### App保活

主要通过下面两个方法：

1，降低omm\_adj(五种进程状态的值)值，尽量保证进行不被系统杀死

2，进程被杀死后，通过其他方式使它复活

一些常用的APP保活的方式：

监听锁屏，监听网络等系统广播，将进程置于前台以提高进程的级别，从而防止进行不那么容易被杀死。

### 降低omm\_adj值

1. **开启前台service**

将Service置为前台，目的时提高进程Service的oom\_adj值，以降低其被系统回收的几率。该方案的原理是，通过使用 startForeground()方法将当前Service置于前台来提高Service的优先级。

需要注意的是，对API大于18而言 startForeground()方法需要弹出一个可见通知，如果你觉得不爽，可以开启另一个Service将通知栏移除，其oom\_adj值还是没变的。

实现代码如下：

/\*\*前台Service，使用startForeground   
 \* 这个Service尽量要轻，不要占用过多的系统资源，否则   
 \* 系统在资源紧张时，照样会将其杀死   
 \*   
 \* Created by jianddongguo on 2017/7/7.   
 \* http://blog.csdn.net/andrexpert   
 \*/    
public class DaemonService extends Service {    
    private static final String TAG = "DaemonService";    
    public static final int NOTICE\_ID = 100;    
    @Nullable    
    @Override    
    public IBinder onBind(Intent intent) {    
        return null;    
    }    
    @Override    
    public void onCreate() {    
        super.onCreate();    
        if(Contants.DEBUG)    
            Log.d(TAG,"DaemonService---->onCreate被调用，启动前台service");    
        //如果API大于18，需要弹出一个可见通知    
        if(Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.JELLY\_BEAN\_MR2){    
            Notification.Builder builder = new Notification.Builder(this);    
            builder.setSmallIcon(R.mipmap.ic\_launcher);    
            builder.setContentTitle("KeepAppAlive");    
            builder.setContentText("DaemonService is runing...");    
            startForeground(NOTICE\_ID,builder.build());    
            // 如果觉得常驻通知栏体验不好    
            // 可以通过启动CancelNoticeService，将通知移除，oom\_adj值不变    
            Intent intent = new Intent(this,CancelNoticeService.class);    
            startService(intent);    
        }else{    
            startForeground(NOTICE\_ID,new Notification());    
        }    
    }    
    @Override    
    public int onStartCommand(Intent intent, int flags, int startId) {    
        // 如果Service被终止    
        // 当资源允许情况下，重启service    
        return START\_STICKY;    
    }    
    @Override    
    public void onDestroy() {    
        super.onDestroy();    
        // 如果Service被杀死，干掉通知    
        if(Build.VERSION.SDK\_INT   
                >= Build.VERSION\_CODES.JELLY\_BEAN\_MR2){    
            NotificationManager mManager =   
                (NotificationManager)getSystemService(NOTIFICATION\_SERVICE);    
            mManager.cancel(NOTICE\_ID);    
        }    
        if(Contants.DEBUG)    
            Log.d(TAG,"DaemonService---->onDestroy，前台service被杀死");    
        // 重启自己    
        Intent intent = new Intent(getApplicationContext(),DaemonService.class);    
        startService(intent);    
    }    
}

这里还用到了两个技巧：一是在onStartCommand方法中返回START\_STICKY，其作用是当Service进程被kill后，系统会尝试重新创建这个Service，且会保留Service的状态为开始状态，但不保留传递的Intent对象，onStartCommand方法一定会被重新调用。

其二在onDestory方法中重新启动自己，也就是说，只要Service在被销毁时走到了onDestory这里我们就重新启动它。

1. **监听锁屏广播**

由于静态注册广播接收器，无法接收到系统的锁屏(Intent.ACTION\_SCREEN\_OFF)和开屏(Intent.ACTION\_SCREEN\_ON)广播，因此必须通过动态注册来监听。另外，这里还使用了接口将监听的结果回调给调用者。

Java中为对象的引用分了四个级别：强引用、软引用、弱引用、虚引用。这里，我们使用了弱引用WeakReference来防止内存泄漏，为了解释这个问题，我们举这么一个例子：有两个类class A和class B，分别实例化这两个类得到a,b，其中a又作为实例化B时传入的构造参数，代码如下：

A a = new A();  
B b = new B(a);

从这两行代码来看，a是对象A的引用，b是对象B的引用，对象B同时依赖于对象A，对象A和对象B之间形成了强引用。当a=null时，a不在指向对象A，通常情况下，对象A在不被其他对象引用时会被GC回收，但是由于B还依赖于对象A，对象A不会被GC回收，从而造成内存泄漏(除非b=null，对象A和对象B才会被GC同时回收)。

如果使用弱引用的话，对象A只会被WeakReference所依赖，当a=null时，GC会回收它，从而避免了内存泄漏。

1. **循环播放一段无声音频，及其恶心**