errno));

delete ret;

return 0;

}

return reinterpret\_cast<jlong>(ret);

}

init\_timerfd()中利用epoll+timerfd的方式，创建timerfd文件并加入到epoll监听中，创建的定时器文件中，我们主要使用的是CLOCK\_REALTIME\_ALARM，CLOCK\_BOOTTIME\_ALARM，CLOCK\_POWEROFF\_ALARM，分别对应RTC\_WAKEUP(RTC)，ELAPSED\_REALTIME\_WAKEUP(ELAPSED\_REALTIME)，RTC\_POWEROFF\_WAKEUP。下面具体看一下AlarmImplTimerFd::set设置定时器的实现。  
frameworks/base/services/core/jni/com\_android\_server\_AlarmManagerService.cpp

int AlarmImplTimerFd::set(int type, struct timespec \*ts)

{

if (type > ANDROID\_ALARM\_TYPE\_COUNT) {

errno = EINVAL;

return -1;

}

if (!ts->tv\_nsec && !ts->tv\_sec) {

ts->tv\_nsec = 1;

}

/\* timerfd interprets 0 = disarm, so replace with a practically

equivalent deadline of 1 ns \*/

struct itimerspec spec;

memset(&spec, 0, sizeof(spec));

memcpy(&spec.it\_value, ts, sizeof(spec.it\_value));

// 直接调用timerfd\_settime设置Alarm定时时间

return timerfd\_settime(fds[type], TFD\_TIMER\_ABSTIME, &spec, NULL);

}

在waitForAlarm中将等待Alarm的到来，下面看AlarmImplTimerFd::waitForAlarm的实现。

frameworks/base/services/core/jni/com\_android\_server\_AlarmManagerService.cpp

int AlarmImplTimerFd::waitForAlarm()

{

epoll\_event events[N\_ANDROID\_TIMERFDS];

// 利用epolle\_wait监听定时器的事件

int nevents = epoll\_wait(epollfd, events, N\_ANDROID\_TIMERFDS, -1);

if (nevents < 0) {

return nevents;

}

int result = 0;

// 事件到来，循环读取定时器文件

for (int i = 0; i < nevents; i++) {

uint32\_t alarm\_idx = events[i].data.u32;

uint64\_t unused;

ssize\_t err = read(fds[alarm\_idx], &unused, sizeof(unused));

if (err < 0) {

if (alarm\_idx == ANDROID\_ALARM\_TYPE\_COUNT && errno == ECANCELED) {

// 时间改变

result |= ANDROID\_ALARM\_TIME\_CHANGE\_MASK;

} else {

return err;

}

} else {

// 设置result为触发的alarm\_idx

result |= (1 << alarm\_idx);

}

}

// 返回结果给AlarmManagerService

return result;

}

waitForAlarm中一直在epoll\_wait监听等待Alarm fd事件，当事件到来，循环读取定时器文件并向上层返回触发的Alarm index或时间改变事件。