目录

[Doze概述 1](#_Toc32033)

[Doze相关命令 2](#_Toc20775)

[进入和退出Doze 2](#_Toc20420)

[Doze白名单 5](#_Toc5839)

[其他命令和字段意义 6](#_Toc8253)

[DeviceIdleController的启动流程 6](#_Toc12979)

[DeviceIdleController的状态变化 11](#_Toc20415)

[Light doze – scheduleLightAlarmLocked 13](#_Toc6464)

[Deep doze – scheduleAlarmLocked 16](#_Toc16640)

[小结 20](#_Toc19044)

[DeviceIdleController相关限制 20](#_Toc17410)

[DeviceIdleController对网络的限制 20](#_Toc3769)

[DeviceIdleController对wakelock限制 23](#_Toc23818)

[DeviceIdleController对job的延迟 24](#_Toc10025)

# Doze概述

如果用户关闭设备屏幕但仍处于移动状态时，则设备进入轻度Doze模式，此外，轻度Doze模式只适合Android7.0及以上版本

如果用户长时间没有主动使用其设备，处于静止状态且屏幕已关闭，则系统会使设备进入Doze模式，也就是深度Doze模式

当系统处于深度Doze模式下，系统和白名单之外的应用将受到以下限制

▪ 无法访问网络

▪ Wake Locks被忽略

▪ AlarmManager闹铃会被推迟到下一个maintenance window响应，alarms定期系统会退出Doze模式一小段时间让app完成推迟的活动，此段时间称为maintenance window(维护时段)，在这段时间系统运行此前挂起的syncs，jobs，alarms，并且让app能够访问网络

但使用AlarmManager的setAndAllowWhileIdle、setExactAndAllowWhileIdle和setAlarmClock时，alarms定义事件仍会启动。在这些alarms启动前，系统会短暂地退出Doze模式

▪ 系统不执行Wi-Fi/GPS扫描

▪ 系统不允许sync adapters运行

▪ 系统不允许JobScheduler运行

而位于白名单中的应用可以

▪ 继续使用网络并保留部分wake lock

▪ Job和同步仍然会被推迟

▪ 常规的AlarmManager闹铃也不会被触发

# Doze相关命令

## 进入和退出Doze

device idle有很多的命令，我们先看下如何进入和退出DOZE，再看下白名单添加方式

**(1)模拟未充电状态**

**1)查看当前的手机状态**

adb shell dumpsys battery

Current Battery Service state:

AC powered: false

USB powered: true --表示USB正在充电

Wireless powered: false

Max charging current: 0

Max charging voltage: 0

Charge counter: 347575

status: 5

health: 2

present: true

level: 100

scale: 100

voltage: 4346

temperature: 250

technology:

**2)模拟手机未充电状态**

adb shell dumpsys battery unplug

adb shell dumpsys battery

Current Battery Service state:

(UPDATES STOPPED -- use 'reset' to restart)

AC powered: false

USB powered: false --USB未充电

Wireless powered: false

Max charging current: 0

Max charging voltage: 0

Charge counter: 347575

status: 5

health: 2

present: true

level: 100

scale: 100

voltage: 4346

temperature: 250

technology:

**(2)IDLE有效化**

adb shell dumpsys deviceidle enable

Deep idle mode enabled

Light idle mode enable

**(3)进入IDLE模式方法**

有两种方法。

第一种是，屏幕亮着状态按下电源按钮关闭屏幕，敲入命令让其进入IDLE模式。

adb shell dumpsys deviceidle step

Stepped to deep: IDLE\_PENDING

adb shell dumpsys deviceidle step

Stepped to deep: SENSING

adb shell dumpsys deviceidle step

Stepped to deep: LOCATING

adb shell dumpsys deviceidle step

Stepped to deep: IDLE

第二种是，敲入命令强制让手机进入IDLE模式。

adb shell dumpsys deviceidle force-idle

Now forced in to deep idle mode

我们可以敲入命令查看并验证当前手机是否已经进入IDLE模式

adb shell dumpsys deviceidle

Settings:

light\_after\_inactive\_to=+3m0s0ms

light\_pre\_idle\_to=+3m0s0ms

light\_idle\_to=+5m0s0ms

light\_idle\_factor=2.0

light\_max\_idle\_to=+15m0s0ms

light\_idle\_maintenance\_min\_budget=+1m0s0ms

light\_idle\_maintenance\_max\_budget=+5m0s0ms

min\_light\_maintenance\_time=+5s0ms

min\_deep\_maintenance\_time=+30s0ms

inactive\_to=+30m0s0ms

sensing\_to=+4m0s0ms

locating\_to=+30s0ms

location\_accuracy=20.0m

motion\_inactive\_to=+10m0s0ms

idle\_after\_inactive\_to=+30m0s0ms

idle\_pending\_to=+5m0s0ms

max\_idle\_pending\_to=+10m0s0ms

idle\_pending\_factor=2.0

quick\_doze\_delay\_to=+1m0s0ms

idle\_to=+1h0m0s0ms

max\_idle\_to=+6h0m0s0ms

idle\_factor=2.0

min\_time\_to\_alarm=+1h0m0s0ms

max\_temp\_app\_whitelist\_duration=+5m0s0ms

mms\_temp\_app\_whitelist\_duration=+1m0s0ms

sms\_temp\_app\_whitelist\_duration=+20s0ms

notification\_whitelist\_duration=+30s0ms

wait\_for\_unlock=true

pre\_idle\_factor\_long=1.67

pre\_idle\_factor\_short=0.33

Idling history:

deep-idle: -12m12s404ms

Whitelist (except idle) system apps:

com.android.providers.calendar

com.android.providers.downloads

com.android.cellbroadcastreceiver

com.android.proxyhandler

com.android.shell

com.android.providers.contacts

Whitelist system apps:

com.android.providers.downloads

com.android.cellbroadcastreceiver

com.android.shell

Whitelist (except idle) all app ids:

2000

10036

10040

10041

10043

10054

Whitelist all app ids:

2000

10040

10054

mLightEnabled=true mDeepEnabled=true

mForceIdle=true

mUseMotionSensor=true mMotionSensor={Sensor name="Significant motion", vendor="Samsung", version=1, type=17, maxRange=1.0, resolution=1.0, power=0.001, minDelay=-1}

mScreenOn=false

mScreenLocked=true

mNetworkConnected=false

mCharging=false

mMotionActive=true

mNotMoving=false

mLocating=false mHasGps=true mHasNetwork=false mLocated=false

mState=IDLE mLightState=OVERRIDE

mInactiveTimeout=+30m0s0ms

mNextAlarmTime=+47m47s593ms

mNextIdlePendingDelay=+5m0s0ms

mNextIdleDelay=+2h0m0s0ms

**(4)恢复手机状态**

当我们模拟完状态之后要恢复回去，以便手机能够正常使用。

敲入命令 adb shell dumpsys deviceidle disable 和 adb shell dumpsys battery reset 即可让手机恢复状态。

再敲入 adb shell dumpsys battery 查看一下手机是否已经恢复状态。

Current Battery Service state:

AC powered: false

USB powered: true

Wireless powered: false

Max charging current: 0

Max charging voltage: 0

Charge counter: 273825

status: 2

health: 2

present: true

level: 78

scale: 100

voltage: 4188

temperature: 250

technology:

我们可以看到，手机已经从IDLE模式恢复到正常状态。

## Doze白名单

系统内置的白名单需要在/etc/permissions/platform.xml中添加

<!-- This is a core platform component that needs to freely run in the background -->

<allow-in-power-save package="com.android.cellbroadcastreceiver" />

<allow-in-power-save package="com.android.shell" />

<!-- Whitelist system providers -->

<allow-in-power-save-except-idle package="com.android.providers.calendar" />

<allow-in-power-save-except-idle package="com.android.providers.contacts" />

而三方应用在/data/system/deviceidle.xml，如果测试机没有这个文件，我们可以推送一份进去

adb push deviceidle.xml /data/system/deviceidle.xml

<?xml version='1.0' encoding='utf-8' standalone='yes' ?>

<config>

<wl n="com.tencent.mm" />

<wl n="com.tencent.mobileqq" />

<wl n="com.kiloo.subwaysurf" />

</config>

或者是在设置中将应用加入白名单，这样会产生/data/system/deviceidle.xml，通过“设置”>“应用和通知”>“特殊应用权限”>“电池优化”中添加或移除，然后我们可以看下

adb shell dumpsys deviceidle

…

Whitelist user apps:

com.tencent.mm

com.tencent.mobileqq

com.kiloo.subwaysurf

…

## 其他命令和字段意义

|  |  |  |
| --- | --- | --- |
| dumpsys deviceidle get light | mLightState | 获得值 |
| dumpsys deviceidle get deep | mState | 获得值 |
| dumpsys deviceidle get force | mForceIdle | 获得值 |
| dumpsys deviceidle get quick | mQuickDozeActivated | 获得值 |
| dumpsys deviceidle get screen | mScreenOn | 获得值 |
| dumpsys deviceidle get charging | mCharging | 获得值 |
| dumpsys deviceidle get network | mNetworkConnected | 获得值 |
| dumpsys deviceidle enable | mLightEnabled, mDeepEnabled | 设置true |
| dumpsys deviceidle disable | mLightEnabled, mDeepEnabled | 设置false |

dumpsys deviceidle

|  |  |  |
| --- | --- | --- |
| 代码中过的变量 | 命令中的keyword | 米8 |
| INACTIVE\_TIMEOUT | inactive\_to | 15m0s0ms |
| LIGHT\_IDLE\_AFTER\_INACTIVE\_TIMEOUT | light\_after\_inactive\_to | 3m0s0ms |
| IDLE\_AFTER\_INACTIVE\_TIMEOUT | idle\_after\_inactive\_to | 15m0s0ms |
| SENSING\_TIMEOUT | sensing\_to | 4m0s0ms |
| LOCATING\_TIMEOUT | locating\_to | 5s0ms |
| IDLE\_TIMEOUT | idle\_to | 1h0m0s0ms |
| IDLE\_FACTOR | idle\_factor | 2.0 |
| MAX\_IDLE\_TIMEOUT | max\_idle\_to | 6h0m0s0ms |
| IDLE\_PENDING\_TIMEOUT | idle\_pending\_to | 5m0s0ms |
| IDLE\_PENDING\_FACTOR | idle\_pending\_factor | 2.0 |
| MAX\_IDLE\_PENDING\_TIMEOUT | max\_idle\_pending\_to | 10m0s0ms |
|  |  |  |
|  |  |  |
|  |  |  |

本文用的是Android 10系统源码进行分析，资源文件位置frameworks/base/core/res/res/values/

# DeviceIdleController的启动流程

1. 开始时跟其他service一样，由SystemServer拉起来

frameworks/base/services/java/com/android/server/SystemServer.java

startOtherServices() –> mSystemServiceManager.startService(DeviceIdleController.class);

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

DeviceIdleController.java-->onStart()

public void onStart() {

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

synchronized (this) {

// 由adb shell dumpsys deviceidle enable / disable 开启或者禁止，默认禁止

mLightEnabled = mDeepEnabled = getContext().getResources().getBoolean(

com.android.internal.R.bool.config\_enableAutoPowerModes);

SystemConfig sysConfig = SystemConfig.getInstance();

// 从/etc/permissions/platform.xml获得"allow-in-power-save-except-idle"下的数据，即省电模式下且不在idle状态的白名单列表，即列表中的app能够在省电模式下在后台运行

ArraySet<String> allowPowerExceptIdle = sysConfig.getAllowInPowerSaveExceptIdle();

for (int i=0; i<allowPowerExceptIdle.size(); i++) {

String pkg = allowPowerExceptIdle.valueAt(i);

try {

//获取白名单列表中的系统应用

ApplicationInfo ai = pm.getApplicationInfo(pkg, PackageManager.MATCH\_SYSTEM\_ONLY);

int appid = UserHandle.getAppId(ai.uid);

mPowerSaveWhitelistAppsExceptIdle.put(ai.packageName, appid); //添加到ArrayMap中

mPowerSaveWhitelistSystemAppIdsExceptIdle.put(appid, true);//添加到SparseBooleanArray中

} catch (PackageManager.NameNotFoundException e) {

}

}

// 从/etc/permissions/platform.xml获得"allow-in-power-save"下的数据，即可以在省电模式下也可以在DOZE模式下运行

ArraySet<String> allowPower = sysConfig.getAllowInPowerSave();

for (int i=0; i<allowPower.size(); i++) {

String pkg = allowPower.valueAt(i);

try {

ApplicationInfo ai = pm.getApplicationInfo(pkg, PackageManager.MATCH\_SYSTEM\_ONLY);

int appid = UserHandle.getAppId(ai.uid);

// These apps are on both the whitelist-except-idle as well as the full whitelist, so they apply in all cases.

mPowerSaveWhitelistAppsExceptIdle.put(ai.packageName, appid);

mPowerSaveWhitelistSystemAppIdsExceptIdle.put(appid, true);

mPowerSaveWhitelistApps.put(ai.packageName, appid);

mPowerSaveWhitelistSystemAppIds.put(appid, true);

} catch (PackageManager.NameNotFoundException e) {

}

}

//设置Contants内容观察者

mConstants = mInjector.getConstants(this, mHandler, getContext().getContentResolver());

//读取/data/system/deviceidle.xml文件，将读取的app添加到表示用户设置的白名单中

readConfigFileLocked();

updateWhitelistAppIdsLocked();

mNetworkConnected = true;

mScreenOn = true;

mScreenLocked = false;

// Start out assuming we are charging.

// If we aren't, we will at least get a battery update the next time the level drops.

mCharging = true;

mActiveReason = ACTIVE\_REASON\_UNKNOWN;

mState = STATE\_ACTIVE; //设备保持活动状态，深度Doze的初始值

mLightState = LIGHT\_STATE\_ACTIVE; //设备保持活动状态，轻度Doze的初始值

mInactiveTimeout = mConstants.INACTIVE\_TIMEOUT;

// 应用于INACTIVE\_TIMEOUT和IDLE\_AFTER\_INACTIVE\_TIMEOUT的乘机因子，可以调节该数据来更快或者更慢的进入STATE\_IDLE，不要将此应用于感应超时或定位超时，因为：

-两个都短

-设备传感器可能需要时间才能稳定下来

如果设备正在运行，也不要应用该因子，因为设备运行提供的信号比预测算法更强。

mPreIdleFactor = 1.0f;

mLastPreIdleFactor = 1.0f;

}

mBinderService = new BinderService();

publishBinderService(Context.DEVICE\_IDLE\_CONTROLLER, mBinderService);

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

}

1. 在onStart()方法中，首先通过SystemConfig读取了两类白名单列表：在低电量模式下后台允许运行的应用的白名单、

在低电量模式和Doze模式下都允许后台运行的应用白名单：

frameworks/base/core/java/com/android/server/SystemConfig.java

public ArraySet<String> getAllowInPowerSaveExceptIdle() {

return mAllowInPowerSaveExceptIdle;

}

public ArraySet<String> getAllowInPowerSave() {

return mAllowInPowerSave;

}

private void readPermissionsFromXml(File permFile, int permissionFlag) {

…

} else if ("allow-in-power-save-except-idle".equals(name) && allowAll) {

String pkgname = parser.getAttributeValue(null, "package");

if (pkgname == null) {

Slog.w(TAG, "<allow-in-power-save-except-idle> without package in "

+ permFile + " at " + parser.getPositionDescription());

} else {

mAllowInPowerSaveExceptIdle.add(pkgname);

}

XmlUtils.skipCurrentTag(parser);

continue;

} else if ("allow-in-power-save".equals(name) && allowAll) {

String pkgname = parser.getAttributeValue(null, "package");

if (pkgname == null) {

Slog.w(TAG, "<allow-in-power-save> without package in " + permFile + " at "

+ parser.getPositionDescription());

} else {

mAllowInPowerSave.add(pkgname);

}

XmlUtils.skipCurrentTag(parser);

continue;

…

}

1. onStart()调用updateWhitelistAppIdsLocked()更新白名单列表

private void updateWhitelistAppIdsLocked() {

mPowerSaveWhitelistExceptIdleAppIdArray = buildAppIdArray(mPowerSaveWhitelistAppsExceptIdle,

mPowerSaveWhitelistUserApps, mPowerSaveWhitelistExceptIdleAppIds);

mPowerSaveWhitelistAllAppIdArray = buildAppIdArray(mPowerSaveWhitelistApps,

mPowerSaveWhitelistUserApps, mPowerSaveWhitelistAllAppIds);

mPowerSaveWhitelistUserAppIdArray = buildAppIdArray(null,

mPowerSaveWhitelistUserApps, mPowerSaveWhitelistUserAppIds);

if (mLocalActivityManager != null) {

mLocalActivityManager.setDeviceIdleWhitelist(

mPowerSaveWhitelistAllAppIdArray, mPowerSaveWhitelistExceptIdleAppIdArray);

}

if (mLocalPowerManager != null) {

if (DEBUG) {

Slog.d(TAG, "Setting wakelock whitelist to "

+ Arrays.toString(mPowerSaveWhitelistAllAppIdArray));

}

mLocalPowerManager.setDeviceIdleWhitelist(mPowerSaveWhitelistAllAppIdArray);

}

passWhiteListsToForceAppStandbyTrackerLocked();//AppStateTracker，dumpsys jobscheduler或者alarm可以查询到，里面还有当APP状态或者名单等发生变化的listener回调

}

1. 执行完onStart()方法后，就开始执行最后一个生命周期方法onBootPhase()

public void onBootPhase(int phase) {

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

synchronized (this) {

mAlarmManager = mInjector.getAlarmManager();

mLocalAlarmManager = getLocalService(AlarmManagerInternal.class);

mBatteryStats = BatteryStatsService.getService();

mLocalActivityManager = getLocalService(ActivityManagerInternal.class);

mLocalActivityTaskManager = getLocalService(ActivityTaskManagerInternal.class);

mLocalPowerManager = getLocalService(PowerManagerInternal.class);

mPowerManager = mInjector.getPowerManager();

mActiveIdleWakeLock = mPowerManager.newWakeLock(PowerManager.PARTIAL\_WAKE\_LOCK,

"deviceidle\_maint");

mActiveIdleWakeLock.setReferenceCounted(false);//非计数锁，只要执行一次release()就能释所有非计数锁

mGoingIdleWakeLock = mPowerManager.newWakeLock(PowerManager.PARTIAL\_WAKE\_LOCK,

"deviceidle\_going\_idle");

mGoingIdleWakeLock.setReferenceCounted(true);//设置wakelock锁为计数锁，一次申请对应一次释放

mNetworkPolicyManager = INetworkPolicyManager.Stub.asInterface(

ServiceManager.getService(Context.NETWORK\_POLICY\_SERVICE));

mNetworkPolicyManagerInternal = getLocalService(NetworkPolicyManagerInternal.class);

mSensorManager = mInjector.getSensorManager();

if (mUseMotionSensor) {

int sigMotionSensorId = getContext().getResources().getInteger(

com.android.internal.R.integer.config\_autoPowerModeAnyMotionSensor);//默认是0

if (sigMotionSensorId > 0) { //可用于自动省电模式时的传感器id，0表示没有可用传感器

mMotionSensor = mSensorManager.getDefaultSensor(sigMotionSensorId, true);

}

if (mMotionSensor == null && getContext().getResources().getBoolean(

com.android.internal.R.bool.config\_autoPowerModePreferWristTilt)) {//默认是false

mMotionSensor = mSensorManager.getDefaultSensor(//获取一个WristTilt传感器(手腕抖动)

Sensor.TYPE\_WRIST\_TILT\_GESTURE, true);

}

if (mMotionSensor == null) {

//如果以上条件都不满足，则获取一个SMD传感器.

mMotionSensor = mSensorManager.getDefaultSensor(Sensor.TYPE\_SIGNIFICANT\_MOTION, true);

}

}

//是否在进入Doze模式时预先获取位置，默认是true

if (getContext().getResources().getBoolean(

com.android.internal.R.bool.config\_autoPowerModePrefetchLocation)) {

mLocationRequest = new LocationRequest()

.setQuality(LocationRequest.ACCURACY\_FINE)

.setInterval(0)

.setFastestInterval(0)

.setNumUpdates(1);

}

//感觉看着像针对其他设备定义的约束条件，举个例子比如有个最小状态minState

mConstraintController = mInjector.getConstraintController(mHandler, getLocalService(LocalService.class));

if (mConstraintController != null) {

mConstraintController.start();

}

//自动省电模式下传感器检测的阈值度

float angleThreshold = getContext().getResources().getInteger(

com.android.internal.R.integer.config\_autoPowerModeThresholdAngle) / 100f;

//用于检测设备是否已静止

mAnyMotionDetector = mInjector.getAnyMotionDetector(mHandler, mSensorManager, this,

angleThreshold);

mAppStateTracker.onSystemServicesReady();//开始更新APP状态

//用于Doze状态发生改变时发送广播

mIdleIntent = new Intent(PowerManager.ACTION\_DEVICE\_IDLE\_MODE\_CHANGED);

mIdleIntent.addFlags(Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY | Intent.FLAG\_RECEIVER\_FOREGROUND);

//用于当轻度Doze状态发生改变时发送广播

mLightIdleIntent = new Intent(PowerManager.ACTION\_LIGHT\_DEVICE\_IDLE\_MODE\_CHANGED);

mLightIdleIntent.addFlags(Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY | Intent.FLAG\_RECEIVER\_FOREGROUND);

//注册监听电池状态改变的广播

IntentFilter filter = new IntentFilter();

filter.addAction(Intent.ACTION\_BATTERY\_CHANGED);

getContext().registerReceiver(mReceiver, filter);

//注册监听卸载应用的广播

filter = new IntentFilter();

filter.addAction(Intent.ACTION\_PACKAGE\_REMOVED);

filter.addDataScheme("package");

getContext().registerReceiver(mReceiver, filter);

//注册监听网络连接改变的广播

filter = new IntentFilter();

filter.addAction(ConnectivityManager.CONNECTIVITY\_ACTION);

getContext().registerReceiver(mReceiver, filter);

//注册监听亮灭屏的广播

filter = new IntentFilter();

filter.addAction(Intent.ACTION\_SCREEN\_OFF);

filter.addAction(Intent.ACTION\_SCREEN\_ON);

getContext().registerReceiver(mInteractivityReceiver, filter);

//将适用于所有情况的白名单列表通知给AMS、PMS、AlarmManagerService

mLocalActivityManager.setDeviceIdleWhitelist(

mPowerSaveWhitelistAllAppIdArray, mPowerSaveWhitelistExceptIdleAppIdArray);

mLocalPowerManager.setDeviceIdleWhitelist(mPowerSaveWhitelistAllAppIdArray);

mLocalPowerManager.registerLowPowerModeObserver(ServiceType.QUICK\_DOZE,

state -> {

synchronized (DeviceIdleController.this) {

updateQuickDozeFlagLocked(state.batterySaverEnabled);

}

});

updateQuickDozeFlagLocked(

mLocalPowerManager.getLowPowerState(//省电模式是否使能，以便快速的进入DOZE

ServiceType.QUICK\_DOZE).batterySaverEnabled);

mLocalActivityTaskManager.registerScreenObserver(mScreenObserver);

passWhiteListsToForceAppStandbyTrackerLocked();//更新APP状态

updateInteractivityLocked();//根据屏幕的亮灭来进行状态的更新

}

updateConnectivityState(null);//更新网络连接状态

}

}

updateInteractivityLocked()用于更新交互状态

void updateInteractivityLocked() {

// The interactivity state from the power manager tells us whether the display is

// in a state that we need to keep things running so they will update at a normal frequency.

boolean screenOn = mPowerManager.isInteractive();//获取设备是否处于交互状态

if (DEBUG) Slog.d(TAG, "updateInteractivityLocked: screenOn=" + screenOn);

if (!screenOn && mScreenOn) {//表示当前不处于交互状态且上次处于交互状态

mScreenOn = false;

if (!mForceIdle) {//是否强制进入Idle

becomeInactiveIfAppropriateLocked();//进入Idle模式的入口方法

}

} else if (screenOn) {

mScreenOn = true;

if (!mForceIdle && (!mScreenLocked || !mConstants.WAIT\_FOR\_UNLOCK)) {

mActiveReason = ACTIVE\_REASON\_SCREEN;

becomeActiveLocked("screen", Process.myUid());//退出Idle模式

}

}

}

# DeviceIdleController的状态变化

对于充电状态，在onBootPhase函数中已经提到，DeviceIdleController监听了ACTION\_BATTERY\_CHANGED广播

private final BroadcastReceiver mReceiver = new BroadcastReceiver() {

@Override public void onReceive(Context context, Intent intent) {

switch (intent.getAction()) {

…

case Intent.ACTION\_BATTERY\_CHANGED: {

boolean present = intent.getBooleanExtra(BatteryManager.EXTRA\_PRESENT, true);

boolean plugged = intent.getIntExtra(BatteryManager.EXTRA\_PLUGGED, 0) != 0;

synchronized (DeviceIdleController.this) {

updateChargingLocked(present && plugged);

}

} break;

…

}

}

};

void updateChargingLocked(boolean charging) {

if (DEBUG) Slog.i(TAG, "updateChargingLocked: charging=" + charging);

if (!charging && mCharging) {

mCharging = false; //从充电状态变为不充电状态

if (!mForceIdle) { //mForceIdle值一般为false，dumpsys deviceidle force-idle

becomeInactiveIfAppropriateLocked(); //判断是否进入Doze模式

}

} else if (charging) {

mCharging = charging; //进入充电状态

if (!mForceIdle) {

becomeActiveLocked("charging", Process.myUid());

}

}

}

在之前的分析中，我们知道updateInteractivityLocked(判断当前屏幕是否亮着)和updateChargingLocked(判断是否在充电)，如果屏幕熄灭或者没在充电的话，则会调用becomeInactiveIfAppropriateLocked开始准备进入Doze状态。

要进入Doze流程，就是调用这个函数，首先要保证屏幕灭屏然后没有充电。这里还有mDeepEnable和mLightEnable前面说过是在配置中定义的，一般默认是关闭(也就是不开Doze模式)。这里mLightEnabled是对应禁止wakelock持锁的，禁止网络。而mDeepEnabled对应是检测设备是否静止，除了禁止wakelock、禁止网络、还会屏蔽alarm。

void becomeInactiveIfAppropriateLocked() {

verifyAlarmStateLocked();

final boolean isScreenBlockingInactive = mScreenOn && (!mConstants.WAIT\_FOR\_UNLOCK || !mScreenLocked);

if (DEBUG) {

Slog.d(TAG, "becomeInactiveIfAppropriateLocked():"

+ " isScreenBlockingInactive=" + isScreenBlockingInactive

+ " (mScreenOn=" + mScreenOn

+ ", WAIT\_FOR\_UNLOCK=" + mConstants.WAIT\_FOR\_UNLOCK

+ ", mScreenLocked=" + mScreenLocked + ")"

+ " mCharging=" + mCharging

+ " mForceIdle=" + mForceIdle

);

}

if (!mForceIdle && (mCharging || isScreenBlockingInactive)) {

return;

}

// Become inactive and determine if we will ultimately go idle.

if (mDeepEnabled) { // mDeepEnabled默认没有使能，但是我们需要分析的使能的情况下

if (mQuickDozeActivated) {// false，快速进入doze，默认没有使能，米8也没有使能，快速进入doze就是少了中间各种状态的转换，直接进入deep doze模式，我们暂时先不看这块，重点关注在通常的情况

if (mState == STATE\_QUICK\_DOZE\_DELAY || mState == STATE\_IDLE

|| mState == STATE\_IDLE\_MAINTENANCE) {

// Already "idling". Don't want to restart the process.

// mLightState can't be LIGHT\_STATE\_ACTIVE if mState is any of these 3

// values, so returning here is safe.

return;

}

if (DEBUG) {

Slog.d(TAG, "Moved from " + stateToString(mState) + " to STATE\_QUICK\_DOZE\_DELAY");

}

mState = STATE\_QUICK\_DOZE\_DELAY;

// Make sure any motion sensing or locating is stopped.

resetIdleManagementLocked();

// Wait a small amount of time in case something (eg: background service from

// recently closed app) needs to finish running.

scheduleAlarmLocked(mConstants.QUICK\_DOZE\_DELAY\_TIMEOUT, false);

EventLogTags.writeDeviceIdle(mState, "no activity");

} else if (mState == STATE\_ACTIVE) {// 将状态设置成STATE\_INACTIVE

mState = STATE\_INACTIVE;

if (DEBUG) Slog.d(TAG, "Moved from STATE\_ACTIVE to STATE\_INACTIVE");

resetIdleManagementLocked();// 重置事件

long delay = mInactiveTimeout;// dumpsys deviceidle | grep inactive\_to查看，米8是15分钟

// 除了mActiveReason为ACTIVE\_REASON\_MOTION之外，都需要乘以mPreIdleFactor

if (shouldUseIdleTimeoutFactorLocked()) {

delay = (long) (mPreIdleFactor \* delay);

}

// 开始检测是否可以进入Doze模式的Idle状态

scheduleAlarmLocked(delay, false);

EventLogTags.writeDeviceIdle(mState, "no activity");

}

}

if (mLightState == LIGHT\_STATE\_ACTIVE && mLightEnabled) {// mLightEnabled默认没有使能，但是我们需要分析的使能的情况下

mLightState = LIGHT\_STATE\_INACTIVE;

if (DEBUG) Slog.d(TAG, "Moved from LIGHT\_STATE\_ACTIVE to LIGHT\_STATE\_INACTIVE");

resetLightIdleManagementLocked();// 重置事件

scheduleLightAlarmLocked(mConstants.LIGHT\_IDLE\_AFTER\_INACTIVE\_TIMEOUT);

EventLogTags.writeDeviceIdleLight(mLightState, "no activity");

}

}

## Light doze – scheduleLightAlarmLocked

light doze模式下会禁止网络、wakelock，但是不会禁止alarm，到时间后调用mLightAlarmListener回调

void scheduleLightAlarmLocked(long delay) {

if (DEBUG) Slog.d(TAG, "scheduleLightAlarmLocked(" + delay + ")");

mNextLightAlarmTime = SystemClock.elapsedRealtime() + delay;

mAlarmManager.set(AlarmManager.ELAPSED\_REALTIME\_WAKEUP,

mNextLightAlarmTime, "DeviceIdleController.light", mLightAlarmListener, mHandler);

}

private final AlarmManager.OnAlarmListener mLightAlarmListener

= new AlarmManager.OnAlarmListener() {

@Override

public void onAlarm() {

synchronized (DeviceIdleController.this) {

stepLightIdleStateLocked("s:alarm");

}

}

};

分析stepLightIdleStateLocked之前，我们先来看下light doze模式相关值

private int mLightState; //轻度doze模式当前状态

static final int LIGHT\_STATE\_ACTIVE = 0; //设备处于活动状态

static final int LIGHT\_STATE\_INACTIVE = 1; //设备处于不活动状态

static final int LIGHT\_STATE\_PRE\_IDLE = 3; //设备进入light空闲状态前，需要等待完成必要操作

static final int LIGHT\_STATE\_IDLE = 4; //设备处于light空闲状态，trying to stay asleep as much as possible

static final int LIGHT\_STATE\_WAITING\_FOR\_NETWORK = 5; //设备处于light空闲状态，要进入维护状态，先等待网络连接

static final int LIGHT\_STATE\_IDLE\_MAINTENANCE = 6; //设备处于light空闲状态，暂时退出idle做regular maintenance

static final int LIGHT\_STATE\_OVERRIDE = 7; //设备处于轻空闲状态被覆盖，开始进入深度doze模式

(1) LIGHT\_STATE\_INACTIVE -----通过alarm处理-----> LIGHT\_STATE\_PRE\_IDLE

(2) LIGHT\_STATE\_PRE\_IDLE，会发送MSG\_REPORT\_IDLE\_ON\_LIGHT，这个消息的处理会禁止网络、禁止wakelock，-----通过alarm处理-----> LIGHT\_STATE\_IDLE

(3) LIGHT\_STATE\_IDLE -----通过alarm处理-----> LIGHT\_STATE\_WAITING\_FOR\_NETWORK

(4) LIGHT\_STATE\_WAITING\_FOR\_NETWORK，会先退出Doze状态(这个时候网络、wakelock恢复) -----通过alarm处理-----> LIGHT\_STATE\_IDLE\_MAINTENANCE

(5) LIGHT\_STATE\_IDLE\_MAINTENANCE重复(2)的操作

void stepLightIdleStateLocked(String reason) {

if (mLightState == LIGHT\_STATE\_OVERRIDE) {

//设备处于轻空闲状态被覆盖，开始进入深度doze模式，没什么可做的

return;

}

if (DEBUG) Slog.d(TAG, "stepLightIdleStateLocked: mLightState=" + mLightState);

EventLogTags.writeDeviceIdleLightStep();

switch (mLightState) {

case LIGHT\_STATE\_INACTIVE:

mCurIdleBudget = mConstants.LIGHT\_IDLE\_MAINTENANCE\_MIN\_BUDGET;

// Reset the upcoming idle delays.

mNextLightIdleDelay = mConstants.LIGHT\_IDLE\_TIMEOUT;

mMaintenanceStartTime = 0;

if (!isOpsInactiveLocked()) {

// We have some active ops going on... give them a chance to finish before going in to our first idle.

mLightState = LIGHT\_STATE\_PRE\_IDLE;

EventLogTags.writeDeviceIdleLight(mLightState, reason);

scheduleLightAlarmLocked(mConstants.LIGHT\_PRE\_IDLE\_TIMEOUT);

break;

}

// Nothing active, fall through to immediately idle.

case LIGHT\_STATE\_PRE\_IDLE:

case LIGHT\_STATE\_IDLE\_MAINTENANCE:

if (mMaintenanceStartTime != 0) {

long duration = SystemClock.elapsedRealtime() - mMaintenanceStartTime;

if (duration < mConstants.LIGHT\_IDLE\_MAINTENANCE\_MIN\_BUDGET) {

// We didn't use up all of our minimum budget; add this to the reserve.

mCurIdleBudget += (mConstants.LIGHT\_IDLE\_MAINTENANCE\_MIN\_BUDGET-duration);

} else {

// We used more than our minimum budget; this comes out of the reserve.

mCurIdleBudget -= (duration-mConstants.LIGHT\_IDLE\_MAINTENANCE\_MIN\_BUDGET);

}

}

mMaintenanceStartTime = 0;

scheduleLightAlarmLocked(mNextLightIdleDelay);

mNextLightIdleDelay = Math.min(mConstants.LIGHT\_MAX\_IDLE\_TIMEOUT,

(long)(mNextLightIdleDelay \* mConstants.LIGHT\_IDLE\_FACTOR));

if (mNextLightIdleDelay < mConstants.LIGHT\_IDLE\_TIMEOUT) {

mNextLightIdleDelay = mConstants.LIGHT\_IDLE\_TIMEOUT;

}

if (DEBUG) Slog.d(TAG, "Moved to LIGHT\_STATE\_IDLE.");

mLightState = LIGHT\_STATE\_IDLE;

EventLogTags.writeDeviceIdleLight(mLightState, reason);

addEvent(EVENT\_LIGHT\_IDLE, null);

mGoingIdleWakeLock.acquire();

mHandler.sendEmptyMessage(MSG\_REPORT\_IDLE\_ON\_LIGHT);

break;

case LIGHT\_STATE\_IDLE:

case LIGHT\_STATE\_WAITING\_FOR\_NETWORK:

if (mNetworkConnected || mLightState == LIGHT\_STATE\_WAITING\_FOR\_NETWORK) {

// We have been idling long enough, now it is time to do some work.

mActiveIdleOpCount = 1;

mActiveIdleWakeLock.acquire();

mMaintenanceStartTime = SystemClock.elapsedRealtime();

if (mCurIdleBudget < mConstants.LIGHT\_IDLE\_MAINTENANCE\_MIN\_BUDGET) {

mCurIdleBudget = mConstants.LIGHT\_IDLE\_MAINTENANCE\_MIN\_BUDGET;

} else if (mCurIdleBudget > mConstants.LIGHT\_IDLE\_MAINTENANCE\_MAX\_BUDGET) {

mCurIdleBudget = mConstants.LIGHT\_IDLE\_MAINTENANCE\_MAX\_BUDGET;

}

scheduleLightAlarmLocked(mCurIdleBudget);

if (DEBUG) Slog.d(TAG,

"Moved from LIGHT\_STATE\_IDLE to LIGHT\_STATE\_IDLE\_MAINTENANCE.");

mLightState = LIGHT\_STATE\_IDLE\_MAINTENANCE;

EventLogTags.writeDeviceIdleLight(mLightState, reason);

addEvent(EVENT\_LIGHT\_MAINTENANCE, null);

mHandler.sendEmptyMessage(MSG\_REPORT\_IDLE\_OFF); //醒一下(开启网络、恢复wakelock)

} else {

// We'd like to do maintenance, but currently don't have network

// connectivity... let's try to wait until the network comes back.

// We'll only wait for another full idle period, however, and then give up.

scheduleLightAlarmLocked(mNextLightIdleDelay);

if (DEBUG) Slog.d(TAG, "Moved to LIGHT\_WAITING\_FOR\_NETWORK.");

mLightState = LIGHT\_STATE\_WAITING\_FOR\_NETWORK;

EventLogTags.writeDeviceIdleLight(mLightState, reason);

}

break;

}

}

## Deep doze – scheduleAlarmLocked

分析stepIdleStateLocked之前，我们先来看下deep doze模式相关值

private int mState;

static final int STATE\_ACTIVE = 0; //设备处于活动状态

static final int STATE\_INACTIVE = 1; //Device is inactive (screen off, no motion) and we are waiting to for idle.

static final int STATE\_IDLE\_PENDING = 2; //Device is past the initial inactive period, and waiting for the next idle period.

static final int STATE\_SENSING = 3; //设备当前正在感应运动

static final int STATE\_LOCATING = 4; //设备当前正在查找位置(可能仍在感知)

static final int STATE\_IDLE = 5; //Device is in the idle state, trying to stay asleep as much as possible.

static final int STATE\_IDLE\_MAINTENANCE = 6;//设备在deep idle状态，但是要临时退出该状态处理常规maintenance任务

/\* Device is inactive and should go straight into idle (foregoing motion and location

monitoring), but allow some time for current work to complete first. \*/

static final int STATE\_QUICK\_DOZE\_DELAY = 7;

下面我们再来看deep idle模式，这个模式除了禁止网络、wakelock还会禁止alarm。

becomeInactiveIfAppropriateLocked –> scheduleAlarmLocked

void scheduleAlarmLocked(long delay, boolean idleUntil) {

if (DEBUG) Slog.d(TAG, "scheduleAlarmLocked(" + delay + ", " + idleUntil + ")");

if (mUseMotionSensor && mMotionSensor == null

&& mState != STATE\_QUICK\_DOZE\_DELAY

&& mState != STATE\_IDLE

&& mState != STATE\_IDLE\_MAINTENANCE) {

//在onBootPhase时，获取过位置检测传感器

//如果终端没有配置位置检测传感器，那么终端永远不会进入到真正的Doze idle状态

return;

}

mNextAlarmTime = SystemClock.elapsedRealtime() + delay;//获取从设备boot后经历的时间值 + delay

if (idleUntil) {

DeviceIdleController是使用一个特殊的私有方法(AlarmManager.setIdleUntil())来注册下一个唤醒alarm。

当AlarmManagerService 看到它时，所有的标准应用alarm都强制进入到一个等待状态直到下一个DeviceIdleController alarm触发。

mAlarmManager.setIdleUntil(AlarmManager.ELAPSED\_REALTIME\_WAKEUP,

mNextAlarmTime, "DeviceIdleController.deep", mDeepAlarmListener, mHandler);

} else {

mAlarmManager.set(AlarmManager.ELAPSED\_REALTIME\_WAKEUP,

mNextAlarmTime, "DeviceIdleController.deep", mDeepAlarmListener, mHandler);

}

}

需要注意的是，DeviceIdleController一直在监控屏幕状态和充电状态，一旦不满足Doze模式的条件，前面提到的becomeActiveLocked函数就会被调用。mAlarmManager设置的定时唤醒事件将被取消掉，mDeepAlarmListener的onAlarm函数不会被调用。

我们看下mDeepAlarmListener.onAlarm，

final AlarmManager.OnAlarmListener mDeepAlarmListener

= new AlarmManager.OnAlarmListener() {

@Override

public void onAlarm() {

synchronized (DeviceIdleController.this) {

stepIdleStateLocked("s:alarm");

}

}

};

void stepIdleStateLocked(String reason) {

if (DEBUG) Slog.d(TAG, "stepIdleStateLocked: mState=" + mState);

EventLogTags.writeDeviceIdleStep();

//个人觉得，下面这段代码，是针对Idle状态设计的

//如果在Idle状态收到Alarm，那么将先唤醒终端，然后重新判断是否需要进入Idle态

//在介绍Doze模式原理时提到过，若应用调用AlarmManager的一些指定接口，仍然可以在Idle状态进行工作

final long now = SystemClock.elapsedRealtime();

if ((now+mConstants.MIN\_TIME\_TO\_ALARM) > mAlarmManager.getNextWakeFromIdleTime()) {

if (mState != STATE\_ACTIVE) {

mActiveReason = ACTIVE\_REASON\_ALARM;

becomeActiveLocked("alarm", Process.myUid());

becomeInactiveIfAppropriateLocked();

}

return;

}

if (mNumBlockingConstraints != 0 && !mForceIdle) {

// We have some constraints from other parts of the system server preventing

// us from moving to the next state.

if (DEBUG) {

Slog.i(TAG, "Cannot step idle state. Blocked by: " + mConstraints.values().stream()

.filter(x -> x.active)

.map(x -> x.name)

.collect(Collectors.joining(",")));

}

return;

}

//以下是Doze模式的状态转变相关的代码

(1) STATE\_INACTIVE -> STATE\_IDLE\_PENDING

(2) STATE\_IDLE\_PENDING –> STATE\_SENSING

(3) STATE\_SENSING -> STATE\_LOCATING

(4) STATE\_LOCATING -> STATE\_IDLE\_MAINTENANCE

(5) STATE\_IDLE\_MAINTENANCE -> STATE\_IDLE

switch (mState) {

case STATE\_INACTIVE:

//已经很长时间没有活动了，是时候开始looking for motion and sleep some more while doing so

//注册一个mMotionListener, 检测是否移动, 如果检测到移动，将重新进入到ACTIVE状态

startMonitoringMotionLocked();

long delay = mConstants.IDLE\_AFTER\_INACTIVE\_TIMEOUT;

if (shouldUseIdleTimeoutFactorLocked()) {

delay = (long) (mPreIdleFactor \* delay);

}

scheduleAlarmLocked(delay, false);

moveToStateLocked(STATE\_IDLE\_PENDING, reason);

break;

case STATE\_IDLE\_PENDING:

//保持息屏、未充电、静止状态，经过xx min后，进入此分支

moveToStateLocked(STATE\_SENSING, reason);

cancelLocatingLocked();

mLocated = false;

mLastGenericLocation = null;

mLastGpsLocation = null;

updateActiveConstraintsLocked();

//检查是否发生了移动

if (mUseMotionSensor && mAnyMotionDetector.hasSensor()) {

scheduleSensingTimeoutAlarmLocked(mConstants.SENSING\_TIMEOUT);

mNotMoving = false;

mAnyMotionDetector.checkForAnyMotion();

break;

} else if (mNumBlockingConstraints != 0) {

cancelAlarmLocked();

break;

}

mNotMoving = true;

// Otherwise, fall through and check this off the list of requirements.

case STATE\_SENSING:

//目前个人查询之后，有很多地方会调用到stepIdleStateLocked，比如从运动到静止将会调用它，然后进入此分支，开始获取定位信息

cancelSensingTimeoutAlarmLocked();

moveToStateLocked(STATE\_LOCATING, reason);

scheduleAlarmLocked(mConstants.LOCATING\_TIMEOUT, false);

LocationManager locationManager = mInjector.getLocationManager();

if (locationManager != null //网络定位

&& locationManager.getProvider(LocationManager.NETWORK\_PROVIDER) != null) {

locationManager.requestLocationUpdates(mLocationRequest,

mGenericLocationListener, mHandler.getLooper());

mLocating = true;

} else {

mHasNetworkLocation = false;

}

if (locationManager != null //GPS定位

&& locationManager.getProvider(LocationManager.GPS\_PROVIDER) != null) {

mHasGps = true;

locationManager.requestLocationUpdates(LocationManager.GPS\_PROVIDER, 1000, 5,

mGpsLocationListener, mHandler.getLooper());

mLocating = true;

} else {

mHasGps = false;

}

// If we have a location provider, we're all set, the listeners will move state forward.

if (mLocating) { //无法定位则直接进入下一个case

break;

}

// Otherwise, we have to move from locating into idle maintenance.

case STATE\_LOCATING:

//停止定位和运动检测，直接进入到STATE\_IDLE\_MAINTENANCE

cancelAlarmLocked();

cancelLocatingLocked();

mAnyMotionDetector.stop();

// Intentional fallthrough -- time to go into IDLE state.

case STATE\_QUICK\_DOZE\_DELAY:

// Reset the upcoming idle delays.

mNextIdlePendingDelay = mConstants.IDLE\_PENDING\_TIMEOUT; //米8，5m0s0ms

mNextIdleDelay = mConstants.IDLE\_TIMEOUT; //米8，1h0m0s0ms

// Everything is in place to go into IDLE state.

case STATE\_IDLE\_MAINTENANCE:

scheduleAlarmLocked(mNextIdleDelay, true);

if (DEBUG) Slog.d(TAG, "Moved to STATE\_IDLE. Next alarm in " + mNextIdleDelay + " ms.");

//退出周期逐步递增，每次乘2

mNextIdleDelay = (long)(mNextIdleDelay \* mConstants.IDLE\_FACTOR);

if (DEBUG) Slog.d(TAG, "Setting mNextIdleDelay = " + mNextIdleDelay);

mIdleStartTime = SystemClock.elapsedRealtime();

//周期有最大值6h

mNextIdleDelay = Math.min(mNextIdleDelay, mConstants.MAX\_IDLE\_TIMEOUT);

if (mNextIdleDelay < mConstants.IDLE\_TIMEOUT) {

mNextIdleDelay = mConstants.IDLE\_TIMEOUT;

}

moveToStateLocked(STATE\_IDLE, reason);

if (mLightState != LIGHT\_STATE\_OVERRIDE) {

mLightState = LIGHT\_STATE\_OVERRIDE;

cancelLightAlarmLocked();

}

addEvent(EVENT\_DEEP\_IDLE, null);

mGoingIdleWakeLock.acquire();

//通知PMS、NetworkPolicyManagerService等Doze模式开启，即进入Idle状态

//此时PMS disable一些非白名单WakeLock；NetworkPolicyManagerService开始限制一些应用的网络访问

//消息处理的具体流程比较直观，此处不再深入分析

mHandler.sendEmptyMessage(MSG\_REPORT\_IDLE\_ON);

break;

case STATE\_IDLE:

//进入到这个case时，本次的Idle状态暂时结束，开启maintenance window

//We have been idling long enough, now it is time to do some work.

mActiveIdleOpCount = 1;

mActiveIdleWakeLock.acquire();

//定义重新进入Idle的时间为5min(也就是手机可处于Maintenance window的时间)

scheduleAlarmLocked(mNextIdlePendingDelay, false);

if (DEBUG) Slog.d(TAG, "Moved from STATE\_IDLE to STATE\_IDLE\_MAINTENANCE. " +

"Next alarm in " + mNextIdlePendingDelay + " ms.");

mMaintenanceStartTime = SystemClock.elapsedRealtime();

//调整mNextIdlePendingDelay，乘2(最大为10min)

mNextIdlePendingDelay = Math.min(mConstants.MAX\_IDLE\_PENDING\_TIMEOUT,

(long)(mNextIdlePendingDelay \* mConstants.IDLE\_PENDING\_FACTOR));

if (mNextIdlePendingDelay < mConstants.IDLE\_PENDING\_TIMEOUT) {

mNextIdlePendingDelay = mConstants.IDLE\_PENDING\_TIMEOUT;

}

moveToStateLocked(STATE\_IDLE\_MAINTENANCE, reason);

addEvent(EVENT\_DEEP\_MAINTENANCE, null);

//通知PMS等暂时退出了Idle状态，可以进行一些工作

//此时PMS enable一些非白名单WakeLock；NetworkPolicyManagerService开始允许应用的网络访问

//消息处理的具体流程比较直观，此处不再深入分析

mHandler.sendEmptyMessage(MSG\_REPORT\_IDLE\_OFF);

break;

}

}

## 小结

Light doze和Deep doze的状态转换图就先不画了，通过代码来看就是通过alarm的回调和message收发，再结合各种的条件判断进行状态的赋值以及在该状态下禁止非白名单下的网络、wakelock、alarm等等。

# DeviceIdleController相关限制

## DeviceIdleController对网络的限制

我们知道，进入idle模式后调用NetworkPolicyManagerService.java的setDeviceIdleMode()对网络进行限制，最底层是iptables的配置，现在我们简要分析网络限制的流程

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

frameworks/base/services/core/java/com/android/server/net/NetworkPolicyManagerService.java

@Override

public void setDeviceIdleMode(boolean enabled) {

mContext.enforceCallingOrSelfPermission(MANAGE\_NETWORK\_POLICY, TAG);

Trace.traceBegin(Trace.TRACE\_TAG\_NETWORK, "setDeviceIdleMode");

try {

synchronized (mUidRulesFirstLock) {

if (mDeviceIdleMode == enabled) {

return;

}

mDeviceIdleMode = enabled;

mLogger.deviceIdleModeEnabled(enabled);

if (mSystemReady) {

// Device idle change means we need to rebuild rules for all

// known apps, so do a global refresh.

updateRulesForRestrictPowerUL();

}

}

if (enabled) {

EventLogTags.writeDeviceIdleOnPhase("net");

} else {

EventLogTags.writeDeviceIdleOffPhase("net");

}

} finally {

Trace.traceEnd(Trace.TRACE\_TAG\_NETWORK);

}

}

NetworkPolicyManagerService.java-->updateRulesForRestrictPowerUL()

private void updateRulesForRestrictPowerUL() {

Trace.traceBegin(Trace.TRACE\_TAG\_NETWORK, "updateRulesForRestrictPowerUL");

try {

updateRulesForDeviceIdleUL();

updateRulesForPowerSaveUL();

updateRulesForAllAppsUL(TYPE\_RESTRICT\_POWER);

} finally {

Trace.traceEnd(Trace.TRACE\_TAG\_NETWORK);

}

}

到这里，我们需要先看下INetd

system/netd/server/binder/android/net/INetd.aidl

// No specific chain is chosen, use general firewall chain(fw\_input, fw\_output)

const int FIREWALL\_CHAIN\_NONE = 0;

// Specify DOZABLE chain(fw\_dozable) which is used in dozable mode

const int FIREWALL\_CHAIN\_DOZABLE = 1;//Doze下的网络策略，iptables -t filter -L fw\_dozable

// Specify STANDBY chain(fw\_standby) which is used in standby mode

const int FIREWALL\_CHAIN\_STANDBY = 2;//前后台的网络策略，iptables -t filter -L fw\_standby，处在白名单的放行，否则匹配到最后一条默认规则，被丢弃

// Specify POWERSAVE chain(fw\_powersave) which is used in power save mode

const int FIREWALL\_CHAIN\_POWERSAVE = 3;//省电模式下的网络策略，iptables -t filter -L fw\_powersave

我们回到代码

void updateRulesForDeviceIdleUL() {

Trace.traceBegin(Trace.TRACE\_TAG\_NETWORK, "updateRulesForDeviceIdleUL");

try {

updateRulesForWhitelistedPowerSaveUL(mDeviceIdleMode, FIREWALL\_CHAIN\_DOZABLE,

mUidFirewallDozableRules);

} finally {

Trace.traceEnd(Trace.TRACE\_TAG\_NETWORK);

}

}

private void updateRulesForWhitelistedPowerSaveUL(boolean enabled, int chain,

SparseIntArray rules) {

if (enabled) {

// Sync the whitelists before enabling the chain. We don't care about the rules if

// we are disabling the chain.

final SparseIntArray uidRules = rules;

uidRules.clear();

final List<UserInfo> users = mUserManager.getUsers();

for (int ui = users.size() - 1; ui >= 0; ui--) {

UserInfo user = users.get(ui);

updateRulesForWhitelistedAppIds(uidRules, mPowerSaveTempWhitelistAppIds, user.id);

updateRulesForWhitelistedAppIds(uidRules, mPowerSaveWhitelistAppIds, user.id);

if (chain == FIREWALL\_CHAIN\_POWERSAVE) {

updateRulesForWhitelistedAppIds(uidRules,

mPowerSaveWhitelistExceptIdleAppIds, user.id);

}

}

for (int i = mUidState.size() - 1; i >= 0; i--) {

//如果应用在白名单里，这是rule为FIREWALL\_RULE\_ALLOW

if (isProcStateAllowedWhileIdleOrPowerSaveMode(mUidState.valueAt(i))) {

uidRules.put(mUidState.keyAt(i), FIREWALL\_RULE\_ALLOW);

}

}

setUidFirewallRulesUL(chain, uidRules, CHAIN\_TOGGLE\_ENABLE);

} else {

setUidFirewallRulesUL(chain, null, CHAIN\_TOGGLE\_DISABLE);

}

}

我们在看下最后一个参数mUidFirewallDozableRules，是一个SparseIntArray 对象

final SparseIntArray mUidFirewallDozableRules = new SparseIntArray();

在setUidFirewallRule中，就应用的uid和rule 存入进去

private void setUidFirewallRule(int chain, int uid, int rule) {

。。。

try {

if (chain == FIREWALL\_CHAIN\_DOZABLE) {

mUidFirewallDozableRules.put(uid, rule);

} else if (chain == FIREWALL\_CHAIN\_STANDBY) {

mUidFirewallStandbyRules.put(uid, rule);

} else if (chain == FIREWALL\_CHAIN\_POWERSAVE) {

mUidFirewallPowerSaveRules.put(uid, rule);

}

try {

mNetworkManager.setFirewallUidRule(chain, uid, rule);

mLogger.uidFirewallRuleChanged(chain, uid, rule);

} catch (IllegalStateException e) {

Log.wtf(TAG, "problem setting firewall uid rules", e);

} catch (RemoteException e) {

// ignored; service lives in system\_server

}

} finally {

Trace.traceEnd(Trace.TRACE\_TAG\_NETWORK);

}

}

参数分析完后，我们看下setUidFirewallRulesUL()

private void setUidFirewallRulesUL(int chain, @Nullable SparseIntArray uidRules,

@ChainToggleType int toggle) {

if (uidRules != null) {

setUidFirewallRulesUL(chain, uidRules);

}

if (toggle != CHAIN\_TOGGLE\_NONE) {

enableFirewallChainUL(chain, toggle == CHAIN\_TOGGLE\_ENABLE);

}

}

setUidFirewallRulesUL -> setUidFirewallRulesUL –> mNetworkManager.setFirewallUidRules

NetworkManagementService.java-->setFirewallUidRules()

mNetdService.firewallReplaceUidChain("fw\_dozable", true, uids);

## DeviceIdleController对wakelock限制

frameworks/base/services/core/java/com/android/server/power/PowerManagerService.java

setDeviceIdleMode -> setDeviceIdleModeInternal

boolean setDeviceIdleModeInternal(boolean enabled) {

synchronized (mLock) {

if (mDeviceIdleMode == enabled) {

return false;

}

mDeviceIdleMode = enabled;

updateWakeLockDisabledStatesLocked();

}

if (enabled) {

EventLogTags.writeDeviceIdleOnPhase("power");

} else {

EventLogTags.writeDeviceIdleOffPhase("power");

}

return true;

}

private void updateWakeLockDisabledStatesLocked() {

boolean changed = false;

final int numWakeLocks = mWakeLocks.size();

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

final WakeLock wakeLock = mWakeLocks.get(i);

if ((wakeLock.mFlags & PowerManager.WAKE\_LOCK\_LEVEL\_MASK) == PowerManager.PARTIAL\_WAKE\_LOCK) {

//如果mDeviceIdleMode为true，满足条件判断，wakelock的mDisabled为true，代码就不贴了

if (setWakeLockDisabledStateLocked(wakeLock)) {

changed = true;

if (wakeLock.mDisabled) {

// This wake lock is no longer being respected.

notifyWakeLockReleasedLocked(wakeLock);

} else {

notifyWakeLockAcquiredLocked(wakeLock);

}

}

}

}

if (changed) {

mDirty |= DIRTY\_WAKE\_LOCKS;

updatePowerStateLocked();

}

}

updatePowerStateLocked –> updateWakeLockSummaryLocked -> getWakeLockSummaryFlags

private int getWakeLockSummaryFlags(WakeLock wakeLock) {

switch (wakeLock.mFlags & PowerManager.WAKE\_LOCK\_LEVEL\_MASK) {

case PowerManager.PARTIAL\_WAKE\_LOCK:

if (!wakeLock.mDisabled) {//只有其mDisabled为false时，才会让其持cpu锁。

// We only respect this if the wake lock is not disabled.

return WAKE\_LOCK\_CPU;

}

break;

…

return 0;

}

## DeviceIdleController对job的延迟

在Doze状态发生改变时，比如退出LightDoze、DeepDoze等，DeviceIdleJobsController会接收到广播

frameworks/base/services/core/java/com/android/server/job/controllers/DeviceIdleJobsController.java

private final BroadcastReceiver mBroadcastReceiver = new BroadcastReceiver() {

@Override

public void onReceive(Context context, Intent intent) {

switch (intent.getAction()) {

case PowerManager.ACTION\_LIGHT\_DEVICE\_IDLE\_MODE\_CHANGED:

case PowerManager.ACTION\_DEVICE\_IDLE\_MODE\_CHANGED:

updateIdleMode(mPowerManager != null && (mPowerManager.isDeviceIdleMode()

|| mPowerManager.isLightDeviceIdleMode()));

break;