**关于 Android 进程保活，你所需要知道的一切**

[D\_clock爱吃葱花](https://www.jianshu.com/u/ec95b5891948)

2016.04.17 17:04\* 字数 3556 阅读 90564评论 154喜欢 766赞赏 7

**声明：本文作者授权微信公众号Android程序员（AndroidTrending）在微信公众号平台原创首发。**

早前，我在知乎上回答了这样一个问题：[怎么让 Android 程序一直后台运行，像 QQ 一样不被杀死？](https://link.jianshu.com?t=https:/www.zhihu.com/question/29826231/answer/79475911)。关于 Android 平台的进程保活这一块，想必是所有 Android 开发者瞩目的内容之一。你到网上搜 Android 进程保活，可以搜出各种各样神乎其技的做法，绝大多数都是极其不靠谱。前段时间，Github还出现了一个很火的**“黑科技”**进程保活库，声称可以做到**进程永生不死**。

怀着学习和膜拜的心情进去Github围观，结果发现很多人提了 Issue 说各种各样的机子无法成功保活。

看到这里，我瞬间就放心了。**坦白的讲，我是真心不希望有这种黑科技存在的，它只会滋生更多的流氓应用，拖垮我大 Android 平台的流畅性。**

扯了这么多，接下来就直接进入本文的正题，谈谈关于进程保活的知识。提前声明以下四点

* **本文是本人开发 Android 至今综合各方资料所得**
* **不以节能来维持进程保活的手段，都是耍流氓**
* **本文不是教你做永生不死的进程，如果指望实现进程永生不死，请忽略本文**
* **本文有错误的地方，欢迎留下评论互相探讨（拍砖请轻拍）**

**保活手段**

当前业界的Android进程保活手段主要分为\*\* 黑、白、灰 \*\*三种，其大致的实现思路如下：

**黑色保活**：不同的app进程，用广播相互唤醒（包括利用系统提供的广播进行唤醒）

**白色保活**：启动前台Service

**灰色保活**：利用**系统的漏洞**启动前台Service

**黑色保活**

所谓黑色保活，就是利用不同的app进程使用广播来进行相互唤醒。举个3个比较常见的场景：

**场景1**：开机，网络切换、拍照、拍视频时候，利用系统产生的广播唤醒app

**场景2**：接入第三方SDK也会唤醒相应的app进程，如微信sdk会唤醒微信，支付宝sdk会唤醒支付宝。由此发散开去，就会直接触发了下面的 **场景3**

**场景3**：假如你手机里装了支付宝、淘宝、天猫、UC等阿里系的app，那么你打开任意一个阿里系的app后，有可能就顺便把其他阿里系的app给唤醒了。（只是拿阿里打个比方，其实BAT系都差不多）

**没错，我们的Android手机就是一步一步的被上面这些场景给拖卡机的。**

针对**场景1**，估计Google已经开始意识到这些问题，所以在最新的Android N取消了 ACTION\_NEW\_PICTURE（拍照），ACTION\_NEW\_VIDEO（拍视频），CONNECTIVITY\_ACTION（网络切换）等三种广播，无疑给了很多app沉重的打击。我猜他们的心情是下面这样的

而开机广播的话，记得有一些定制ROM的厂商早已经将其去掉。

针对**场景2**和**场景3**，因为调用SDK唤醒app进程属于正常行为，此处不讨论。但是在借助LBE分析app之间的唤醒路径的时候，发现了两个问题：

1. 很多推送SDK也存在唤醒app的功能
2. app之间的唤醒路径真是多，且错综复杂

我把自己使用的手机测试结果给大家围观一下（**我的手机是小米4C，刷了原生的Android5.1系统，且已经获得Root权限才能查看这些唤醒路径**）

15组相互唤醒路径

全部唤醒路径

我们直接点开 **简书** 的唤醒路径进行查看

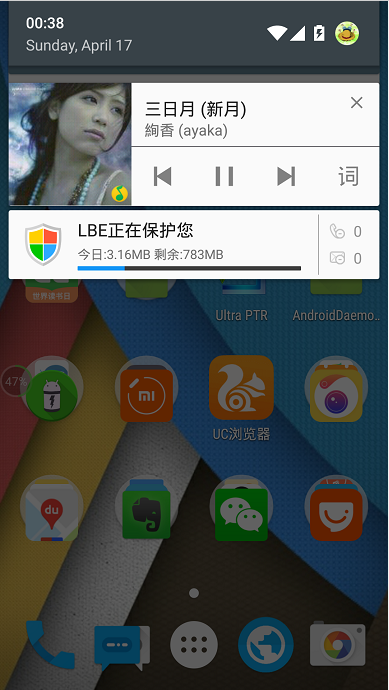
简书唤醒路径

可以看到以上3条唤醒路径，但是涵盖的唤醒应用总数却达到了23+43+28款，数目真心惊人。请注意，这只是我手机上一款app的唤醒路径而已，到了这里是不是有点细思极恐。

当然，这里依然存在一个疑问，就是LBE分析这些唤醒路径和互相唤醒的应用是基于什么思路，我们不得而知。所以我们也无法确定其分析结果是否准确，如果有LBE的童鞋看到此文章，不知可否告知一下思路呢？但是，手机打开一个app就唤醒一大批，我自己可是亲身体验到这种酸爽的......

**白色保活**

白色保活手段非常简单，就是调用系统api启动一个前台的Service进程，这样会在系统的通知栏生成一个Notification，用来让用户知道有这样一个app在运行着，哪怕当前的app退到了后台。如下方的LBE和QQ音乐这样：



**灰色保活**

灰色保活，这种保活手段是应用范围最广泛。它是利用系统的漏洞来启动一个前台的Service进程，与普通的启动方式区别在于，它不会在系统通知栏处出现一个Notification，看起来就如同运行着一个后台Service进程一样。这样做带来的好处就是，用户无法察觉到你运行着一个前台进程（因为看不到Notification）,但你的进程优先级又是高于普通后台进程的。那么如何利用系统的漏洞呢，大致的实现思路和代码如下：

* 思路一：API < 18，启动前台Service时直接传入new Notification()；
* 思路二：API >= 18，同时启动两个id相同的前台Service，然后再将后启动的Service做stop处理；

public class GrayService extends Service {

private final static int GRAY\_SERVICE\_ID = 1001;

@Override

public int onStartCommand(Intent intent, int flags, int startId) {

if (Build.VERSION.SDK\_INT < 18) {

startForeground(GRAY\_SERVICE\_ID, new Notification());//API < 18 ，此方法能有效隐藏Notification上的图标

} else {

Intent innerIntent = new Intent(this, GrayInnerService.class);

startService(innerIntent);

startForeground(GRAY\_SERVICE\_ID, new Notification());

}

return super.onStartCommand(intent, flags, startId);

}

...

...

/\*\*

\* 给 API >= 18 的平台上用的灰色保活手段

\*/

public static class GrayInnerService extends Service {

@Override

public int onStartCommand(Intent intent, int flags, int startId) {

startForeground(GRAY\_SERVICE\_ID, new Notification());

stopForeground(true);

stopSelf();

return super.onStartCommand(intent, flags, startId);

}

}

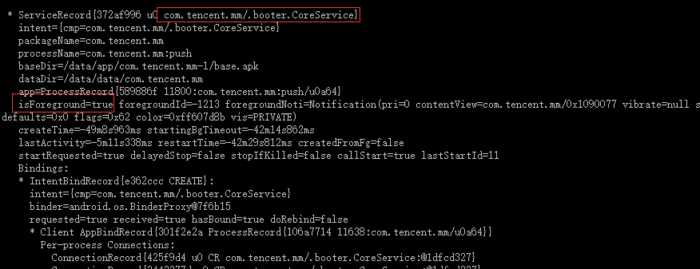
}

代码大致就是这样，能让你神不知鬼不觉的启动着一个前台Service。其实市面上很多app都用着这种灰色保活的手段，什么？你不信？好吧，我们来验证一下。流程很简单，打开一个app，看下系统通知栏有没有一个 Notification，如果没有，我们就进入手机的adb shell模式，然后输入下面的shell命令

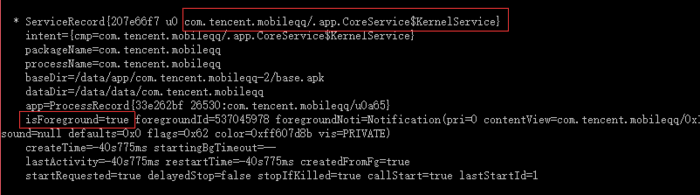
dumpsys activity services PackageName

打印出指定包名的所有进程中的Service信息，看下有没有 **isForeground=true** 的关键信息。如果通知栏没有看到属于app的 Notification 且又看到 **isForeground=true** 则说明了，此app利用了这种灰色保活的手段。

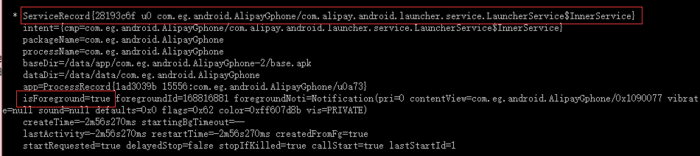
下面分别是我手机上微信、qq、支付宝、陌陌的测试结果，大家有兴趣也可以自己验证一下。



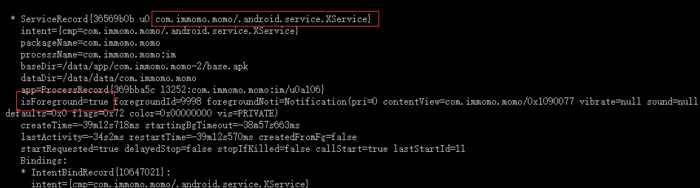
微信



手Q

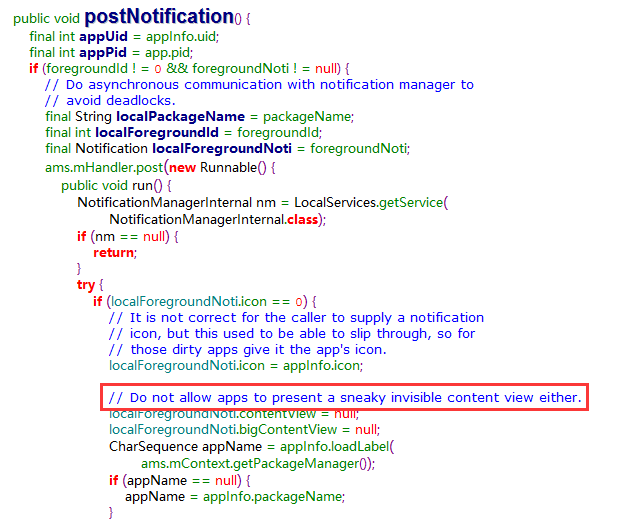


支付宝



陌陌

其实Google察觉到了此漏洞的存在，并逐步进行封堵。这就是为什么这种保活方式分 API >= 18 和 API < 18 两种情况，从Android5.0的ServiceRecord类的postNotification函数源代码中可以看到这样的一行注释



当某一天 API >= 18 的方案也失效的时候，我们就又要另谋出路了。需要注意的是，**使用灰色保活并不代表着你的Service就永生不死了，只能说是提高了进程的优先级。如果你的app进程占用了大量的内存，按照回收进程的策略，同样会干掉你的app。**感兴趣于灰色保活是如何利用系统漏洞不显示 Notification 的童鞋，可以研究一下系统的 ServiceRecord、NotificationManagerService 等相关源代码，因为不是本文的重点，所以不做详述。

**唠叨的分割线**

到这里基本就介绍完了\*\* 黑、白、灰 \*\*三种实现方式，仅仅从代码层面去讲保活是不够的，我希望能够通过系统的进程回收机制来理解保活，这样能够让我们更好的避免踩到进程被杀的坑。

**进程回收机制**

熟悉Android系统的童鞋都知道，系统出于体验和性能上的考虑，app在退到后台时系统并不会真正的kill掉这个进程，而是将其缓存起来。打开的应用越多，后台缓存的进程也越多。在系统内存不足的情况下，系统开始依据自身的一套进程回收机制来判断要kill掉哪些进程，以腾出内存来供给需要的app。这套杀进程回收内存的机制就叫 **Low Memory Killer** ，它是基于Linux内核的 **OOM Killer（Out-Of-Memory killer）**机制诞生。

了解完 **Low Memory Killer**，再科普一下**oom\_adj**。什么是**oom\_adj**？它是linux内核分配给每个系统进程的一个值，代表进程的优先级，进程回收机制就是根据这个优先级来决定是否进行回收。对于**oom\_adj**的作用，你只需要记住以下几点即可：

* **进程的oom\_adj越大，表示此进程优先级越低，越容易被杀回收；越小，表示进程优先级越高，越不容易被杀回收**
* **普通app进程的oom\_adj>=0,系统进程的oom\_adj才可能<0**

那么我们如何查看进程的**oom\_adj**值呢，需要用到下面的两个shell命令

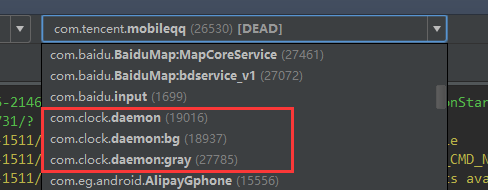
ps | grep PackageName //获取你指定的进程信息



这里是以我写的demo代码为例子，红色圈中部分别为下面三个进程的ID

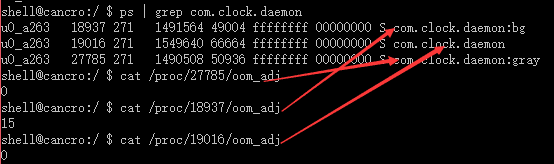
UI进程：**com.clock.daemon**  
普通后台进程：**com.clock.daemon:bg**  
灰色保活进程：**com.clock.daemon:gray**

当然，这些进程的id也可以通过AndroidStudio获得

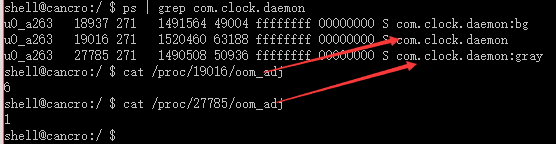


接着我们来再来获取三个进程的**oom\_adj**

cat /proc/进程ID/oom\_adj



从上图可以看到UI进程和灰色保活Service进程的**oom\_adj=0**，而普通后台进程**oom\_adj=15**。到这里估计你也能明白，**为什么普通的后台进程容易被回收，而前台进程则不容易被回收了吧。**但明白这个还不够，接着看下图



上面是我把app切换到后台，再进行一次**oom\_adj**的检验，你会发现UI进程的值从0变成了6,而灰色保活的Service进程则从0变成了1。这里可以观察到，**app退到后台时，其所有的进程优先级都会降低。但是UI进程是降低最为明显的，因为它占用的内存资源最多，系统内存不足的时候肯定优先杀这些占用内存高的进程来腾出资源。所以，为了尽量避免后台UI进程被杀，需要尽可能的释放一些不用的资源，尤其是图片、音视频之类的**。

从Android官方文档中，我们也能看到优先级从高到低列出了这些不同类型的进程：**Foreground process**、**Visible process**、**Service process**、**Background process**、**Empty process**。而这些进程的oom\_adj分别是多少，又是如何挂钩起来的呢？推荐大家阅读下面这篇文章：

[http://www.cnblogs.com/angeldevil/archive/2013/05/21/3090872.html](https://link.jianshu.com?t=http:/www.cnblogs.com/angeldevil/archive/2013/05/21/3090872.html)

**总结（文末有福利）**

絮絮叨叨写完了这么多，最后来做个小小的总结。回归到开篇提到QQ进程不死的问题，我也曾认为存在这样一种技术。可惜我把手机root后，杀掉QQ进程之后就再也起不来了。有些手机厂商把这些知名的app放入了自己的白名单中，保证了进程不死来提高用户体验（如微信、QQ、陌陌都在小米的白名单中）。如果从白名单中移除，他们终究还是和普通app一样躲避不了被杀的命运，为了尽量避免被杀，还是老老实实去做好优化工作吧。

**所以，进程保活的根本方案终究还是回到了性能优化上，进程永生不死终究是个彻头彻尾的伪命题！**

**补充更新 （2016-04-20）**

**有童鞋问，在华为的机子上发现微信和手Q的UI进程退到后台，oom\_adj的值一点都没有变，是不是有什么黑科技在其中。为此，我稍稍验证了一下，验证方式就是把demo工程的包名改成手机QQ的，编译运行在华为的机子上，发现我的进程怎么杀也都是不死的，退到后台oom\_adj的值同样不发生变化，而恢复原来的包名就不行了。所以，你懂的，手Q就在华为机子的白名单中。**

文章到此结束，相关简单的实践代码请看

[https://github.com/D-clock/AndroidDaemonService](https://link.jianshu.com?t=https:/github.com/D-clock/AndroidDaemonService)

# Service防杀死

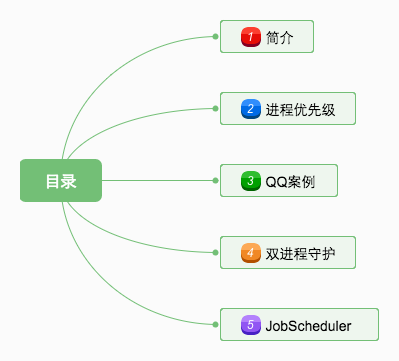
# https://www.jianshu.com/p/ced404f406b8

[](https://www.jianshu.com/u/1fa8a6a319f4)

[慕涵盛华](https://www.jianshu.com/u/1fa8a6a319f4)

2017.08.31 20:19\* 字数 1676 阅读 469评论 12喜欢 26

# 目录



# 简介

首先要弄明确一个问题，就是我们平时说的Service防杀死，其实防止是Service所在的进程被杀死，而不是Service这个组件，因为Android 只杀死进程，而不是组件。  
当我们说进程优先级的时候是以 activity、service 这样的组件来说的，这些组件的优先级是在进程的级别上，不是组件级别上。只要一个组件的状态发生变化，就会影响进程的优先级；比如：启动一个前台服务，那么就会将该服务所在的整个进程变为前台进程。  
弄清楚了这个问题后，下面我们首先了解一下进程的优先级。

# 进程优先级

前台进程 > 可见进程 > 服务进程 > 后台进程 > 空进程。

##### 1.前台进程；Foreground process

* 用户正在交互的Activity（onResume（））
* 当某个Service绑定正在交互的Activity。
* 被主动调用为前台Service（startForeground()）
* 组件正在执行生命周期的回调（onCreate()/onStart()/onDestroy()）
* BroadcastReceiver 正在执行onReceive();

##### 2.可见进程；Visible process

* 我们的Activity处在onPause()（没有进入onStop()）
* 绑定到前台Activity的Service。

##### 3.服务进程；Service process

简单的startService()启动。

##### 4.后台进程；Background process

对用户没有直接影响的进程----Activity出于onStop()的时候。  
android:process=":xxx"

##### 5.空进程； Empty process

不含有任何的活动的组件。（android设计的，为了第二次启动更快）

# 防止进程被杀死

所谓进程防杀死，就是做到进程尽量不被系统杀死，并不能保证100%存活，因为受到内存，手机厂商的限制等。上面提到进程优先级，优先级越高越不容易被杀死，所以要想防止进程被杀死，就要提高进程的优先级。

#### QQ的做法

QQ采取在锁屏的时候启动一个1个像素的Activity，当用户解锁以后将这个Activity结束掉，同时把自己的核心服务再开启一次。下面我们就简单模拟一下。

* 在首页我们放置一个按钮，点击按钮启动1个像素的Activity
* /\*\* 启动一个像素的Activity \*/
* public void start(View view){
* Intent intent = new Intent(MainActivity.this, LiveActivity.class);
* startActivity(intent);
* finish();
* Toast.makeText(this, "启动完成", Toast.LENGTH\_SHORT).show();
* }
* 设置Activity为一个像素，首先我们要将Activity的背景设置为透明，否则显示的是黑色
* <style name="LiveStyle" parent="Theme.AppCompat.Light.NoActionBar">
* <item name="android:windowBackground">@android:color/transparent</item>
* <item name="android:windowFrame">@null</item>
* <item name="android:windowNoTitle">true</item>
* <item name="android:windowIsFloating">true</item>
* <item name="android:backgroundDimEnabled">false</item>
* <item name="android:windowContentOverlay">@null</item>
* <item name="android:windowIsTranslucent">true</item>
* <item name="android:windowAnimationStyle">@null</item>
* <item name="android:windowDisablePreview">true</item>
* <item name="android:windowNoDisplay">false</item>
* </style>
* 显示一个像素
* @Override
* protected void onCreate(@Nullable Bundle savedInstanceState) {
* super.onCreate(savedInstanceState);
* Window window = getWindow();
* window.setGravity(Gravity.LEFT | Gravity.TOP);
* WindowManager.LayoutParams params = window.getAttributes();
* params.width = 1; // 1 px
* params.height = 1;
* params.x = 0;
* params.y = 0;
* window.setAttributes(params);
* }

从效果图中看出，我们启动Activity后，手机页面不能滑动，当点击返回键的时候，又能滑动了，说明成功启动了1px 的Activity，下面监听手机锁屏事件。

封装一个工具类，专门监听锁屏事件

public class ScreenListener {

private Context mContext;

private ScreenBroadcastReceiver mScreenReceiver;

private ScreenStateListener mScreenStateListener;

public ScreenListener(Context context) {

mContext = context;

mScreenReceiver = new ScreenBroadcastReceiver();

}

//监听锁屏事件的广播接受者

private class ScreenBroadcastReceiver extends BroadcastReceiver {

private String action = null;

@Override

public void onReceive(Context context, Intent intent) {

action = intent.getAction();

if (Intent.ACTION\_SCREEN\_ON.equals(action)) {//开屏

mScreenStateListener.onScreenOn();

} else if (Intent.ACTION\_SCREEN\_OFF.equals(action)) { //锁屏

mScreenStateListener.onScreenOff();

} else if (Intent.ACTION\_USER\_PRESENT.equals(action)) { //解锁

mScreenStateListener.onUserPresent();

}

}

}

//初始化

public void init(ScreenStateListener listener) {

mScreenStateListener = listener;

registerListener();

getScreenState();

}

//根据锁屏状态回调对应的方法

private void getScreenState() {

PowerManager manager = (PowerManager) mContext

.getSystemService(Context.POWER\_SERVICE);

if (manager.isScreenOn()) {

if (mScreenStateListener != null) {

mScreenStateListener.onScreenOn();

}

} else {

if (mScreenStateListener != null) {

mScreenStateListener.onScreenOff();

}

}

}

public void unregisterListener() {

mContext.unregisterReceiver(mScreenReceiver);

}

//动态注册锁屏广播

private void registerListener() {

IntentFilter filter = new IntentFilter();

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

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

filter.addAction(Intent.ACTION\_USER\_PRESENT);

mContext.registerReceiver(mScreenReceiver, filter);

}

//锁屏事件回调接口

public interface ScreenStateListener {

void onScreenOn();

void onScreenOff();

void onUserPresent();

}

}

启动一个Service专门管理Activity

public class MyService extends Service{

@Override

public IBinder onBind(Intent intent) {

return null;

}

@Override

public void onCreate() {

super.onCreate();

ScreenListener listener = new ScreenListener(this);

listener.init(new ScreenListener.ScreenStateListener() {

@Override

public void onScreenOn() {

//开屏，销毁1px Activity

LiveActivityManager.getInstance(MyService.this).finishKeepLiveActivity();

}

@Override

public void onScreenOff() {

//锁屏，启动 1px Activity

LiveActivityManager.getInstance(MyService.this).startKeepLiveActivity();

}

@Override

public void onUserPresent() { //解锁 }

});

}

}

在应用启动的时候启动该服务

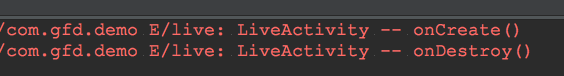
Intent intent = new Intent(this, MyService.class);

startService(intent);

添加权限

<uses-permission android:name="android.permission.RECEIVE\_BOOT\_COMPLETED"/>

<uses-permission android:name="android.permission.RECEIVE\_USER\_PRESENT"/>



通过日志的输出可以看出，在锁屏的时候调用了onCreate(),在亮屏的时候调用了onDestroy();

上面只是简单的模拟了一下，但是实现上述功能还有一个前提就是MyService这个服务得一直存活，不然没办法监听处理了。

# 双进程守护

双进程守护，可以防止单个进程杀死，同时可以防止第三方的软件清理掉。一个进程被杀死，另外一个进程又被他启动。相互监听启动，因为杀进程是一个一个杀的。本质是和杀进程时间赛跑。

* 1.创建两个Service，实现互相监听
* <service android:name=".LocalService"/>
* <service
* android:name=".RemoteService"
* android:process=":remote"/> // 开启一个新的进程

LocalService和RemoteService位于不同的进程中，实现互相的监听，他们的代码一样，只不过是监听的对象不一样，所以下面只贴出了一个文件的代码。

#### LocalService.java

public class LocalService extends Service {

private MyConnection connection;

private MyBinder binder;

@Nullable

@Override

public IBinder onBind(Intent intent) {

return binder;

}

@Override

public void onCreate() {

super.onCreate();

connection = new MyConnection();

binder = new MyBinder();

}

@Override

public int onStartCommand(Intent intent,int flags, int startId) {

//启动的时候绑定RemoteService

LocalService.this.bindService(new Intent(LocalService.this,RemoteService.class),connection,

Context.BIND\_IMPORTANT);

return START\_STICKY;

}

// 绑定RemoteService建立的链接

class MyConnection implements ServiceConnection{

@Override

public void onServiceConnected(ComponentName name, IBinder service) {

Log.e("service","绑定RemoteService");

}

@Override

public void onServiceDisconnected(ComponentName name) {

//RemoteService被杀死，重新启动绑定

Intent service = new Intent(LocalService.this,RemoteService.class);

LocalService.this.startService(service);

LocalService.this.bindService(service,connection, Context.BIND\_IMPORTANT);

}

}

//LocalService和RemoteService通信的Binder

class MyBinder extends RemoteInterface.Stub{

@Override

public String getServicName() throws RemoteException {

return "LocalService";

}

}

@Override

public void onDestroy() {

super.onDestroy();

//销毁的时候要解绑

unbindService(connection);

}

}

从上述代码中可以看出，在LocalService一启动的时候（onStartCommand），去绑定RemoteService,对应的连接为connection，那么当RemoteService被解绑的时候就会调用onServiceDisconnected方法，在该方法只能再次启动和绑定RemoteService。同理在RemoteService中也是一样的逻辑，这样就能实现两个服务互相监听，一个被杀死，另一个立马再次启动它。

从效果图中看出，我们关闭一个服务，又会被立刻启动起来，通过右边启动的时间也可以看出来，如果想提高进程的优先级，可以在onStartCommand()方法中调用 startForeground()将进程提升为前台进程。这样虽然能实现进程常驻，但是两个服务一直在后台运行，是非常耗费资源的，尤其是电量。那么有没有更好一点的办法呢？那就是下面要说的内容了。

# Jobscheduler

Android 5.0系统以后，Google为了优化Android系统，提高使用流畅度以及延长电池续航，加入了在应用后台/锁屏时，系统会回收应用，同时自动销毁应用拉起的Service的机制。同时为了满足在特定条件下需要执行某些任务的需求，google在全新一代操作系统上，采取了Job （jobservice & JobInfo）的方式，即每个需要后台的业务处理为一个job，通过系统管理job，来提高资源的利用率，从而提高性能，节省电源。这样又能满足APP开发的要求，又能满足系统性能的要求。

#### 实现进程防杀死

把任务加到系统调度队列中，当到达任务窗口期的时候就会执行，我们可以在这个任务里面启动我们的进程。这样可以做到将近杀不死的进程。

#### 具体实现

* 派生JobService 子类，定义需要执行的任务
* 从Context 中获取JobScheduler 实例
* 构建JobInfo 实例，指定 JobService任务实现类及其执行条件
* 通过JobScheduler 实例加入到任务队列

@SuppressLint("NewApi")

public class JobHandleService extends JobService{

private int kJobId = 0;

@Override

public void onCreate() {

super.onCreate();

}

@Override

public int onStartCommand(Intent intent, int flags, int startId) {

scheduleJob(getJobInfo());

return START\_NOT\_STICKY;

}

@Override

public boolean onStartJob(JobParameters params) {

boolean isLocalServiceWork = isServiceWork(this, "com.gfd.demo.LocalService");

boolean isRemoteServiceWork = isServiceWork(this, "com.gfd.demo.RemoteService");

if(!isLocalServiceWork||

!isRemoteServiceWork){

this.startService(new Intent(this,LocalService.class));

this.startService(new Intent(this,RemoteService.class));

}

return true;

}

@Override

public boolean onStopJob(JobParameters params) {

scheduleJob(getJobInfo());

return true;

}

public void scheduleJob(JobInfo t) {

JobScheduler tm =

(JobScheduler) getSystemService(Context.JOB\_SCHEDULER\_SERVICE);

tm.schedule(t);

}

public JobInfo getJobInfo(){

JobInfo.Builder builder = new JobInfo.Builder(kJobId++, new ComponentName(this, JobHandleService.class));

builder.setRequiredNetworkType(JobInfo.NETWORK\_TYPE\_ANY);

builder.setPersisted(true);

builder.setRequiresCharging(false);

builder.setRequiresDeviceIdle(false);

builder.setPeriodic(10);

}

//判断Service是否在存活

public boolean isServiceWork(Context mContext, String serviceName) {

boolean isWork = false;

ActivityManager myAM = (ActivityManager) mContext

.getSystemService(Context.ACTIVITY\_SERVICE);

List<RunningServiceInfo> myList = myAM.getRunningServices(100);

if (myList.size() <= 0) {

return false;

}

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

String mName = myList.get(i).service.getClassName().toString();

if (mName.equals(serviceName)) {

isWork = true;

break;

}

}

return isWork;

}

}

[**Android开发之怎样监听让Service不被杀死**](https://www.cnblogs.com/wzjhoutai/p/7105038.html)

**https://www.cnblogs.com/wzjhoutai/p/7105038.html**

**一、Service简单介绍**

Service是在一段不定的时间执行在后台，不和用户交互应用组件。

每一个Service必须在manifest中 通过<service>来声明。

能够通过contect.startservice和contect.bindserverice来启动。和其它的应用组件一样，执行在进程的主线程中。这就是说假设service须要非常多耗时或者堵塞的操作，须要在其子线程中实现（或者用系统提供的IntentService，它继承了Service，它处理数据是用自身新开的线程）。

**本地服务** Local Service 用于应用程序内部。

它能够启动并执行，直至有人停止了它或它自己停止。

在这样的方式下。它以调用Context.startService()启动。而以调用Context.stopService()结束。它能够调用Service.stopSelf() 或 Service.stopSelfResult()来自己停止。不论调用了多少次startService()方法，你仅仅须要调用一次stopService()来停止服务。

用于实现应用程序自己的一些耗时任务。比方查询升级信息，并不占用应用程序比方Activity所属线程，而是单开线程后台运行，这样用户体验比較好。

**远程服务** Remote Service 用于android系统内部的应用程序之间。

它能够通过自定义并暴露出来的接口进行程序操作。client建立一个到服务对象的连接。并通过那个连接来调用服务。连接以调用Context.bindService()方法建立，以调用 Context.unbindService()关闭。

多个client能够绑定至同一个服务。假设服务此时还没有载入，bindService()会先载入它。

可被其它应用程序复用。比方天气预报服务。其它应用程序不须要再写这种服务，调用已有的就可以。

1。Service的生命周期

2，Service执行方式

以startService()启动service，系统将通过传入的Intent在底层搜索相关符合Intent里面信息的service。

假设服务没有启动则先执行onCreate，然后执行onStartCommand （可在里面处理启动时传过来的Intent和其它參数），直到明显调用stopService或者stopSelf才将停止Service。

不管执行startService多少次，仅仅要调用一次stopService或者stopSelf，Service都会停止。使用stopSelf(int)方法能够保证在处理好intent后再停止。  
onStartCommand ，在2.0后被引入用于service的启动函数，2.0之前为public void onStart(Intent intent, int startId) 。

使用bindService()方法启用服务。调用者与服务绑定在了一起，调用者一旦退出，服务也就终止。

onBind()仅仅有採用Context.bindService()方法启动服务时才会回调该方法。该方法在调用者与服务绑定时被调用，当调用者与服务已经绑定，多次调用Context.bindService()方法并不会导致该方法被多次调用。

採用Context.bindService()方法启动服务时仅仅能调用onUnbind()方法解除调用者与服务解除。服务结束时会调用onDestroy()方法。

3，拥有service的进程具有较高的优先级

官方文档告诉我们，Android系统会尽量保持拥有service的进程执行，仅仅要在该service已经被启动(start)或者client连接(bindService)到它。

当内存不足时，须要保持。拥有service的进程具有较高的优先级。

1． 假设service正在调用onCreate,onStartCommand或者onDestory方法。那么用于当前service的进程则变为前台进程以避免被killed。

2． 假设当前service已经被启动(start)。拥有它的进程则比那些用户可见的进程优先级低一些，可是比那些不可见的进程更重要，这就意味着service一般不会被killed.  
3． 假设client已经连接到service (bindService),那么拥有Service的进程则拥有最高的优先级，能够觉得service是可见的。  
4． 假设service能够使用startForeground(int, Notification)方法来将service设置为前台状态，那么系统就觉得是对用户可见的。并不会在内存不足时killed。  
假设有其它的应用组件作为Service,Activity等执行在同样的进程中。那么将会添加该进程的重要性。

**二、保证service不死**

onStartCommand方法，返回START\_STICKY。

1、START\_STICKY

在执行onStartCommand后service进程被kill后，那将保留在開始状态，可是不保留那些传入的intent。不久后service就会再次尝试又一次创建，由于保留在開始状态，在创建     service后将保证调用onstartCommand。

假设没有传递不论什么開始命令给service。那将获取到null的intent。

2、START\_NOT\_STICKY

在执行onStartCommand后service进程被kill后，而且没有新的intent传递给它。Service将移出開始状态，而且直到新的明显的方法（startService）调用才又一次创建。

由于假设没有传递不论什么未决定的intent那么service是不会启动，也就是期间onstartCommand不会接收到不论什么null的intent。  
3、START\_REDELIVER\_INTENT

在执行onStartCommand后service进程被kill后。系统将会再次启动service。并传入最后一个intent给onstartCommand。

直到调用stopSelf(int)才停止传递intent。假设在被kill后还有未处理好的intent。那被kill后服务还是会自己主动启动。因此onstartCommand不会接收到不论什么null的intent。

**[java]** [view plaincopy](http://blog.csdn.net/mad1989/article/details/22492519)[在CODE上查看代码片](https://code.csdn.net/snippets/263696)

1. @Override
2. **public** **int** onStartCommand(Intent intent, **int** flags, **int** startId) {
3. flags = START\_STICKY;
4. **return** **super**.onStartCommand(intent, flags, startId);
5. }

【结论】 手动返回START\_STICKY。亲測当service因内存不足被kill。当内存又有的时候。service又被又一次创建，比較不错。

提升service进程优先级

在AndroidManifest.xml文件里对于intent-filter能够通过android:priority = "1000"这个属性设置最高优先级，1000是最高值，假设数字越小则优先级越低，同一时候有用于广播

**[java]** [view plaincopy](http://blog.csdn.net/mad1989/article/details/22492519)[在CODE上查看代码片](https://code.csdn.net/snippets/263696)

1. <service
2. android:name="com.dbjtech.acbxt.waiqin.UploadService"
3. android:enabled="true" >
4. <intent-filter android:priority="1000" >
5. <action android:name="com.dbjtech.myservice" />
6. </intent-filter>
7. </service>

提升service优先级

Android中的进程是托管的，当系统进程空间紧张的时候，会按照优先级自己主动进行进程的回收。Android将进程分为6个等级,它们按优先级顺序由高到低依次是:

   1.前台进程( FOREGROUND\_APP)  
   2.可视进程(VISIBLE\_APP )  
   3. 次要服务进程(SECONDARY\_SERVER )  
   4.后台进程 (HIDDEN\_APP)  
   5.内容供应节点(CONTENT\_PROVIDER)  
   6.空进程(EMPTY\_APP)

当service执行在低内存的环境时，将会kill掉一些存在的进程。因此进程的优先级将会非常重要，能够使用startForeground API将service放到前台状态。这样在低内存时被kill的几率更低。可是假设在极度极度低内存的压力下，该service还是会被kill掉。

在onStartCommand方法内

**[java]** [view plaincopy](http://blog.csdn.net/mad1989/article/details/22492519)[在CODE上查看代码片](https://code.csdn.net/snippets/263696)

1. Notification notification = **new** Notification(R.drawable.ic\_launcher,
2. getString(R.string.app\_name), System.currentTimeMillis());
4. PendingIntent pendingintent = PendingIntent.getActivity(**this**, 0,
5. **new** Intent(**this**, AppMain.**class**), 0);
6. notification.setLatestEventInfo(**this**, "uploadservice", "请保持程序在后台执行",
7. pendingintent);
8. startForeground(0x111, notification);

注意在onDestroy里还须要stopForeground(true);

在onDestroy方法里重新启动service

service +broadcast  方式，就是当service走ondestory的时候，发送一个自己定义的广播。当收到广播的时候。又一次启动service；

**[java]** [view plaincopy](http://blog.csdn.net/mad1989/article/details/22492519)[在CODE上查看代码片](https://code.csdn.net/snippets/263696)

1. <receiver android:name="com.dbjtech.acbxt.waiqin.BootReceiver" >
2. <intent-filter>
3. <action android:name="android.intent.action.BOOT\_COMPLETED" />
4. <action android:name="android.intent.action.USER\_PRESENT" />
5. <action android:name="com.dbjtech.waiqin.destroy" />//这个就是自己定义的action
6. </intent-filter>
7. </receiver>

在onDestroy时：

**[java]** [view plaincopy](http://blog.csdn.net/mad1989/article/details/22492519)[在CODE上查看代码片](https://code.csdn.net/snippets/263696)

1. @Override
2. **public** **void** onDestroy() {
3. stopForeground(**true**);
4. Intent intent = **new** Intent("com.dbjtech.waiqin.destroy");
5. sendBroadcast(intent);
6. **super**.onDestroy();
7. }

在BootReceiver里

**[java]** [view plaincopy](http://blog.csdn.net/mad1989/article/details/22492519)[在CODE上查看代码片](https://code.csdn.net/snippets/263696)

1. **public** **class** BootReceiver **extends** BroadcastReceiver {
3. @Override
4. **public** **void** onReceive(Context context, Intent intent) {
5. **if** (intent.getAction().equals("com.dbjtech.waiqin.destroy")) {
6. //TODO
7. //在这里写又一次启动service的相关操作
8. startUploadService(context);
9. }
11. }
13. }

【结论】当然，你也能够在onDestroy里直接startService()，我通过使用broadcast，能够做很多其它的处理推断操作。

一个不被杀死的进程

看Android的文档知道，当进程长期不活动。或系统须要资源时，会自己主动清理门户，杀死一些Service，和不可见的Activity等所在的进程。

可是假设某个进程不想被杀死（如数据缓存进程，或状态监控进程。或远程服务进程），能够这么做：

**[java]** [view plaincopy](http://blog.csdn.net/mad1989/article/details/22492519)[在CODE上查看代码片](https://code.csdn.net/snippets/263696)

1. <application
2. android:name="com.test.Application"
3. android:allowBackup="true"
4. android:icon="@drawable/ic\_launcher"
5. android:label="@string/app\_name"
6. android:persistent="true"
7. android:theme="@style/AppTheme" >
8. </application>

【结论】据说这个属性不能乱设置，只是设置后，的确发现优先级提高不少，也许是相当于系统级的进程。

系统各种广播监听

通过系统的广播，监听并捕获到。然后推断是否须要又一次启动service。

**[java]** [view plaincopy](http://blog.csdn.net/mad1989/article/details/22492519)[在CODE上查看代码片](https://code.csdn.net/snippets/263696)

1. <uses-permission android:name="android.permission.RECEIVE\_BOOT\_COMPLETED" />  // 开机启动

**[java]** [view plaincopy](http://blog.csdn.net/mad1989/article/details/22492519)[在CODE上查看代码片](https://code.csdn.net/snippets/263696)

1. <receiver android:name="com.dbjtech.acbxt.waiqin.BootReceiver" >
2. <intent-filter>
3. <action android:name="android.intent.action.BOOT\_COMPLETED" />
4. <action android:name="android.intent.action.USER\_PRESENT" />
5. <action android:name="com.dbjtech.waiqin.destroy" />
6. </intent-filter>
7. </receiver>

**[java]** [view plaincopy](http://blog.csdn.net/mad1989/article/details/22492519)[在CODE上查看代码片](https://code.csdn.net/snippets/263696)

1. @Override
2. **public** **void** onReceive(Context context, Intent intent) {
3. **if** (Intent.ACTION\_BOOT\_COMPLETED.equals(intent.getAction())) {
4. System.out.println("手机开机了....");
5. startUploadService(context);
6. }
7. **if** (Intent.ACTION\_USER\_PRESENT.equals(intent.getAction())) {
8. startUploadService(context);
9. }
10. }