**后台任务 - 保持设备唤醒状态**

当Android设备空闲时，屏幕会变暗，然后关闭屏幕，最后会停止CPU的运行，这样可以防止电池电量掉的快。在休眠过程中自定义的Timer、Handler、Thread、Service等都会暂停。但有些时候我们需要改变Android系统默认的这种状态：比如玩游戏时我们需要保持屏幕常亮，比如一些下载操作不需要屏幕常亮但需要CPU一直运行直到任务完成。

## ****保持屏幕常亮****

最好的方式是在Activity中使用**FLAG\_KEEP\_SCREEN\_ON** 的Flag。

**public class** MainActivity **extends** Activity {  
 @Override  
 **protected void** onCreate(Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
 setContentView(R.layout.***activity\_main***);  
 getWindow().addFlags(WindowManager.LayoutParams.***FLAG\_KEEP\_SCREEN\_ON***);  
 }  
}

这个方法的好处是不像唤醒锁（wake locks），需要一些特定的权限（permission）。并且能正确管理不同app之间的切换，不用担心无用资源的释放问题。   
另一个方式是在布局文件中使用**android:keepScreenOn**属性：

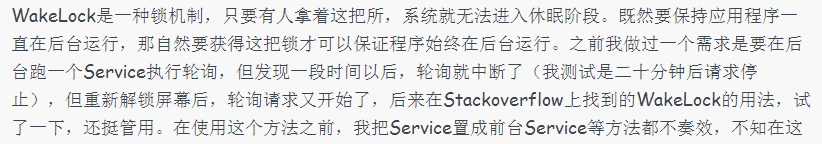
<**RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 android:keepScreenOn="true"**>  
 ...  
</**RelativeLayout**>

android:keepScreenOn = ” true “的作用和FLAG\_KEEP\_SCREEN\_ON一样。使用代码的好处是你允许你在需要的地方关闭屏幕。

注意：一般不需要人为的去掉FLAG\_KEEP\_SCREEN\_ON的flag，windowManager会管理好程序进入后台回到前台的的操作。如果确实需要手动清掉常亮的flag，使用**getWindow().clearFlags(WindowManager.LayoutParams.FLAG\_KEEP\_SCREEN\_ON)**

## ****保持CPU运行****

需要使用**PowerManager**这个系统服务的唤醒锁(wake locks)特征来保持CPU处于唤醒状态。唤醒锁允许程序控制宿主设备的电量状态。创建和持有唤醒锁对电池的续航有较大的影响，所以，除非是真的需要唤醒锁完成尽可能短的时间在后台完成的任务时才使用它。比如在Acitivity中就没必要用了。如果需要关闭屏幕，使用上述的FLAG\_KEEP\_SCREEN\_ON。   
只有一种合理的使用场景，是在使用后台服务**在屏幕关闭情况下hold住CPU完成一些工作**。 要使用唤醒锁，如果不使用唤醒锁来执行后台服务，不能保证因CPU休眠未来的某个时刻任务会停止，这不是我们想要的。 (有的人可能认为我以前写的后台服务就没掉过链子呀运行得挺好的，1.可能是你的任务时间比较短；2.可能CPU被手机里面很多其他的软件一直在唤醒状态。)。下面是很多网友有同样的问题：



唤醒锁可划分为并识别四种用户唤醒锁：

| **标记值** | **CPU** | **屏幕** | **键盘** |
| --- | --- | --- | --- |
| [PARTIAL\_WAKE\_LOCK](http://developer.android.com/reference/android/os/PowerManager.html) | 开启 | 关闭 | 关闭 |
| [SCREEN\_DIM\_WAKE\_LOCK](http://developer.android.com/reference/android/os/PowerManager.html) | 开启 | 变暗 | 关闭 |
| [SCREEN\_BRIGHT\_WAKE\_LOCK](http://developer.android.com/reference/android/os/PowerManager.html) | 开启 | 变亮 | 关闭 |
| [FULL\_WAKE\_LOCK](http://developer.android.com/reference/android/os/PowerManager.html) | 开启 | 变亮 | 变亮 |

请注意，自 API 等级 17 开始，FULL\_WAKE\_LOCK 将被弃用。 应用应使用 FLAG\_KEEP\_SCREEN\_ON。

第一步就是添加唤醒锁权限：

<**uses-permission android:name="android.permission.WAKE\_LOCK"** />

直接使用唤醒锁：

PowerManager **powerManager** = (PowerManager) getSystemService(***POWER\_SERVICE***);  
WakeLock **wakeLock** = **powerManager**.newWakeLock(PowerManager.***PARTIAL\_WAKE\_LOCK***,**"MyWakelockTag"**);  
wakeLock.acquire();

**注意：在使用该类的时候，必须保证acquire和release是成对出现的。**

但推荐的方式是使用**WakefulBroadcastReceiver**：使用广播和Service（典型的**IntentService**）结合的方式可以让你很好地管理后台服务的生命周期。

WakefulBroadcastReceiver是BroadcastReceiver的一种特例。它会为你的APP创建和管理一个PARTIAL\_WAKE\_LOCK 类型的WakeLock。WakefulBroadcastReceiver把工作交接给service（通常是IntentService），并保证交接过程中设备不会进入休眠状态。如果不持有WakeLock，设备很容易在任务未执行完前休眠。最终结果是你的应用不知道会在什么时候能把工作完成，相信这不是你想要的。

使用**WakefulBroadcastReceiver**第一步就是在Manifest中注册：

<**receiver android:name=".MyWakefulReceiver"**></**receiver**>

使用**startWakefulService()**方法来启动服务，与**startService()**相比，在启动服务的同时，并启用了唤醒锁。

**public class** MyWakefulReceiver **extends** WakefulBroadcastReceiver {  
 @Override  
 **public void** onReceive(Context context, Intent intent) {   
 *// Start the service, keeping the device awake while the service is   
 // launching. This is the Intent to deliver to the service.* Intent service = **new** Intent(context, MyIntentService.**class**);   
 *startWakefulService*(context, service);   
 }  
}

当后台服务的任务完成，要调用**MyWakefulReceiver.completeWakefulIntent()**来释放唤醒锁。

**public class** MyIntentService **extends** IntentService {  
 **public static final int *NOTIFICATION\_ID*** = 1;  
 **private** NotificationManager **mNotificationManager**;  
 NotificationCompat.Builder **builder**;  
  
 **public** MyIntentService() {  
 **super**(**"MyIntentService"**);  
 }  
  
 @Override  
 **protected void** onHandleIntent(Intent intent) {  
 Bundle extras = intent.getExtras();   
 *// Do the work that requires your app to keep the CPU running.   
 // ...   
 // Release the wake lock provided by the WakefulBroadcastReceiver.* MyWakefulReceiver.completeWakefulIntent(intent);   
 }  
}

网上采集的一些问题坑点及解决如下：

1.向服务器轮询的代码不执行。

曾经做一个应用，利用Timer和TimerTask，来设置对服务器进行定时的轮询，但是发现机器在某段时间后，轮询就不再进行了。查了很久才发 现是休眠造成的。后来解决的办法是，利用系统的AlarmService来执行轮询。因为虽然系统让机器休眠，节省电量，但并不是完全的关机，系统有一部 分优先级很高的程序还是在执行的，比如闹钟，利用AlarmService可以定时启动自己的程序，让cpu启动，执行完毕再休眠。

2.后台长连接断开。

最近遇到的问题。利用Socket长连接实现QQ类似的聊天功能，发现在屏幕熄灭一段时间后，Socket就被断开。屏幕开启的时候需进行重连，但 每次看Log的时候又发现网络是链接的，后来才发现是cpu休眠导致链接被断开，当你插上数据线看log的时候，网络cpu恢复，一看网络确实是链接的， 坑。最后使用了PARTIAL\_WAKE\_LOCK，保持CPU不休眠。

3.调试时是不会休眠的。  
让我非常郁闷的是，在调试2的时候，就发现，有时Socket会断开，有时不会断开，后来才搞明白，因为我有时是插着数据线进行调试，有时拔掉数据线，这 时Android的休眠状态是不一样的。而且不同的机器也有不同的表现，比如有的机器，插着数据线就会充电，有的不会，有的机器的设置的充电时屏幕不变暗 等等，把自己都搞晕了。其实搞明白这个休眠机制，一切都好说了。

**采用定时重复的Service开启：**

　　1、利用Android自带的定时器AlarmManager实现

Intent intent = **new** Intent(mContext, ServiceTest.**class**);  
PendingIntent pi = PendingIntent.*getService*(mContext, 1, intent, 0);  
AlarmManager alarm = (AlarmManager) getSystemService(Service.***ALARM\_SERVICE***);  
**if**(alarm != **null**)  
{  
 alarm.cancel(pi);  
 *// 闹钟在系统睡眠状态下会唤醒系统并执行提示功能* alarm.setRepeating(AlarmManager.***RTC\_WAKEUP***, System.*currentTimeMillis*() + 1000, 2000, pi)*;// 确切的时间闹钟//alarm.setExact(…);*  
 *//alarm.set(AlarmManager.RTC\_WAKEUP, System.currentTimeMillis(), pi);*}

　2、该定时器可以启动Service服务、发送广播、跳转Activity，并且会在系统睡眠状态下唤醒系统。所以该方法不用获取电源锁和释放电源锁。

**注意：**在19以上版本，setRepeating中设置的频繁只是建议值(6.0 的源码中最小值是60s)，如果要精确一些的用setWindow或者setExact。

首先Android手机有两个处理器，一个叫Application Processor（AP），一个叫Baseband Processor（BP）。AP是ARM架构的处理器，用于运行Linux+Android系统；BP用于运行实时操作系统（RTOS），通讯协议栈运行于BP的RTOS之上。非通话时间，BP的能耗基本上在5mA左右，而AP只要处于非休眠状态，能耗至少在50mA以上，执行图形运算时会更高。另外LCD工作时功耗在100mA左右，WIFI也在100mA左右。一般手机待机时，AP、LCD、WIFI均进入休眠状态，这时Android中应用程序的代码也会停止执行。

Android为了确保应用程序中关键代码的正确执行，提供了Wake Lock的API，使得应用程序有权限通过代码阻止AP进入休眠状态。但如果不领会Android设计者的意图而滥用Wake Lock API，为了自身程序在后台的正常工作而长时间阻止AP进入休眠状态，就会成为待机电池杀手。比如前段时间的某应用，比如现在仍然干着这事的某应用。

那么Wake Lock API有啥用呢？比如心跳包从请求到应答，比如断线重连重新登陆这些**关键逻辑的执行过程，就需要Wake Lock来保护**。而一旦一个关键逻辑执行成功，就应该立即释放掉Wake Lock了。两次心跳请求间隔5到10分钟，基本不会怎么耗电。除非网络不稳定，频繁断线重连，那种情况办法不多。

AlarmManager 是Android 系统封装的用于管理 RTC 的模块，RTC (Real Time Clock) 是一个独立的硬件时钟，可以在 CPU 休眠时正常运行，在预设的时间到达时，通过中断唤醒 CPU。（极光推送就是利用这个来做的。）

## 总结：

1. **关键逻辑的执行过程，就需要Wake Lock来保护**。如断线重连重新登陆
2. **休眠的情况下如何唤醒来执行任务？**用AlarmManager。如推送消息的获取

注意：如果请求网络很差，会要很长的时间，一般我们谷歌建议一定要设置请求超时时间。

其他参考资料：

**alarmManager在手机休眠时无法唤醒Service的问题?(** 为了对付你们这些个“”流氓“”的频繁唤醒的app，各个厂家都开发了心跳对齐。**)**

<https://www.zhihu.com/question/36421849>

**微信 Android 版 6.2 为什么设置了大量长时间的随机唤醒锁？**

<https://www.zhihu.com/question/31136645>

