

Zhangjian的博客

不断迭代，精进不已

目录视图

摘要视图

RSS 订阅

个人资料



ZhangJianIsAStark

访问：198326次

积分：4510

等级：BLOG 5

排名：第6626名

原创：250篇 转载：1篇

译文：1篇 评论：117条

文章搜索

博客专栏



设计模式

文章：8篇

阅读：5218



LeetCode练习记录

文章：108篇

阅读：41111



Android源码学习笔记

文章：45篇

阅读：100765

文章分类

Android6.0 源码分析 (5)

LeetCode (108)

Android7.0 基础业务分析 (25)

Android7.0 数据业务分析 (11)

Android7.0 电源管理分析 (7)

Android开发 (14)

Java零碎知识记录 (3)

Android零碎知识记录 (64)

python (5)

开源框架分析 (2)

设计模式 (8)

赠书 | 异步2周年,技术图书免费选

每周荐书：渗透测试、K8s、架构（评论送书）

项目管理+代码托管+文档协作，开发更流畅

Android7.0 Doze模式

标签：android

2016-11-07 15:30

5768人阅读

评论(0)

收藏

举报

分类：Android7.0 电源管理分析（6）

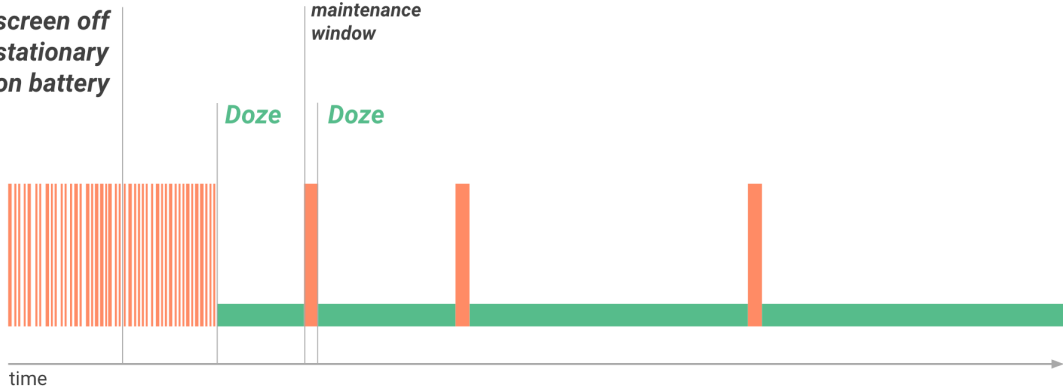
版权声明：转载请注明：http://blog.csdn.net/gaugamela/article

在Android M中，Google就引入了Doze模式。它定义了一种全新的、低能耗的状态。在该状态，后台只有部分任务被允许运行，其它任务都被强制停止。

本篇博客中，我们就来分析一下Android 7.0中Doze模式相关的流程。

一、基本原理

Doze模式可以简单概括为：若判断用户在连续的一段时间内没有使用手机，就延缓终端中APP后台的CPU和网络活动，以达到减少电量消耗的目的。



上面这张图比较经典，基本上说明了Doze模式的含义。图中的横轴表示时间，红色部分表示终端处于唤醒的运行状态，绿色部分就是Doze模式定义的休眠状态。

从图中的描述，我们可以看到：如果一个用户停止充电(on battery: 利用电池供电)，关闭屏幕(screen off)，手机处于静止状态(stationary: 位置没有发生相对移动)，保持以上条件一段时间之后，终端就会进入Doze模式。一旦进入Doze模式，系统就减少(延缓)应用对网络的访问、以及对CPU的占用，来节省电池电量。

如图所示，Doze模式还定义了maintenance window。在maintenance window中，系统允许应用完成它们被延缓的动作，即可以使用CPU资源及访问网络。从图中我们可以看出，当进入Doze模式的条件一直满足时，Doze模式会定期的进入到maintenance window，但进入的间隔越来越长。通过这种方式，Doze模式可以使终端处于较长时间的休眠状态。

需要注意的是：一旦Doze模式的条件不再满足，即用户充电、或打开屏幕、或终端的位置发生了移动，终端就恢复到正常模式。因此，当用户频繁使用手机时，Doze模式几乎是没有什么实际用处的。

具体来讲，当终端处于Doze模式时，进行了以下操作：

1、暂停网络访问。

1 of 12

2017年08月23日 15:42

文章存档

2017年08月

(1)

2017年07月

(2)

2017年06月

(3)

2017年04月

(23)

2017年03月

(28)

展开

阅读排行

Android7.0 init进程源码分析

(5783)

Android7.0 Doze模式

(5749)

Android7.0 PhoneApp的

(4316)

Android 7.0 ActivityManager

(4063)

Android7.0 数据拨号前的

(3806)

Android7.0 数据业务长连

(3783)

Android 7.0 ActivityManager

(3744)

Android 7.0 ActivityManager

(3725)

Android连接指定Wifi的方法

(3587)

Android7.0 PowerManager

(3554)

最新评论

Android Gradle学习记录4 Gradle
执念墨尘枫: 谢谢分享 acmer
http://blog.csdn.net/dorlife

Android Gradle学习记录4 Gradle
qq_39916266: 为范围法

Android Gradle学习记录4 Gradle
暗夜J使者: 感谢分享，学习了！

Android7.0 init进程源码分析
天一方蓝: 这个是收藏些列啊，赞一个

Android Gradle学习记录4 Gradle
天一方蓝: 学习了，谢谢，分享

Android连接指定Wifi的方法
TOM J 汤姆: @Gaugamela:我明白了，这个代码不适合6.0以上的系统，我换了4.4 和5.0的每次都可以连...

Android连接指定Wifi的方法
ZhangJianIsAStark:
@qq_33756493:这场景太复杂了。。。。你这样问，没人可以回答吧。。。。具体原因，得研究下...

Android连接指定Wifi的方法
TOM J 汤姆: 为什么很多时候重连不上了？有的时候连接很快

Android7.0 PowerManagerService
风雨田: @Gaugamela:打电话亮灭屏时间慢，看看DisplayPowerController 里面up...

Android N数据业务总结
yizhi_cainiao: 整篇文章清晰流畅，大赞博主，希望博主能继续分享！

- 2、系统忽略所有的WakeLock。
- 3、标准的AlarmManager alarms被延缓到下一个maintenance window。
- 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。
- 4、系统不再进行WiFi扫描。
- 5、系统不允许sync adapters运行。
- 6、系统不允许JobScheduler运行。

更多基本信息的描述可以参考：
[What is Doze mode in android 6.0 Marshmallow?](#)

二、DeviceIdleController的初始化

Android中的Doze模式主要由DeviceIdleController来控制。

```
1 public class DeviceIdleController extends SystemService
2     implements AnyMotionDetector.DeviceIdleCallback {
3     .....
4 }
```

可以看出DeviceIdleController继承自SystemService，是一个系统级的服务。
同时，继承了AnyMotionDetector定义的接口，便于检测到终端位置变化后进行回调。

接下来我们看看它的初始化过程。

```
1 private void startOtherServices() {
2     .....
3     mSystemServiceManager.startService(DeviceIdleController.class);
4     .....
5 }
```

如上代码所示，SystemService在startOtherServices中启动了DeviceIdleController，将先后调用DeviceIdleController的构造函数和onStart函数。

1、构造函数

```
1 public DeviceIdleController(Context context) {
2     super(context);
3     //deviceidle.xml用于定义idle模式也能正常工作的非系统应用
4     //一般终端似乎并没有定义deviceidle.xml
5     mConfigFile = new AtomicFile(new File(getSystemDir(), "deviceidle.xml"));
6     mHandler = new MyHandler(BackgroundThread.getHandler().getLooper());
7 }
```

DeviceIdleController的构造函数比较简单，就是在创建data/system/deviceidle.xml对应的file文件，同时创建一个对应于后台线程的handler。

2、onStart

```
1 public void onStart() {
2     final PackageManager pm = getContext().getPackageManager();
3
4     synchronized (this) {
5         //读取配置文件，判断Doze模式是否允许被开启
6         mLightEnabled = mDeepEnabled = getResources().getBoolean(
7             com.android.internal.R.bool.config_enableAutoPowerModes);
8
9         //分析PKMS时提到过，PKMS扫描系统目录的xml，将形成SystemConfig
10        SystemConfig sysConfig = SystemConfig.getInstance();
11
12        //获取除了device Idle模式外，都可以运行的系统应用白名单
13        ArraySet<String> allowPowerExceptIdle = sysConfig.getAllowInPowerSaveExceptIdle();
14        for (int i=0; i<allowPowerExceptIdle.size(); i++) {
15            String pkg = allowPowerExceptIdle.valueAt(i);
16            try {
17                ApplicationInfo ai = pm.getApplicationInfo(pkg,
```

```
18         PackageManager.MATCH_SYSTEM_ONLY);
19         int appid = UserHandle.getAppId(ai.uid);
20         mPowerSaveWhitelistAppsExceptIdle.put(ai.packageName, appid);
21         mPowerSaveWhitelistSystemAppIdsExceptIdle.put(appid, true);
22     } catch (PackageManager.NameNotFoundException e) {
23     }
24 }
25
26 //获取device Idle模式下，也可以运行的系统应用白名单
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
```

除去发布服务外，DeviceIdleController在onStart函数中，主要是读取配置文件更新自己的变量，思路比较清晰。

在这里我们仅跟进一下updateWhitelistAppIdsLocked函数：

```
1 private void updateWhitelistAppIdsLocked() {
2     //构造出除去idle模式外，可运行的app id数组（可认为是系统和普通应用的集合）
3     //mPowerSaveWhitelistAppsExceptIdle从系统目录下的xml得到
4     //mPowerSaveWhitelistUserApps从deviceidle.xml得到，或调用接口加入；
5     //mPowerSaveWhitelistExceptIdleAppIds并未使用
6     mPowerSaveWhitelistExceptIdleAppIdArray = buildAppIdArray(mPowerSaveWhitelistAppsExceptId
```

```
7         mPowerSaveWhitelistUserApps, mPowerSaveWhitelistExceptIdleAppIds);
8
9         //构造不受Doze限制的app id数组（可认为是系统和普通应用的集合）
10        //mPowerSaveWhitelistApps从系统目录下的xml得到
11        //mPowerSaveWhitelistAllAppIds并未使用
12        mPowerSaveWhitelistAllAppIdArray = buildAppIdArray(mPowerSaveWhitelistApps,
13            mPowerSaveWhitelistUserApps, mPowerSaveWhitelistAllAppIds);
14
15        //构造不受Doze限制的app id数组（仅普通应用的集合）、
16        //mPowerSaveWhitelistUserAppIds并未使用
17        mPowerSaveWhitelistUserAppIdArray = buildAppIdArray(null,
18            mPowerSaveWhitelistUserApps, mPowerSaveWhitelistUserAppIds);
19
20        if (mLocalPowerManager != null) {
21            .....
22            //PMS拿到的是：系统和普通应用组成的不受Doze限制的app id数组
23            mLocalPowerManager.setDeviceIdleWhitelist(mPowerSaveWhitelistAllAppIdArra
24        }
25
26        if (mLocalAlarmManager != null) {
27            .....
28            //AlarmManagerService拿到的是：普通应用组成的不受Doze限制的app id数组
29            mLocalAlarmManager.setDeviceIdleUserWhitelist(mPowerSaveWhitelistUserAppId
30        }
31    }
```

updateWhitelistAppIdsLocked主要是将白名单交给PMS和AlarmManagerService。

注意Android区分了系统应用白名单、普通应用白名单等，因此上面进行了一些合并操作。

3、onBootPhase

与PowerManagerService一样，DeviceIdleController在初始化的最后一个阶段需要调用onBootPhase函数：

```
1 public void onBootPhase(int phase) {
2     //在系统PHASE_SYSTEM_SERVICES_READY阶段，进一步完成一些初始化
3     if (phase == PHASE_SYSTEM_SERVICES_READY) {
4         synchronized (this) {
5             //初始化一些变量
6             mAlarmManager = (AlarmManager) getContext().getSystemService(Context.ALARM_SERVICE);
7             .....
8
9             mSensorManager = (SensorManager) getContext().getSystemService(Context.SENSOR_SERVICE);
10            //根据配置文件，利用SensorManager获取对应的传感器，保存到mMotionSensor中
11            .....
12
13            //如果配置文件表明：终端需要预获取位置信息
14            //则构造LocationRequest
15            if (getContext().getResources().getBoolean(
16                com.android.internal.R.bool.config_autoPowerModePrefetchLocation)) {
17                mLocationManager = (LocationManager) getContext().getSystemService(
18                    Context.LOCATION_SERVICE);
19                mLocationRequest = new LocationRequest()
20                    .setQuality(LocationRequest.ACCURACY_FINE)
21                    .setInterval(0)
22                    .setFastestInterval(0)
23                    .setNumUpdates(1);
24            }
25
26            //根据配置文件，得到角度变化的门限
27            float angleThreshold = getContext().getResources().getInteger(
28                com.android.internal.R.integer.config_autoPowerModeThresholdAngle) / 100f;
29            //创建一个AnyMotionDetector，同时将DeviceIdleController注册到其中
30            //当AnyMotionDetector检测到手机变化角度超过门限时，就会回调DeviceIdleController的接口
31            mAnyMotionDetector = new AnyMotionDetector(
32                (PowerManager) getContext().getSystemService(Context.POWER_SERVICE),
33                mHandler, mSensorManager, this, angleThreshold);
34
35            //创建两个常用的Intent，用于通知Doze模式的变化
36            mIdleIntent = new Intent(PowerManager.ACTION_DEVICE_IDLE_MODE_CHANGED);
37            mIdleIntent.addFlags(Intent.FLAG_RECEIVER_REGISTERED_ONLY
```



```
38         | Intent.FLAG_RECEIVER_FOREGROUND);
39         mLightIdleIntent = new Intent(PowerManager.ACTION_LIGHT_DEVICE_IDLE_MODE_CHANGED);
40         mLightIdleIntent.addFlags(Intent.FLAG_RECEIVER_REGISTERED_ONLY
41         | Intent.FLAG_RECEIVER_FOREGROUND);
42
43         //监听ACTION_BATTERY_CHANGED广播（ 电池信息发生改变 ）
44         IntentFilter filter = new IntentFilter();
45         filter.addAction(Intent.ACTION_BATTERY_CHANGED);
46         getContext().registerReceiver(mReceiver, filter);
47
48         //监听ACTION_PACKAGE_REMOVED广播（ 包被移除 ）
49         filter = new IntentFilter();
50         filter.addAction(Intent.ACTION_PACKAGE_REMOVED);
51         filter.addDataScheme("package");
52         getContext().registerReceiver(mReceiver, filter);
53
54         //监听CONNECTIVITY_ACTION广播（ 连接状态发生改变 ）
55         filter = new IntentFilter();
56         filter.addAction(ConnectivityManager.CONNECTIVITY_ACTION);
57         getContext().registerReceiver(mReceiver, filter);
58
59         //重新将白名单信息交给PowerManagerService和AlarmManagerService
60         //这个工作在onStart函数中，已经调用updateWhitelistAppIdsLocked进行过了
61         //到onBootPhase时，重新进行一次，可能：一是为了保险；二是，其它进程可能调用接口，更改
62         mLocalPowerManager.setDeviceIdleWhitelist(mPowerSaveWhitelistAllAppIdArray);
63         mLocalAlarmManager.setDeviceIdleUserWhitelist(mPowerSaveWhitelistUserIdArray);
64
65         //监听屏幕显示相关的变化
66         mDisplayManager.registerDisplayListener(mDisplayListener, null);
67
68         //更新屏幕显示相关的信息
69         updateDisplayLocked();
70     }
71     //更新连接状态相关的信息
72     updateConnectivityState(null);
73 }
74 }
```

从代码可以看出，onBootPhase方法：

主要创建一些本地变量，然后根据配置文件初始化一些传感器，同时注册了一些广播接收器和回调接口，最后更新屏幕显示和连接状态相关的信息。

三、DeviceIdleController定义的状态变化

根据前面提到的Doze模式的原理，我们知道手机进入Doze模式的条件是：未充电、手机位置不发生变化、屏幕熄灭。

因此，在DeviceIdleController中监听了这三个条件对应的状态，以决定终端是真正否进入到Doze模式。

对这三个条件的分析，最终都会进入到DeviceIdleController定义的状态变化流程。

因此我们就以充电状态的变化为例，看看DeviceIdleController进行了哪些处理。其余条件的分析，基本类似。

1、充电状态的处理

对于充电状态，在onBootPhase函数中已经提到，DeviceIdleController监听了ACTION_BATTERY_CHANGED广播：

```
1 .....
2 IntentFilter filter = new IntentFilter();
3 filter.addAction(Intent.ACTION_BATTERY_CHANGED);
4 getContext().registerReceiver(mReceiver, filter);
5 .....
```

我们看看receiver中对应的处理：

```
1 private final BroadcastReceiver mReceiver = new BroadcastReceiver() {
2     @Override public void onReceive(Context context, Intent intent) {
3         switch (intent.getAction()) {
4             .....
5             case Intent.ACTION_BATTERY_CHANGED: {
```

```
6         synchronized (DeviceIdleController.this) {
7             //从广播中得到是否在充电的消息
8             int plugged = intent.getIntExtra("plugged", 0);
9             updateChargingLocked(plugged != 0);
10        }
11    } break;
12    }
13    }
14    };
```

根据上面的代码，可以看出当收到电池信息改变的广播后，DeviceIdleController将得到电源是否在充电的消息，然后调用updateChargingLocked函数进行处理。

```
1 void updateChargingLocked(boolean charging) {
2     .....
3     if (!charging && mCharging) {
4         //从充电状态变为不充电状态
5         mCharging = false;
6         //mForceIdle值一般为false
7         if (!mForceIdle) {
8             //判断是否进入Doze模式
9             becomeInactiveIfAppropriateLocked();
10        }
11    } else if (charging) {
12        //进入充电状态
13        mCharging = charging;
14        if (!mForceIdle) {
15            //手机退出Doze模式
16            becomeActiveLocked("charging", Process.myUid());
17        }
18    }
19 }
```

2、becomeActiveLocked

我们先看看becomeActiveLocked函数：

```
1 //activeReason记录的终端变为active的原因
2 void becomeActiveLocked(String activeReason, int activeUid) {
3     .....
4     if (mState != STATE_ACTIVE || mLightState != STATE_ACTIVE) {
5         .....
6         //1、通知PMS等Doze模式结束
7         scheduleReportActiveLocked(activeReason, activeUid);
8
9         //更新DeviceIdleController本地维护的状态
10        //在DeviceIdleController的onStart函数中，我们已经知道了
11        //初始时，mState和mLightState均为Active状态
12        mState = STATE_ACTIVE;
13        mLightState = LIGHT_STATE_ACTIVE;
14
15        mInactiveTimeout = mConstants.INACTIVE_TIMEOUT;
16        mCurIdleBudget = 0;
17        mMaintenanceStartTime = 0;
18
19        //2、重置一些事件
20        resetIdleManagementLocked();
21        resetLightIdleManagementLocked();
22
23        addEvent(EVENT_NORMAL);
24    }
25 }
```

2.1 scheduleReportActiveLocked

```
1 void scheduleReportActiveLocked(String activeReason, int activeUid) {
2     //发送MSG_REPORT_ACTIVE消息
3     Message msg = mHandler.obtainMessage(MSG_REPORT_ACTIVE, activeUid, 0, activeReason);
4     mHandler.sendMessage(msg);
5 }
```

对应的处理流程：

```
1 .....
2 case MSG_REPORT_ACTIVE: {
3     .....
4     //通知PMS Doze模式结束，
5     //于是PMS将一些Doze模式下，disables的WakeLock重新enable
6     //然后调用updatePowerStateLocked函数更新终端的状态
7     final boolean deepChanged = mLocalPowerManager.setDeviceIdleMode(false);
8     final boolean lightChanged = mLocalPowerManager.setLightDeviceIdleMode(false);
9
10    try {
11        //通过NetworkPolicyManagerService更改Ip-Rule，不再限制终端应用上网
12        mNetworkPolicyManager.setDeviceIdleMode(false);
13        //BSS做好对应的记录
14        mBatteryStats.noteDeviceIdleMode(BatteryStats.DEVICE_IDLE_MODE_OFF,
15            activeReason, activeUid);
16    } catch (RemoteException e) {
17    }
18
19    //发送广播
20    if (deepChanged) {
21        getContext().sendBroadcastAsUser(mIdleIntent, UserHandle.ALL);
22    }
23    if (lightChanged) {
24        getContext().sendBroadcastAsUser(mLightIdleIntent, UserHandle.ALL);
25    }
26 }
27 .....
```

从上面的代码可以看出，scheduleReportActiveLocked函数最主要的工作是：

通知PMS等重新更新终端的状态；

通知NetworkPolicyManagerService不再限制应用上网。

发送Doze模式改变的广播。

2.2 resetIdleManagementLocked

```
1 void resetIdleManagementLocked() {
2     //复位一些状态变量
3     mNextIdlePendingDelay = 0;
4     mNextIdleDelay = 0;
5     mNextLightIdleDelay = 0;
6
7     //停止一些工作，主要是位置检测相关的
8     cancelAlarmLocked();
9     cancelSensingTimeoutAlarmLocked();
10    cancelLocatingLocked();
11    stopMonitoringMotionLocked();
12    mAnyMotionDetector.stop();
13 }
```

从上面的代码可以看出，resetIdleManagementLocked的工作相对简单，就是停止进入Doze模式时启动的一些任务。

3、becomeInactiveIfAppropriateLocked

与becomeActiveLocked函数相比，becomeInactiveIfAppropriateLocked函数较为复杂。

因为调用becomeInactiveIfAppropriateLocked函数时，终端可能只是满足进入Doze模式的条件，离进入真正的Doze模式还有很长的“一段路”需要走。

我们看看becomeInactiveIfAppropriateLocked的代码：

```
1 void becomeInactiveIfAppropriateLocked() {
2     .....
3     //屏幕熄灭，未充电
4     if ((!mScreenOn && !mCharging) || mForceIdle) {
5         // Screen has turned off; we are now going to become inactive and start
6         // waiting to see if we will ultimately go idle.
7         if (mState == STATE_ACTIVE && mDeepEnabled) {
```

```
8         mState = STATE_INACTIVE;
9         .....
10        //重置事件
11        resetIdleManagementLocked();
12
13        //开始检测是否可以进入Doze模式的Idle状态
14        //若终端没有watch feature, mInactiveTimeout时间为30min
15        scheduleAlarmLocked(mInactiveTimeout, false);
16        .....
17    }
18
19    if (mLightState == LIGHT_STATE_ACTIVE && mLightEnabled) {
20        mLightState = LIGHT_STATE_INACTIVE;
21        .....
22        resetLightIdleManagementLocked();
23        scheduleLightAlarmLocked(mConstants.LIGHT_IDLE_AFTER_INACTIVE_TIMEOUT);
24    }
25 }
26 }
```

从上面的代码可以看出，在DeviceIdleState中，用mState和mLightState来衡量终端是否真正进入Doze模式。

我们目前仅关注mState变量的改变情况，mLightState的变化流程可类似分析。

此时，mState的状态为INACTIVE。

3.1 scheduleAlarmLocked

我们跟进一下scheduleAlarmLocked函数：

```
1 void scheduleAlarmLocked(long delay, boolean idleUntil) {
2     if (mMotionSensor == null) {
3         //在onBootPhase时，获取过位置检测传感器
4         //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 ilde状态
5         // If there is no motion sensor on this device, then we won't schedule
6         // alarms, because we can't determine if the device is not moving.
7         return;
8     }
9
10    mNextAlarmTime = SystemClock.elapsedRealtime() + delay;
11    if (idleUntil) {
12        //此时IdleUtil的值为false
13        mAlarmManager.setIdleUntil(AlarmManager.ELAPSED_REALTIME_WAKEUP,
14            mNextAlarmTime, "DeviceIdleController.deep", mDeepAlarmListener, mHandler);
15    } else {
16        //30min后唤醒，调用mDeepAlarmListener的onAlarm函数
17        mAlarmManager.set(AlarmManager.ELAPSED_REALTIME_WAKEUP,
18            mNextAlarmTime, "DeviceIdleController.deep", mDeepAlarmListener, mHandler);
19    }
20 }
```

需要注意的是，DeviceIdleController一直在监控屏幕状态和充电状态，一但不满足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。

因此，我们知道了终端必须保持Doze模式的入口条件长达30min，才会进入mDeepAlarmListener.onAlarm：

```
1 private final AlarmManager.OnAlarmListener mDeepAlarmListener
2     = new AlarmManager.OnAlarmListener() {
3     @Override
4     public void onAlarm() {
5         synchronized (DeviceIdleController.this) {
6             //进入到stepIdleStateLocked函数
7             stepIdleStateLocked("s:alarm");
8         }
9     }
10 };
```

此处没有什么多说的，直接调用了stepIdleStateLocked函数。

需要注意的是stepIdleStateLocked将决定DeviceIdleController状态之间的转移。

这种通过AlarmManager设定唤醒时间，然后通过回调接口来调用stepIdleStateLocked的方式，将被多次使用。

3.2 stepIdleStateLocked

此处没有什么多说的，直接来看stepIdleStateLocked函数：

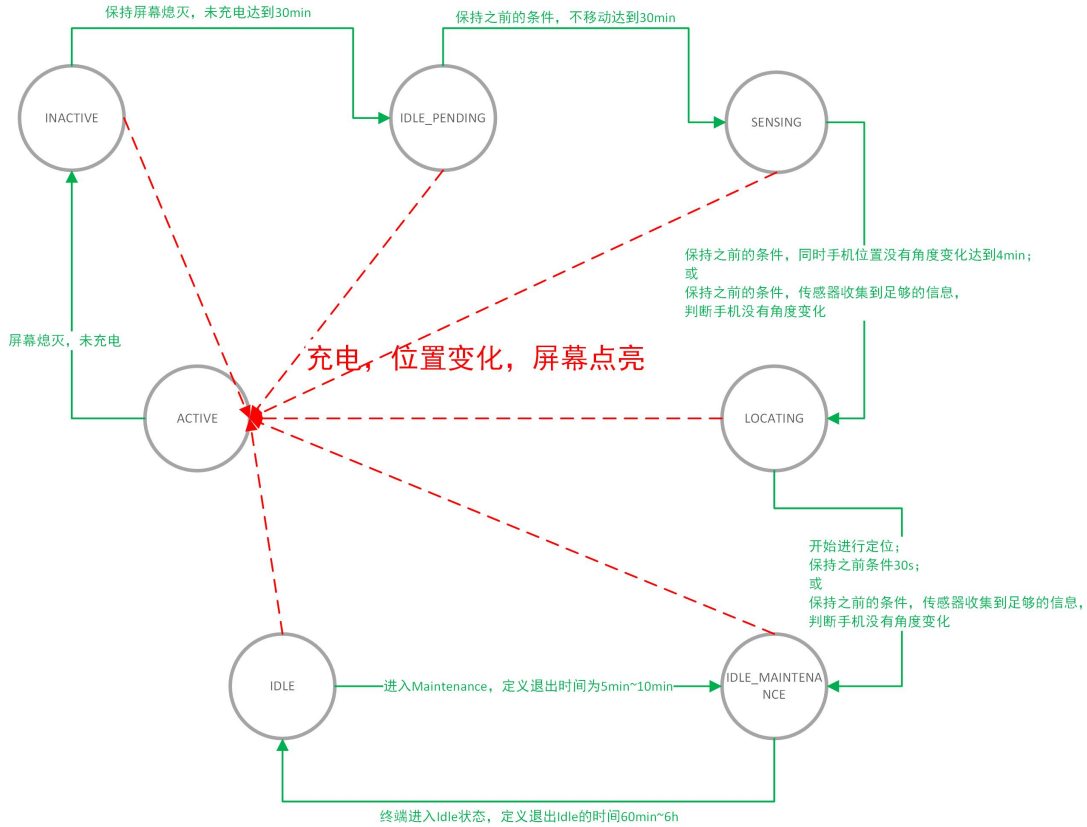
```
1 void stepIdleStateLocked(String reason) {
2     .....
3     final long now = SystemClock.elapsedRealtime();
4     //个人觉得，下面这段代码，是针对Idle状态设计的
5     //如果在Idle状态收到Alarm，那么将先唤醒终端，然后重新判断是否需要进入Idle态
6     //在介绍Doze模式原理时提到过，若应用调用AlarmManager的一些指定接口，仍然可以在Idle状态进
7     if ((now+mConstants.MIN_TIME_TO_ALARM) > mAlarmManager.getNextWakeFromIdleTime()) {
8         // Whoops, there is an upcoming alarm. We don't actually want to go idle.
9         if (mState != STATE_ACTIVE) {
10             becomeActiveLocked("alarm", Process.myUid());
11             becomeInactiveIfAppropriateLocked();
12         }
13         return;
14     }
15
16     //以下是Doze模式的状态转变相关的代码
17     switch (mState) {
18         case STATE_INACTIVE:
19             // We have now been inactive long enough, it is time to start looking
20             // for motion and sleep some more while doing so.
21             //保持屏幕熄灭，同时未充电达到30min，进入此分支
22
23             //注册一个mMotionListener，检测是否移动
24             //如果检测到移动，将重新进入到ACTIVE状态
25             //相应代码比较直观，此处不再深入分析
26             startMonitoringMotionLocked();
27
28             //再次调用scheduleAlarmLocked函数，此次的时间仍为30min
29             //也就说如果不发生退出Doze模式的事件，30min后将再次进入到stepIdleStateLocked函数
30             //不过届时的mState已经变为STATE_IDLE_PENDING
31             scheduleAlarmLocked(mConstants.IDLE_AFTER_INACTIVE_TIMEOUT, false);
32
33             // Reset the upcoming idle delays.
34             //mNextIdlePendingDelay为5min
35             mNextIdlePendingDelay = mConstants.IDLE_PENDING_TIMEOUT;
36             //mNextIdleDelay为60min
37             mNextIdleDelay = mConstants.IDLE_TIMEOUT;
38
39             //状态变为STATE_IDLE_PENDING
40             mState = STATE_IDLE_PENDING;
41             .....
42             break;
43         case STATE_IDLE_PENDING:
44             //保持息屏、未充电、静止状态，经过30min后，进入此分支
45             mState = STATE_SENSING;
46
47             //保持Doze模式条件，4min后再次进入stepIdleStateLocked
48             scheduleSensingTimeoutAlarmLocked(mConstants.SENSING_TIMEOUT);
49
50             //停止定位相关的工作
51             cancelLocatingLocked();
52             mNotMoving = false;
53             mLocated = false;
54             mLastGenericLocation = null;
55             mLastGpsLocation = null;
56
57             //开始检测手机是否发生运动（这里应该是更细致的侧重于角度的变化）
58             //若手机运动过，则重新变为active状态
59             mAnyMotionDetector.checkForAnyMotion();
60             break;
61         case STATE_SENSING:
62             //上面的条件满足后，进入此分支，开始获取定位信息
63             cancelSensingTimeoutAlarmLocked();
```

```
64     mState = STATE_LOCATING;
65     .....
66     //保持条件30s，再次调用stepIdleStateLocked
67     scheduleAlarmLocked(mConstants.LOCATING_TIMEOUT, false);
68
69     //网络定位
70     if (mLocationManager != null
71         && mLocationManager.getProvider(LocationManager.NETWORK_PROVIDER) != null) {
72         mLocationManager.requestLocationUpdates(mLocationRequest,
73             mGenericLocationListener, mHandler.getLooper());
74         mLocating = true;
75     } else {
76         mHasNetworkLocation = false;
77     }
78
79     //GPS定位
80     if (mLocationManager != null
81         && mLocationManager.getProvider(LocationManager.GPS_PROVIDER) != null) {
82         mHasGps = true;
83         mLocationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER,
84             mGpsLocationListener, mHandler.getLooper());
85         mLocating = true;
86     } else {
87         mHasGps = false;
88     }
89
90     // If we have a location provider, we're all set, the listeners will move state
91     // forward.
92     if (mLocating) {
93         //无法定位则直接进入下一个case
94         break;
95     }
96     case STATE_LOCATING:
97         //停止定位和运动检测，直接进入到STATE_IDLE_MAINTENANCE
98         cancelAlarmLocked();
99         cancelLocatingLocked();
100         mAnyMotionDetector.stop();
101
102     case STATE_IDLE_MAINTENANCE:
103         //进入到这个case后，终端开始进入Idle状态，也就是真正的Doze模式
104
105         //定义退出Idle的时间此时为60min
106         scheduleAlarmLocked(mNextIdleDelay, true);
107         .....
108         //退出周期逐步递增，每次乘2
109         mNextIdleDelay = (long)(mNextIdleDelay * mConstants.IDLE_FACTOR);
110         .....
111         //周期有最大值6h
112         mNextIdleDelay = Math.min(mNextIdleDelay, mConstants.MAX_IDLE_TIMEOUT);
113         if (mNextIdleDelay < mConstants.IDLE_TIMEOUT) {
114             mNextIdleDelay = mConstants.IDLE_TIMEOUT;
115         }
116
117         mState = STATE_IDLE;
118         .....
119         //通知PMS、NetworkPolicyManagerService等Doze模式开启，即进入Idle状态
120         //此时PMS disable一些非白名单WakeLock；NetworkPolicyManagerService开始限制一些应用的
121         //消息处理的具体流程比较直观，此处不再深入分析
122         mHandler.sendMessage(MSG_REPORT_IDLE_ON);
123         break;
124
125     case STATE_IDLE:
126         //进入到这个case时，本次的Idle状态暂时结束，开启maintenance window
127
128         // We have been idling long enough, now it is time to do some work.
129         mActiveIdleOpCount = 1;
130         mActiveIdleWakeLock.acquire();
131
132         //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）
133         scheduleAlarmLocked(mNextIdlePendingDelay, false);
134
```

```
135         mMaintenanceStartTime = SystemClock.elapsedRealtime();
136         //调整mNextIdlePendingDelay , 乘2 ( 最大为10min )
137         mNextIdlePendingDelay = Math.min(mConstants.MAX_IDLE_PENDING_TIMEOUT,
138             (long)(mNextIdlePendingDelay * mConstants.IDLE_PENDING_FACTOR));
139
140         if (mNextIdlePendingDelay < mConstants.IDLE_PENDING_TIMEOUT) {
141             mNextIdlePendingDelay = mConstants.IDLE_PENDING_TIMEOUT;
142         }
143
144         mState = STATE_IDLE_MAINTENANCE;
145         .....
146         //通知PMS等暂时退出了Idle状态，可以进行一些工作
147         //此时PMS enable一些非白名单WakeLock；NetworkPolicyManagerService开始允许应用的网络
148         mHandler.sendMessage(MSG_REPORT_IDLE_OFF);
149         break;
150     }
151 }
```

至此，stepIdleStateLocked的流程介绍完毕。

我们知道了，在DeviceIdleController中，为终端定义了7中状态，如下图所示：



手机被操作的时候为Active状态。

当手机关闭屏幕或者拔掉电源的时候，手机开始判断是否进入Doze模式。

经过一系列的状态后，最终会进入到IDLE状态，此时才算进入到真正的Doze模式，系统进入到了深度休眠状态。

此时，系统中非白名单的应用将被禁止访问网络，它们申请的Wakelock也会被disable。

从上面的代码可以看出，系统会周期性的退出Idle状态，进入到MAINTENANCE状态，集中处理相关的任务。

一段时间后，会重新再次回到IDLE状态。每次进入IDLE状态，停留的时间都会是上次的2倍，最大时间限制为6h。

当手机运动，或者点亮屏幕，插上电源等，系统都会重新返回到ACTIVIE状态。

四、总结

本篇博客中，我们分析了Doze模式对应的服务DeviceIdleController。

在了解DeviceIdleController的初始化过程后，我们重点分析了其定义的状态转移过程。

当然这些分析集中在了框架的源码分析上，至于Doze模式对App的影响，建议阅读下面的文章：

[Android M新特性Doze and App Standby模式详解](#)

顶

0

踩

0

上一篇

个人记录-LeetCode 22. Generate Parentheses

下一篇

个人记录-LeetCode 23. Merge k Sorted Lists

相关文章推荐

- Android7.0 Doze模式分析（二）wakelock
 - 【直播】机器学习之凸优化--马博士
 - Android7.0 Doze模式分析 Doze介绍 & DeviceIdle...
 - 【直播】计算机视觉原理及实战--屈教授
 - Android 7.0 Doze模式分析
 - 机器学习&数据挖掘7周实训--韦玮
 - Android 6.0 设备Idle状态介绍
 - 机器学习之数学基础系列--AI100
- Android7.0 Doze模式分析（一）Doze介绍 & Devi...
 - 【套餐】2017软考系统集成项目管理工程
 - Android7.0 BatteryService
 - 【课程】深入探究Linux/VxWorks的设备机
 - android 7.0对开发者会有哪些影响
 - Android 7.0系统启动流程分析
 - Android7.0(Android N)适配教程，心得
 - Android7.0 之 直接启动

查看评论

暂无评论

发表评论

用 户 名 :

haijunz

评论内容 :



提交

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

公司简介 | 招贤纳士 | 广告服务 | 联系方式 | 版权声明 | 法律顾问 | 问题报告 | 合作伙伴 | 论坛反馈

网站客服 杂志客服 微博客服 webmaster@csdn.net 400-660-0108 | 北京创新乐知信息技术有限公司 版权所有 | 江苏知之为计算机有限公司 |

江苏乐知网络技术有限公司

京 ICP 证 09002463 号 | Copyright © 1999-2017, CSDN.NET, All Rights Reserved

