Android (/tags/#Android) Performance (/tags/#Performance) Android Performance Patterns (/tags/#Android Performance Patterns)

Android Performance Patterns——Battery Performance

Android Performance Patterns系列学习思考和实践笔记

Posted by Cheson on April 19, 2017

0. Preface

Google的Android Performance Patterns系列中将电池也归结到了性能的篇章中，通常我在整机项目中知识分类时会将电池这个点划分到功耗底下去，既然这里是按着Google的思路来学习的，那这一章就专门称为电池性能吧，其中涉及到的很多知识点在我的Power主题中已经包含了，届时会给到相关链接和说明。

Android Performance Patterns系列中对电池性能讲解的比较跳跃，第一季最后涉及了电池性能问题的介绍，零散的讲到了几处优化方法，在第二季中也有散失的几处优化思路。在这一篇中我根据自己的思路和对耗电的理解来重新组织了下知识的结构。第一部分介绍影响功耗的一些点；第二部分针对第一部分提到的点给出一些优化思路和手段；第三部分将介绍功耗问题分析的方法和工具，尤其会包含我在手机项目中积累的整机功耗的分析思路和定位方法。

1. Batter-Eat Monsters

Purdue University研究了最受欢迎的一些应用的电量消耗，平均只有30%左右的电量是被程序最核心的方法例如绘制图片，摆放布局等等所使用掉的，剩下的70%左右的电量是被上报数据，检查位置信息，定时检索后台广告信息所使用掉的。这一章来看下这些偷偷吃掉手机电池的怪物都有哪些。

1.1 Wakelock

安卓的电源管理系统设计源于Linux的电源子系统，但是作为移动终端系统，和桌面系统的一个非常大的区别在于休眠的机制不好把握，既要不影响用户使用又要不做无谓耗电的事情，android就提出了“Opportunistic suspend”伺机休眠，为了实现动态休眠管理，在安卓前期引入了wakelock的机制从用户空间就能控制系统的休眠与否，解决了移动设备上动态休眠的问题，但是由于其破坏了Linux Kernel的休眠机制，所以Linux kernel一直没有将此补丁纳入到内核分支中去。故事发展到后来，android放弃了直接用wakelock来作为控制休眠的锁，转投到wake source和autoSuspend中，和Linux kernel又走到了一起，去看autosuspend库中的源码以及kernel中power系统的源码时就会发现很多文件和代码已经成为了历史。以上就是wakelock的一小段简史，本节的正经内容是介绍wakelock如何蚕食我们宝贵的电量的。

上面提到了原始的wakelock机制已经成为历史，但是提供给app设计时使用的接口还是叫wakelock，只是它的概念转变了，app中使用的接口为`PowerManager.WakeLock`。wakelock影响系统进入休眠的影响点有三个方面：1、wakelock的类型；2、是否超时；3、申请和回收。

wakelock在PowerManager.java定义了很多flag可供使用，在会影响系统休眠的flag中可以分成两大类，cpu和screen（其实在kernel部分对应的wakelock类型也就是cpu_block和screen_block）。其中阻止cpu进入休眠也就意味着系统无法进入suspend了，而申请了亮屏的wakelock之后screen将灭屏，而休眠流程的触发条件就是在灭屏时发起的，所以也是意味这系统无法suspend；2、在申请wakelock时可以选用超时的锁`mWakelock.acquire(5*1000)`，如果能预估任务完成的时间，那么在time up的时候会自动释放掉此wakelock；3、你对wakelock可以一无所知，但是当你用它时一定要记住要申请了就要释放（除了超时锁），这是wakelock使用的第一铁则。

综合以上三点，wakelock产生的功耗问题场景就非常明确了：定义了影响系统suspend的flag，只做了申请动作而将释放遗忘了，此wakelock又非超时锁，不会被系统自动回收。

关于wakelock的更多知识可以参考我博客中的关于电源系统的另外一些文章：

Android电源管理之申请电源锁 (https://chendongqi.github.io/blog/2017/02/23/pm_wl_acquire_flow/)

Android电源管理之释放电源锁 (https://chendongqi.github.io/blog/2017/02/25/pm_wl_release_flow/)

Android电源管理之系统休眠 (https://chendongqi.github.io/blog/2017/02/27/pm_suspend/)

1.2 Alarm

Alarm是安卓系统提供给我们的唤醒系统来处于定时任务的一种方式，涉及到功耗问题的主要知识点为Alarm的两个属性：重复和精准。 重复的Alarm是功耗问题的一个重要来源，频繁重复的唤醒系统当然会打搅手机的沉睡，尤其是当重复的间隔时间被设置的过于频繁时。而另一个和功耗息息相关的特征为Alarm的精准性，它和功耗的关系中间有隔了一个“延迟”的概念。在android4.1之后，对Alarm的机制进行了一次功耗的优化，产

生了非精准Alarm，也就是说这类Alarm的触发时间可以被延迟，结合“分批”的方法，可以减少系统被唤醒的次数，此部分的原理和源码解读可以参考博客中的另一篇文章AlarmManagerService之设置alarm流程 (https://chendongqi.github.io/blog/2017/02/14/SetAlarmFlow/)。

所以，当Alarm使用不合理的时候，当出现过于频繁的重复任务，或者是使用了不恰当的精准Alarm时就有可能导致功耗问题的发生。

1.3 Network

网络部分的耗电会是系统耗电占用的大头，我在整机项目中得到的经验，在手机处于idle待机时，待机电流仅有10mA以下，优秀的项目底电流可以达到6mA以下。而测试有网络数据的行为时，电流的增益就需要超过20mA了。这一节来分析网络对功耗的影响，涉及到的点有WiFi网络和数据网络的功耗差异，数据网络的工作状态和功耗的关系。

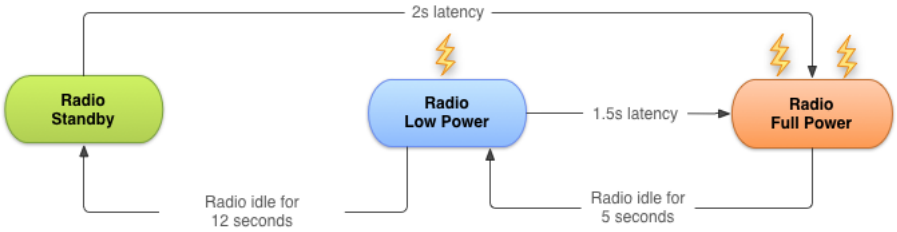
先来谈谈同为无线网络传输的WiFi和数据网络谁更耗电呢？从终结流言：看手机3G/WiFi究竟谁更耗电(http://www.jb51.net/diannaojichu/334466.html)这里的测试结果中可以看出数据流量是比WiFi更耗电的，而从我司测试处拿到的数据，也是同样的结论。那么为什么数据网络会比WiFi更耗电呢？从硬件角度扯开了来谈一下。

最基础的手机的模型，包括了三部分芯片：射频芯片，基带芯片和AP芯片（application processor）。AP芯片包括了cpu和GPU，用来完成应用部分的处理；基带芯片中包含了耳熟能详的Modem以及信道编码，信源编码和信令处理。其中Modem负责将上行信号进行调制或者下行信号的解调；射频芯片部分负责接收/发送载波，或许有些方案中还会涉及到功率放大器（PA， Power Amplifier）来增强信号的强度。当使用数据网络时，就会使用这一套机制来进行通信。而WiFi使用的是单独的WiFi芯片，通过connectivity子系统来进行通信。从硬件的角度来考虑，使用数据网络占用到的资源要更多一些，实际的测试电流也会更大。更多的数据，在MTK平台上开启modem log时，电流会多增加50mA左右。而另一个数据网络和WiFi网络的区别在于范围，数据网络要覆盖的范围更大，射频的发射功率也会较之更大，如果再加上PA，那么功耗就更夸张了。

第二点来看下数据网络工作状态的原理来拆解下功耗的产生。典型的3G网络包含了三种能量状态：

- 1. **Full Power**： 当无线连接被激活的时候，允许设备以最大的传输速率进行操作。
- 1. **Low Power**：一种中间状态，对电量的消耗差不多是 Full power 状态下的50%。
- 1. **Standby**：最小的能量状态，没有被激活或者需求的网络连接。

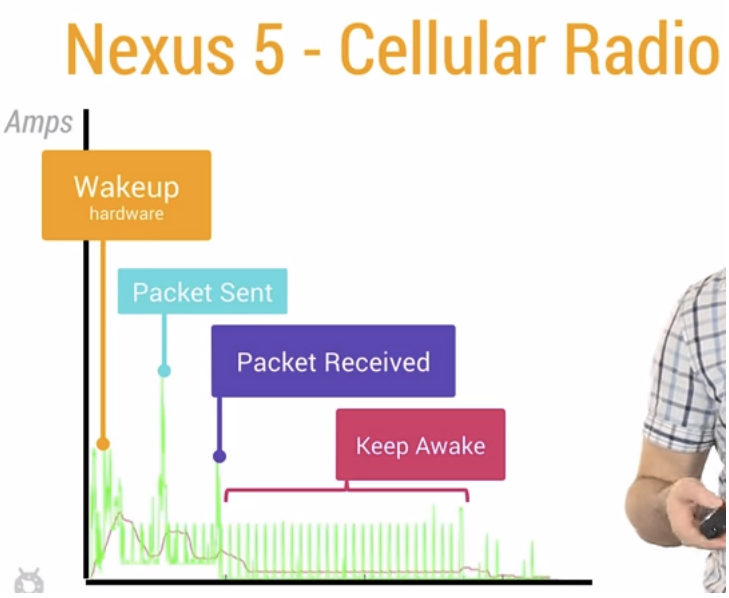
下图为AT&T电信提供的一种3G网络状态机的图示



从Standby切换到Full Power需要两秒的时间，从Low Power切换到Full Power需要1.5秒的时间，从Full Power切换到Low Power有5秒的延迟，Low Power到Standby有12秒的过渡。根据这个模型我们来探究下app的行为对功耗的影响。

假设一个app每20秒发送1秒的未捆绑数据。那么状态机在1分钟内切换的历程将是这样的：Standby->Full Power（2秒），发送数据(1秒)，Full Power->Low Power（5秒），Low Power->Standby(12秒)如此循环3次，也就是说在一分钟内，经历了18秒的Full Power和42秒的Low Power，从未有过Standby的状态。在了解了数据网络状态的切换原理之后其中存在着非常大的优化空间，仅仅是发送1秒中的数据，搞得如此鸡飞狗跳，具体的优化思想会在后面优化章节中提到。

和上面原理类似，下图是来自Google N5的数据网络工作的耗流曲线，通过此图也能看出数据网络在不同工作状态时的耗流是存在很大差异的。



####1.4 Location

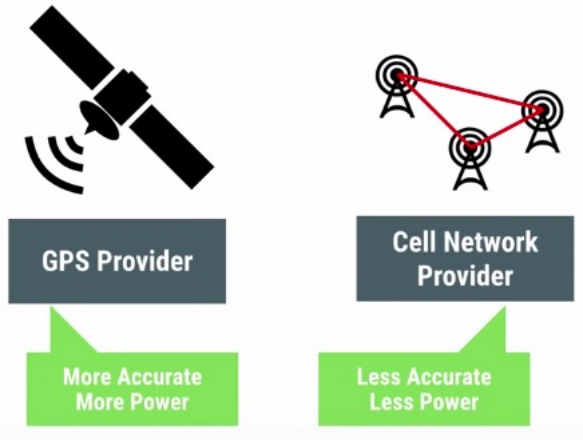
大家在使用手机的感受中肯定包含了如下这条：当使用地图等此类定位服务时，耗电量和手机发热会突增。这里就涉及到了定位服务，它是电池的另一个杀手，而发热是耗电的另一种表现。本节就来聊聊使用定位服务时，什么情况下会浪费手机的宝贵电量。

Location服务跟耗电的关系和Alarm有点类似，一是位置信息的更新频率，二是位置服务的精度（关系到获取位置信息的方式）。从位置服务的通常使用方法中来探究下

```
LocationRequest mLocationRequest = new LocationRequest();
mLocationRequest.setInterval(10000);
mLocationRequest.setFastestInterval(5000);
mLocationRequest.setSmallestDisplacement(5);
mLocationRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);
```

首先是更新频率，包含了主动查询和被动通知两种方式，主动查询包括了两个间隔时间，代码中通过setInterval设置的为该应用本身主动去获取位置信息的间隔时间，而通过setFastestInterval设置的时间的含义如下：当系统中有其他应用在获取信息服务时，此应用可以屏蔽以此值来屏蔽掉更新过于频繁的位置信息，如果收到的位置变化过于频繁，那么触发回调的代码进行逻辑处理的次数也就越多，耗电就显而易见。而另一种被动的通知是当距离超过默认值时就回调方法来处理，所以距离设置不合理的时候也可能导致不必要的浪费。例如汽车导航的应用在高速行驶的场景下，间隔几米就触发回调，那么耗电将是非常可怕的。

另一个和耗电有很大关联的特征是精确度，要知道定位的方式不只是GPS，还包括了基站和WiFi网络。



GPS定位的准确性比较高，相应的耗电也较多，而基站定位使用了三角定位算法，精度一般但是耗电少。另外也有通过WiFi定位，在室内可以作为GPS定位不了的一种补充，其耗电也是较低的。一般来说NETWORK（基站和WiFi）得到的位置精度一般在500-1000米，GPS得到的精度一般在5-50米。而同时通过setPriority设置的参数又可以调整位置信息的精度。所以根据应用的需求合理的设置位置信息的来源和精度是非常重要的一个抉择，设置过高的精度就可能导致电池资源的浪费。

2. Optimization suggestion

耗电的问题都是在做一种抉择，省电的同时必然将导致某些性能的损失，关键在于把握尺度，游刃有余的去合理控制性能和电池的平衡。所以针对以上提及到的一些功耗问题，能给出的一些优化方向很多不能称之为绝对的方案，姑且称之建议吧。需要根据特定的应用场景，需求来采用可选的建议或者是调整适合的参数来改善。

2.1 WakeLock

针对前面分析过的WakeLock的功耗，如果是非超时锁没有release导致的，那么是有确定的方案来修复的，必须保证是有release的逻辑。如果是超时锁，那么就要考虑在wakelock唤醒系统之后是否能"及时"返回到休眠状态。那么其中的利弊就在于超时时间的设定是否合理，如果是一个固定任务时间可衡量的情况下还比较乐观，但是加入涉及到网络传输，等待数据返回这种情况就很可能导致本来1分钟的任务量却等待了10分钟。这种情况下建议使用定时任务来改造这种不确定等待时间的任务。

2.2 Alarm

Alarm的使用建议，在没有特殊要求的情况下，尽量采用非精准Alarm来处理定时任务。可以让系统有空间来自行调整Alarm的触发策略达到省电的目的。提及点题外的或者说是更深层次的Alarm优化，作为ODM行业，我们可以具有更大的自由度来修改AlarmManagerService的处理策略（此方式虽然是Google最为头疼的android开元带来的版本的不统一），我曾将所有非系统应用申请的Alarm作为非精准闹钟来处理，统一加大他们的最大触发延迟时间。像MiUI论坛上经常有人吐槽的Alarm不准，应该就是舍弃了过多的用户体验导致的。

2.3 Network

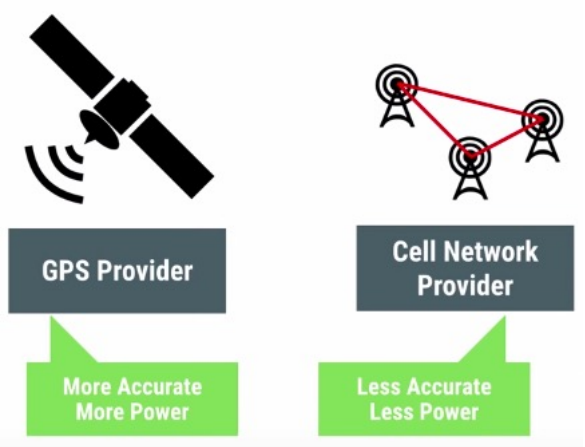
关于网络的功耗，前面分析了数据网络的状态切换的原理，每隔20秒发送1秒的数据，会导致数据网络停留在Full Power状态18秒，Low Power状态42秒。针对这点可以有一种优化策略，将数据打包传输，试着将每隔20秒发送1秒数据的策略改成一分钟发送3秒的数据，考虑这个情况下的数据网络状态。会变成1分钟内会有8秒的Full Power状态，12秒的Low Power和40秒的Standby，功耗的减少显而易见。

除了数据捆绑之外还有一种数据预取的技术可以用来优化，其原理和缓存机制类似，在一次连接中尽最大下载能力预取到用户可能会使用到的资源。预取技术同时也存在着一些可能带来的负面影响：频繁的使用导致取到大量不需要的资源反而会浪费为此付出的电量消耗；应用因为等待预取动作的结束而导致启动时间变慢；预取带来的数据流量消耗的增加。

另外节省上传数据的size势必会带来好处，所以能够想到的就是压缩技术了。可以参考以下内容：Image Compression (https://www.html5rocks.com/en/tutorials/speed/img-compression/)，Text Compression (https://www.html5rocks.com/en/tutorials/speed/txt-compression/)。

2.4 Location

Location相关的优化方向也是考虑间隔的时间和精度。间隔时间部分除了根据需求的特性来尽量调整查询的时间间隔（能调多长是多长）之外，另外有更智能的方式可以选择。试想这样的场景，导航应用每隔10秒查询一次位置信息，当用户在某地驻留1分钟（问路或是查询信息），此时有6次查询到的位置信息是没有变化的，而重复的位置信息在这里也没有任何意义。所以此时可以使用如下的策略：当查询到重复的位置信息时，将下一次的查询时间延迟，具体延迟的时间可以选择设定，加倍或是更长。类似的思路在doze模式中也有使用，在首次进入idle状态时持续1小时，后面就加倍，也许此类策略是google同一批工程师设计的吧。



2.5 JobScheduler

JobScheduler是在Android L上引入的一个机制，其基本思想也是为了把非即时的任务拖延到一起触发。它和AlarmManagerService中的延迟触发和分批机制也存在异曲同工之妙，只是较之后者，JobScheduler可以做的事更多，延迟触发的条件就不仅限于时间角度了，也可以是某个动作和状态的改变，例如连接WiFi、插入充电器等。具体的用法和API等不再这里赘述，可以参考google的介绍JobScheduler (https://developer.android.com/reference/android/app/job/JobScheduler.html)。

3. How to Analyse

这节介绍我在Android手机项目中获得的整机电流问题分析的经验，软件部分的电流问题两个大的原因来自于系统被异常唤醒和系统无法休眠，先来看下系统异常唤醒导致的电流问题。

3.1 Precondition

在分析一个问题时，首先要了解它，这个原则在电流问题分析中尤为重要，经常会遇到不太专业的测试人员报出的乌龙问题浪费大家的宝贵经历。在分析之前首先要问几个问题：1、传导测试是否pass；2、IMEI是否存在；3、天线是否正常；4、射频是否校准；5、Modem是否正常；6、Sim卡是否正常；7、AP是否纯净，VPN网络环境是否正常。当然有些条件是在不同测试case下的组合，如果所有前置条件都没有问题，那么恭喜你拿到了一个真正的电流问题了。

3.2 wakeup source

遇到频繁唤醒导致的电流问题时，我们要做的就是寻找这个唤醒源。通常的唤醒源可以来自于Modem，WiFi和EINT。

首先我会去从kernel log中寻找唤醒源的类型，如下面的log

```
<4>[  458.748910] -(0)[827:system_server][SPM] wake up byCCIF0_MD, timer_out = 789553, r13 = 0x3c061238, debug_flag = 0x1f
...
<4>[  396.639923] -(0)[924:system_server][SPM] wake up byCONN2AP, timer_out = 4847, r13 = 0x10007000, debug_flag = 0x9f
...
<4>[  397.323912] -(0)[924:system_server][SPM] wake up byEINT, timer_out = 478567, r13 = 0x10001000, debug_flag = 0x9f
```

就代表了非常典型的三类唤醒源，CCIF0_MD就是这里Modem的唤醒，这个名称和Modem的架构设计有关。CONN2AP是connectivity子系统的唤醒，通常会是WiFi的唤醒。EINT是中断唤醒，和上层相关的的就是PMIC唤醒，PMIC的中断号可以查询驱动配置来确认，MTK平台上默认通常会是206。

先来看PMIC，PMIC唤醒中又包括了两类，一个是power键，在唤醒log的后面可以看到 `power key generate` 这类关键字，这种唤醒被视作用户或者测试行为，就可以过滤掉，然而这个power键的信息可以作为其他依据，如Power Monitor波形和log时间点对照，判断power键是否有被触发等。PMIC的另一种来源就是RTC Alarm，对应到上层就是app设置的Alarm唤醒，这个就是上层应用导致的功耗问题的重要来源。MTK平台项目在唤醒log的后面可以看到有 `rtc_tasklet_handler` 此类log，就可以判定这此唤醒是RTC Alarm唤醒。然后要做的下一步就是去寻找Alarm的来源。

如何找到是谁设置的Alarm？在syslog中搜索 `AlarmManager: send alarm` 或者 `wakeup alarm Alarm=` 此类信息，注意后面的type是0或者2的。为什么是0或者2，在申请Alarm时有4中可选的类型，分别是RTC、Elapsed和Wakeup的4种组合，带Wakeup的就是会唤醒系统，它们对应的int类型的值分别就是0和2。

接下来再看CONN2AP，我接触到的通常就是WiFi的唤醒，WiFi唤醒的问题通常也要先做下排除，确认WiFi天线是否异常，连接的AP是否纯净，一台AP连接过多会产生一些网络问题导致功耗测试的不准确。因为WiFi问题本身没有过多的系统信息可以提供，所以此类问题到这一步会转给WiFi专家去分析WiFi相关log。之前是有遇到过局域网内网络广播和组播导致的WiFi频繁唤醒的问题，修改的策略是在suspend的时候防火墙去过滤掉WiFi的广播和组播，此修改的side effects中不会带来广域网数据传输的影响。

CONN2AP的唤醒还有一个来源就是有数据传输，进一步的就需要去分析是哪个应用在占用此资源。从实战的经验里获得了两种去定位的方式：1、抓netlog，通过wireshark工具解析netlog之后根据唤醒的时间点找到netlog中访问的ip地址，再从mainlog中去定位ip地址执行的站点或者域名就基本可以定位到问题；2、更简洁的方式可以在mainlog中搜索关键字“libc-netbsd”，可以找到类似如下的信息

```
11-04 17:39:36.858645  1763  3085 D libc-netbsd: [getaddrinfo]: hostname=mtalk.google.com; servname=(null); netid=0; mark=0
11-04 17:39:36.858763  1763  3085 D libc-netbsd: [getaddrinfo]: ai_addrlen=0; ai_canonname=(null); ai_flags=4; ai_family=0
11-04 17:39:36.859041  1763  3085 D libc-netbsd: [getaddrinfo]: hostname=mtalk.google.com; servname=(null); netid=0; mark=0
11-04 17:39:36.859143  1763  3085 D libc-netbsd: [getaddrinfo]: ai_addrlen=0; ai_canonname=(null); ai_flags=1024; ai_family=0
11-04 17:39:36.861148   209  3087 D libc-netbsd: [getaddrinfo]: hostname=mtalk.google.com; servname=(null); netid=0; mark=0
11-04 17:39:36.861266   209  3087 D libc-netbsd: [getaddrinfo]: ai_addrlen=0; ai_canonname=(null); ai_flags=1024; ai_family=0
11-04 17:39:36.861387   209  3087 D libc-netbsd: getaddrinfo: mtalk.google.com get result from proxy gai_error = 11
```

如果能找到唤醒匹配的时间点出现此类log也可以定位是在和host发生数据传输。

最后一大类是Modem唤醒的，要通过进一步的channel id来确定具体的唤醒源，参照如下表格：

Channel ID	Channel Name	User	进一步Debug手段
6/8	CCCI_UART1	Modem Log	关闭Modem log
42/43	CCCI_MD_LOG	Modem Log	关闭Modem log
32/33	CCCI_RPC	Modem使用AP资源（如sim卡中断）	检查DWS、硬件中断是否异常
14/15	CCCI_FS	Modem读写nvram	需要协议分析modem log
20/22/24/26/28/30	CCCI_CCMNI	有数据传输	分析NetLog
34/36	CCCI_IPC	4GModem跟WCN有频段重叠，必要时同步导致	需要协议分析modem log
10/12	CCCI_UART2	MUXD，通常是AT指令	Radio Log基本可以，必要时再抓Modem log

PREVIOUS
ANDROID电源管理之DOZE模式专题系列（九）
(/2017/04/10/PM_DOZE_STATE_SUMMARY/)

NEXT
ANDROID电源管理之DOZE模式专题系列（十一）
(/2017/05/12/PM_DOZE_CONN_CONTROL/)

FEATURED TAGS (/tags/)

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

FRIENDS

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

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