

Android 进程保活招式大全

目前市面上的应用，貌似除了微信和手 Q 都会比较担心被用户或者系统（厂商）杀死问题。本文对 Android 进程拉活进行一个总结。

Android 进程拉活包括两个层面：

- A. 提供进程优先级，降低进程被杀死的概率
- B. 在进程被杀死后，进行拉活

本文下面就从这两个方面做一下总结。

1. 进程的优先级

Android 系统将尽量长时间地保持应用进程，但为了新建进程或运行更重要的进程，最终需要清除旧进程来回收内存。为了确定保留或终止哪些进程，系统会根据进程中正在运行的组件以及这些组件的状态，将每个进程放入“重要性层次结构”中。必要时，系统会首先消除重要性最低的进程，然后是清除重要性稍低一级的进程，依此类推，以回收系统资源。

进程的重要性，划分 5 级：

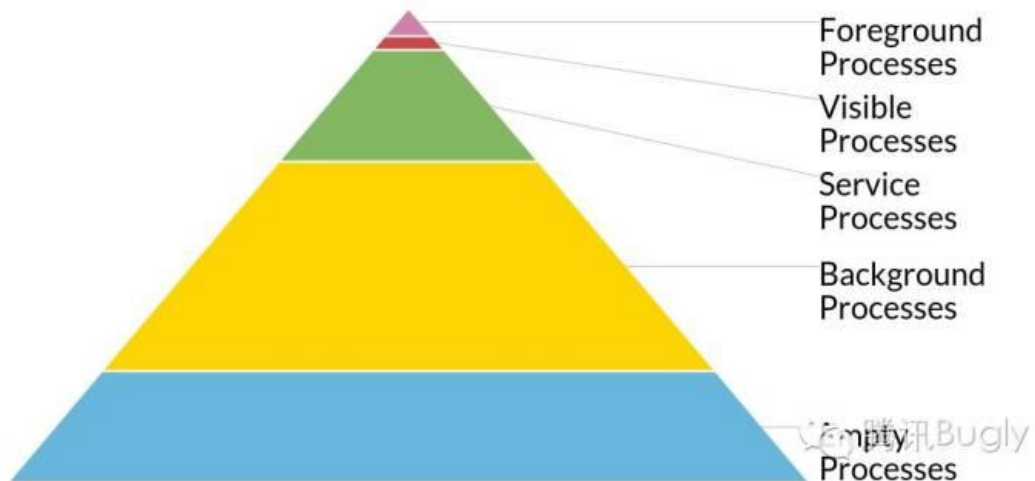
前台进程(Foreground process)

可见进程(Visible process)

服务进程(Service process)

后台进程(Background process)

空进程(Empty process)



前台进程的重要性最高，依次递减，空进程的重要性最低，下面分别来阐述每种级别的进程

1.1. 前台进程 —— Foreground process

用户当前操作所必需的进程。通常在任意给定时间前台进程都为数不多。只有在内存不足以支持它们同时继续运行这一万不得已的情况下，系统才会终止它们。

- A. 拥有用户正在交互的 Activity（已调用 `onResume()`）
- B. 拥有某个 Service，后者绑定到用户正在交互的 Activity
- C. 拥有正在“前台”运行的 Service（服务已调用 `startForeground()`）
- D. 拥有正执行一个生命周期回调的 Service（`onCreate()`、`onStart()` 或 `onDestroy()`）
- E. 拥有正执行其 `onReceive()` 方法的 BroadcastReceiver

1.2. 可见进程 —— Visible process

没有任何前台组件、但仍会影响用户在屏幕上所见内容的进程。可见进程被视为是极其重要的进程，除非为了维持所有前台进程同时运行而必须终止，否则系统不会终止这些进程。

- A. 拥有不在前台、但仍对用户可见的 Activity（已调用 `onPause()`）。
- B. 拥有绑定到可见（或前台）Activity 的 Service

1.3. 服务进程 —— Service process

尽管服务进程与用户所见内容没有直接关联，但是它们通常在执行一些用户关心的操作（例如，在后台播放音乐或从网络下载数据）。因此，除非内存不足以维持所有前台进程和可见进程同时运行，否则系统会让服务进程保持运行状态。

- A. 正在运行 `startService()` 方法启动的服务，且不属于上述两个更高类别进程的进程。

1.4. 后台进程 —— Background process

后台进程对用户体验没有直接影响，系统可能随时终止它们，以回收内存供前台进程、可见进程或服务进程使用。通常会有很多后台进程在运行，因此它们会保存在 LRU 列表中，以确保包含用户最近查看的 Activity 的进程最后一个被终止。如果某个 Activity 正确实现了生命周期方法，并保存了其当前状态，则终止其进程不会对用户体验产生明显影响，因为当用户导航回该 Activity 时，Activity 会恢复其所有可见状态。

- A. 对用户不可见的 Activity 的进程（已调用 Activity 的 `onStop()` 方法）

1.5. 空进程 —— Empty process

保留这种进程的的唯一目的是用作缓存，以缩短下次在其中运行组件所需的启动时间。为使总体系统资源在进程缓存和底层内核缓存之间保持平衡，系统往往会终止这些进程。

A. 不含任何活动应用组件的进程

详情参见：<http://developer.android.com/intl/zh-cn/guide/components/processes-and-threads.html>

2. Android 进程回收策略

Android 中对于内存的回收，主要依靠 Lowmemorykiller 来完成，是一种根据 OOM_ADJ 阈值级别触发相应力度的内存回收的机制。

关于 OOM_ADJ 的说明如下：

ADJ级别	取值	解释
UNKNOWN_ADJ	16	一般指将要会缓存进程，无法获取确定值
CACHED_APP_MAX_ADJ	15	不可见进程的adj最大值 1
CACHED_APP_MIN_ADJ	9	不可见进程的adj最小值 2
SERVICE_B_AD	8	B List中的Service (较老的、使用可能性更小)
PREVIOUS_APP_ADJ	7	上一个App的进程(往往通过按返回键)
HOME_APP_ADJ	6	Home进程
SERVICE_ADJ	5	服务进程(Service process)
HEAVY_WEIGHT_APP_ADJ	4	后台的重量级进程，system/rootdir/init.rc文件中设置
BACKUP_APP_ADJ	3	备份进程
PERCEPTIBLE_APP_ADJ	2	可感知进程，比如后台音乐播放
VISIBLE_APP_ADJ	1	可见进程(Visible process)
FOREGROUND_APP_ADJ	0	前台进程 (Foreground process)
PERSISTENT_SERVICE_ADJ	-11	关联着系统或persistent进程
PERSISTENT_PROC_ADJ	-12	系统persistent进程，比如telephony
SYSTEM_ADJ	-16	系统进程
NATIVE_ADJ	-17	native进程 (不被系统管理)



其中红色部分代表比较容易被杀死的 Android 进程 (OOM_ADJ>=4),绿色部分表示不容易被杀死的 Android 进程, 其他表示非 Android 进程 (纯 Linux 进程)。在 Lowmemorykiller 回收内存时会根据进程的级别优先杀死 OOM_ADJ 比较大的进程，对于优先级相同的进程则进一步受到进程所占内存和进程存活时间的影响。

Android 手机中进程被杀死可能有如下情况：

进程杀死场景	调用接口	可能影响范围
触发系统进程管理机制	Lowmemorykiller	从进程importance值由大到小依次杀死，释放内存
被第三方应用杀死（无Root）	killBackgroundProcess	只能杀死OOM_ADJ为4以上的进程
被第三方应用杀死（有Root）	force-stop或者kill	理论上可以杀所有进程，一般只杀非系统关键进程和非前台和可见进程
厂商杀进程功能	force-stop或者kill	理论上可以杀所有进程，包括Native进程
用户主动“强行停止”进程	force-stop	只能停用第三方和非system/phone进程应用（停用system进程应用会造成Android重启） 腾讯Bugly

综上，可以得出减少进程被杀死概率无非就是想办法提高进程优先级，减少进程在内存不足等情况下被杀死的概率。

3. 提升进程优先级的方案

3.1. 利用 Activity 提升权限

3.1.1. 方案设计思想

监控手机锁屏解锁事件，在屏幕锁屏时启动 1 个像素的 Activity，在用户解锁时将 Activity 销毁掉。注意该 Activity 需设计成用户无感知。

通过该方案，可以使进程的优先级在屏幕锁屏时间由 4 提升为最高优先级 1。

3.1.2. 方案适用范围

适用场景： 本方案主要解决第三方应用及系统管理工具在检测到锁屏事件后一段时间（一般为 5 分钟以内）内会杀死后台进程，已达到省电的目的问题。

适用版本： 适用于所有的 Android 版本。

3.1.3. 方案具体实现

首先定义 Activity，并设置 Activity 的大小为 1 像素：

```
KeepLiveActivity.java
22     private final static String TAG = "keeplive";
23
24     @Override
25     protected void onCreate(Bundle savedInstanceState) {
26         super.onCreate(savedInstanceState);
27         Log.d(TAG, "LiveActivity -> onCreate");
28
29         RefWatcher refWatcher = FZApplication.getRefWatcher(this);
30         refWatcher.watch(this);
31
32         Window window = getWindow();
33         window.setGravity(Gravity.LEFT | Gravity.TOP);
34         WindowManager.LayoutParams params = window.getAttributes();
35         params.x = 0;
36         params.y = 0;
37         params.height = 1;
38         params.width = 1;
39         window.setAttributes(params);
40
41         FZApplication.self().mLiveActivity = this;
42     }
43
```

其次，从 AndroidManifest 中通过如下属性，排除 Activity 在 RecentTask 中的显示：

```
XLBDaemon Manifest
<activity
    android:name="com.xlb.keeplive.KeepLiveActivity"
    android:configChanges="keyboardHidden/orientation/screenSize/navigation/keyboard"
    android:excludeFromRecents="true"
    android:exported="false"
    android:finishOnTaskLaunch="false"
    android:launchMode="singleInstance"
    android:process=":live"
    android:theme="@style/LiveActivityStyle" />
```

最后，控制 Activity 为透明：


```
LiveActivity.java  XLBDAemon Manifest  styles.xml

<!-- Application theme. -->
<style name="AppTheme" parent="AppBaseTheme">
    <!-- All customizations that are NOT specific to a particular API-level can
</style>

<style name="LiveActivityStyle">
    <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:windowIsTranslucent">true</item>
    <item name="android:windowContentOverlay">@null</item>
    <item name="android:backgroundDimEnabled">false</item>
    <item name="android:windowBackground">@null</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>
```

Activity 启动与销毁时机的控制:

```
public class KeepLiveReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        //Log.i(TAG, "LiveReceiver -> onReceive, action : " + action);

        if (action.equals(Intent.ACTION_SCREEN_OFF)) {
            KeepLiveManager.getInstance().startKeepLiveActivity();
        } else if (action.equals(Intent.ACTION_USER_PRESENT)) {
            KeepLiveManager.getInstance().finishKeepLiveActivity();
        }
        KeepLiveManager.getInstance().startKeepServiceLive();
    }
}
```

3.2. 利用 Notification 提升权限

3.2.1. 方案设计思想

Android 中 Service 的优先级为 4，通过 setForeground 接口可以将后台 Service 设置为前台 Service，使进程的优先级由 4 提升为 2，从而使进程的优先级仅仅低于用户当前正在交互的进程，与可见进程优先级一致，使进程被杀死的概率大大降低。

3.2.2. 方案实现挑战

从 Android2.3 开始调用 setForeground 将后台 Service 设置为前台 Service 时，必须在

系统的通知栏发送一条通知，也就是前台 Service 与一条可见的通知时绑定在一起的。对于不需要常驻通知栏的应用来说，该方案虽好，但却是用户感知的，无法直接使用。

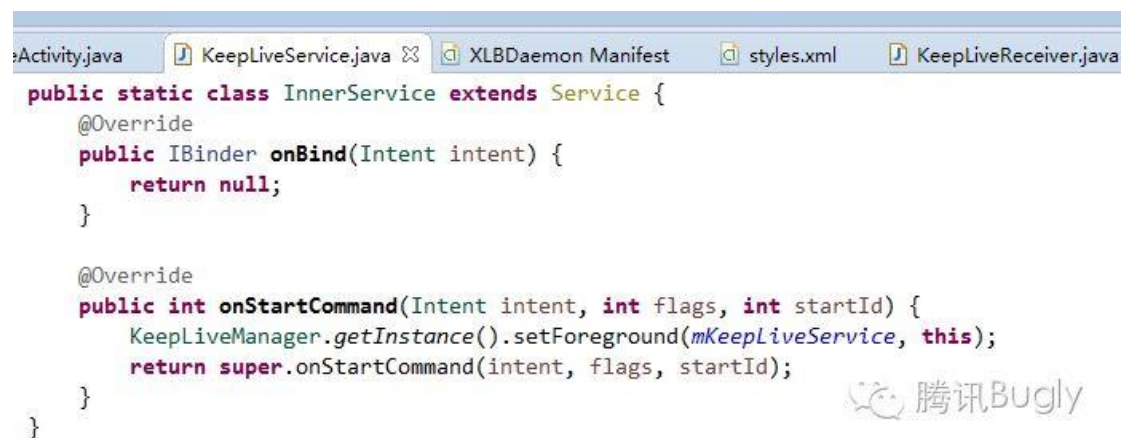
3.2.3. 方案挑战应对措施

通过实现一个内部 Service，在 LiveService 和其内部 Service 中同时发送具有相同 ID 的 Notification，然后将内部 Service 结束掉。随着内部 Service 的结束，Notification 将会消失，但系统优先级依然保持为 2。

3.2.4. 方案适用范围

适用于目前已知所有版本。

3.2.5. 方案具体实现



```
public static class InnerService extends Service {
    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        KeepLiveManager.getInstance().setForeground(mKeepLiveService, this);
        return super.onStartCommand(intent, flags, startId);
    }
}
```

4. 进程死后拉活的方案

4.1. 利用系统广播拉活

4.1.1. 方案设计思想

在发生特定系统事件时，系统会发出响应的广播，通过在 AndroidManifest 中“静态”注册对应的广播监听器，即可在发生响应事件时拉活。

常用的用于拉活的广播事件包括：

开机广播	RECEIVE_BOOT_COMPLETED
网络变化	ACCESS_NETWORK_STATE CHANGE_NETWORK_STATE ACCESS_WIFI_STATE CHANGE_WIFI_STATE ACCESS_FINE_LOCATION ACCESS_LOCATION_EXTRA_COMMANDS
文件挂载	MOUNT_UNMOUNT_FILESYSTEMS
屏幕亮灭	SCREEN_ON SCREEN_OFF
锁屏解锁	RECEIVE_USER_PRESENT
应用安装卸载	PACKAGE_ADDED PACKAGE_REMOVED

4.1.2. 方案适用范围

适用于全部 Android 平台。但存在如下几个缺点：

- 1) 广播接收器被管理软件、系统软件通过“自启管理”等功能禁用的场景无法接收到广播，从而无法自启。
- 2) 系统广播事件不可控，只能保证发生事件时拉活进程，但无法保证进程挂掉后立即拉活。

因此，该方案主要作为备用手段。

4.2. 利用第三方应用广播拉活

4.2.1. 方案设计思想

该方案总的设计思想与接收系统广播类似，不同的是该方案为接收第三方 Top 应用广播。

通过反编译第三方 Top 应用，如：手机 QQ、微信、支付宝、UC 浏览器等，以及友盟、信鸽、个推等 SDK，找出它们外发的广播，在应用中进行监听，这样当这些应用发出广播时，就会将我们的应用拉活。

4.2.2. 方案适用范围

该方案的有效程度除与系统广播一样的因素外，主要受如下因素限制：

- 1) 反编译分析过的第三方应用的多少
- 2) 第三方应用的广播属于应用私有，当前版本中有效的广播，在后续版本随时就可能被移除或被改为不外发。

这些因素都影响了拉活的效果。

4.3. 利用系统 Service 机制拉活

4.3.1. 方案设计思想

将 Service 设置为 START_STICKY，利用系统机制在 Service 挂掉后自动拉活：



4.3.2. 方案适用范围

如下两种情况无法拉活：

- 1、Service 第一次被异常杀死后会在 5 秒内重启，第二次被杀死会在 10 秒内重启，第三次会在 20 秒内重启，一旦在短时间内 Service 被杀死达到 5 次，则系统不再拉起。
- 2、进程被取得 Root 权限的管理工具或系统工具通过 forestop 停止掉，无法重启。

4.4. 利用 Native 进程拉活

4.4.1. 方案设计思想

主要思想：利用 Linux 中的 fork 机制创建 Native 进程，在 Native 进程中监控主进程的存活，当主进程挂掉后，在 Native 进程中立即对主进程进行拉活。

主要原理：在 Android 中所有进程和系统组件的生命周期受 ActivityManagerService 的统一管理。而且，通过 Linux 的 fork 机制创建的进程为纯 Linux 进程，其生命周期不受 Android 的管理。

4.4.2. 方案实现挑战

挑战一：在 Native 进程中如何感知主进程死亡。

要在 Native 进程中感知主进程是否存活有两种实现方式：

在 Native 进程中通过死循环或定时器，轮训判断主进程是否存活，当主进程不存活时进行拉活。该方案的很大缺点是不停的轮询执行判断逻辑，非常耗电。

在主进程中创建一个监控文件，并且在主进程中持有文件锁。在拉活进程启动后申请文件锁将会被堵塞，一旦可以成功获取到锁，说明主进程挂掉，即可进行拉活。由于 Android 中的应用都运行于虚拟机之上，Java 层的文件锁与 Linux 层的文件锁是不同的，要实现该功能需要封装 Linux 层的文件锁供上层调用。

封装 Linux 文件锁的代码如下：



```
64 * 给文件加锁，成功返回0，失败返回1
65 */
66 int linuxflock(JNIEnv *env, jobject obj, jstring lockFilePath)
67 {
68     LOGI("flock");
69     int fd = linuxgetlockfd(env, lockFilePath);
70     LOGD("Lock file fd : %d", fd);
71     if(fd > 0)
72     {
73
74         if(linuxtestflock(fd) == 0)
75         {
76             LOGD("Try to lock the file");
77             return flock(fd, LOCK_EX); //创建文件排它锁
78         }
79     }
80     return 1;
81 }
```

Native 层中堵塞申请文件锁的部分代码：

```
*flock.cpp  watchdog.c  KeepLiveActivity.java  KeepLiveService.java  *XLBDae

85 void *thread_watch(void* arg)
86 {
87     LOGI("jni thread_watch");
88
89     int fd = 0;
90     if ((fd = open(mLockFilePath, O_RDWR)) == -1)
91     {
92         LOGE("jni thread_watch open file failed, errno : %d", errno);
93         pthread_exit(0);
94     }
95
96     LOGD("jni thread_watch, open file success , try to lock the file");
97
98     set_watch_state_symbol(1);
99     if(flock(fd, LOCK_EX) == 0)//申请文件的排它锁
100     {
101         LOGD("jni thread_watch successfully get file lock");
102         try_pullup();
103     }
104     else
105     {
106         set_watch_state_symbol(0);
107         LOGD("jni thread_watch, lock file error !!!");
108     }
109     close(fd);//关闭文件会将锁自动释放
110     return 0;
```

挑战二：在 Native 进程中如何拉活主进程。

通过 Native 进程拉活主进程的部分代码如下，即通过 am 命令进行拉活。通过指定“--include-stopped-packages”参数来拉活主进程处于 forestop 状态的情况。

```
ck.cpp  watchdog.c  KeepLiveActivity.java  KeepLiveService.java  *XLBDaemon Manifest  styles.xml  KeepLiveManager

static void try_pullup()
{
    LOGI("jni try_pullup");

    if(!is_application_exist(mPackageName))
    {
        LOGD("jni try_pullup process exit, because xlb has been removed !");
        set_watch_state_symbol(0);
        freeAppMsg();
        exit(0);
    }

    LOGD("jni try_pullup try to wake up live service ...");

    char pkg[256] = "";
    strcat(pkg, "-n ");
    strcat(pkg, mPackageName);
    strcat(pkg, "/com.xlb.keeplive.KeepLiveService");

    int ret = execlp("am", "am", "startservice", "--user", mSerial, "-n", pkg, "--include-stopped-packages", NULL);
    LOGD("start service, ret = %d", ret);

    freeAppMsg();
    set_watch_state_symbol(0);
    pthread_exit(0);
}
```

挑战三：如何保证 Native 进程的唯一。

从可扩展性和进程唯一等多方面考虑，将 Native 进程设计层 C/S 结构模式，主进程与 Native 进程通过 Localsocket 进行通信。在 Native 进程中利用 Localsocket 保证 Native 进程的唯一性，不至于出现创建多个 Native 进程以及 Native 进程变成僵尸进程等问题。



```
int main(const int argc, const char *argv[]) {
    LOGI("jni do_main");

    if(argc < 2)
    {
        return -1;
    }

    struct sockaddr addr;
    socklen_t alen;
    int lsocket, s, count;

    const char* socket_name = argv[1];
    LOGD("jni socket name : %s", socket_name);

    lsocket = socket_local_server(socket_name, ANDROID_SOCKET_NAMESPACE_ABSTRACT, SOCK_STREAM);

    if (lsocket < 0) {
        LOGE("jni Failed to get socket from environment: %s", strerror(errno));
        exit(1);
    }

    fcntl(lsocket, F_SETFD, FD_CLOEXEC);

    LOGD("jni native pid = %d", getpid());
    char cmdBuf[BUFFER_MAX];
    MEM_ZERO(cmdBuf, BUFFER_MAX);
}
```

4.4.3. 方案适用范围

该方案主要适用于 Android5.0 以下版本手机。

该方案不受 forcestop 影响，被强制停止的应用依然可以被拉活，在 Android5.0 以下版本拉活效果非常好。

对于 Android5.0 以上手机，系统虽然会将 native 进程内的所有进程都杀死，这里其实就是系统“依次”杀死进程时间与拉活逻辑执行时间赛跑的问题，如果可以跑的比系统逻辑快，依然可以有效拉起。记得网上有人做过实验，该结论是成立的，在某些 Android 5.0 以上机型有效。

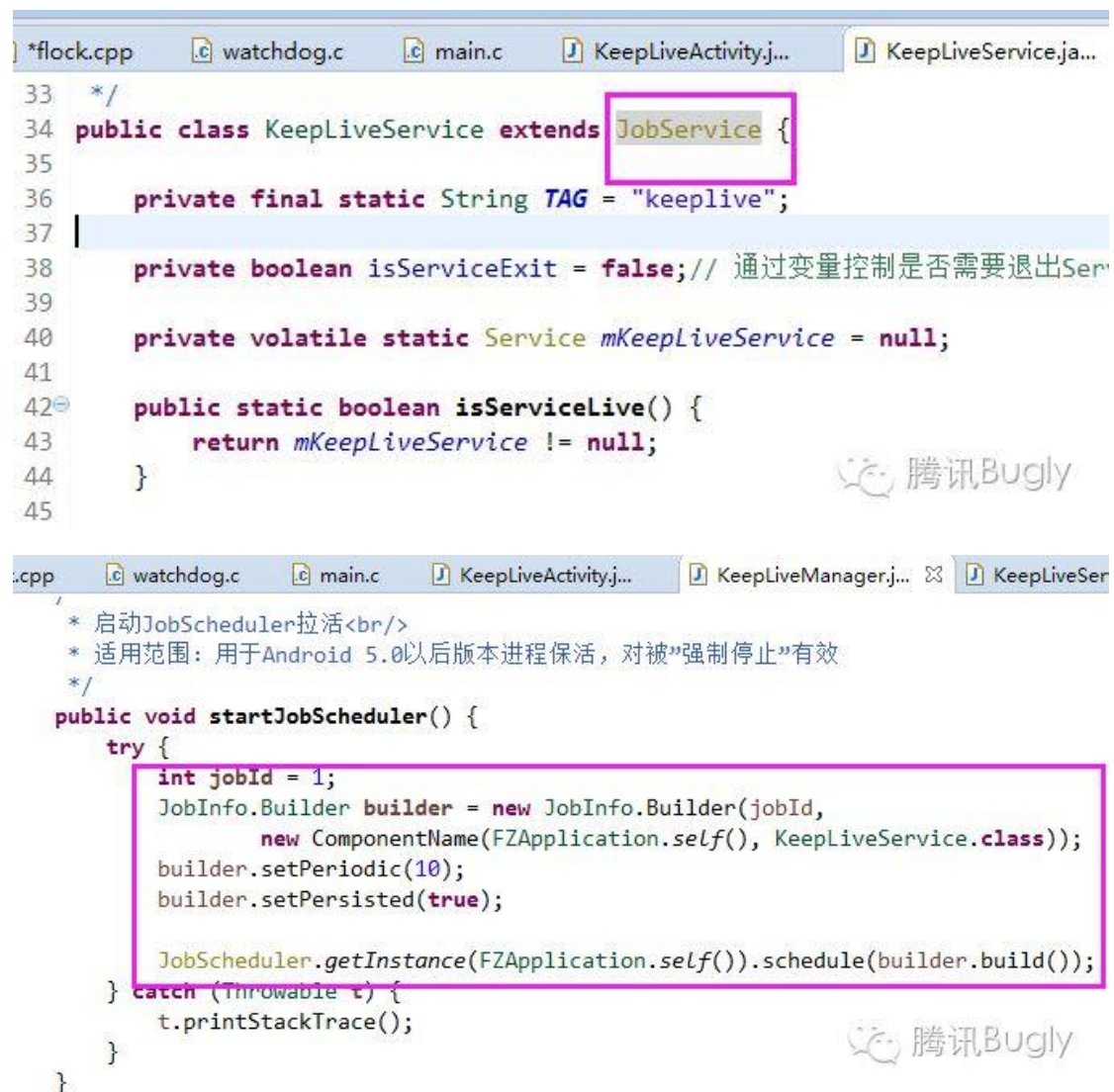
4.5. 利用 JobScheduler 机制拉活

4.5.1. 方案设计思想

Android5.0 以后系统对 Native 进程等加强了管理，Native 拉活方式失效。系统在 Android5.0 以上版本提供了 JobScheduler 接口，系统会定时调用该进程以使应用进行一些逻辑操作。

在本项目中，我对 JobScheduler 进行了进一步封装，兼容 Android5.0 以下版本。封装后

JobScheduler 接口的使用如下：



```
33  */
34  public class KeepLiveService extends JobService {
35
36      private final static String TAG = "keeplive";
37
38      private boolean isServiceExit = false; // 通过变量控制是否需要退出Ser
39
40      private volatile static Service mKeepLiveService = null;
41
42      public static boolean isServiceLive() {
43          return mKeepLiveService != null;
44      }
45
...
* 启动JobScheduler拉活<br/>
* 适用范围: 用于Android 5.0以后版本进程保活, 对被"强制停止"有效
*/
public void startJobScheduler() {
    try {
        int jobId = 1;
        JobInfo.Builder builder = new JobInfo.Builder(jobId,
            new ComponentName(FZApplication.self(), KeepLiveService.class));
        builder.setPeriodic(10);
        builder.setPersisted(true);
        JobScheduler.getInstance(FZApplication.self()).schedule(builder.build());
    } catch (Throwable t) {
        t.printStackTrace();
    }
}
```

4.5.2. 方案适用范围

该方案主要适用于 Android5.0 以上版本手机。

该方案在 Android5.0 以上版本中不受 forcestop 影响, 被强制停止的应用依然可以被拉活, 在 Android5.0 以上版本拉活效果非常好。

仅在小米手机可能会出现有时无法拉活的问题。

4.6. 利用账号同步机制拉活

4.6.1. 方案设计思想

Android 系统的账号同步机制会定期同步账号进行, 该方案目的在于利用同步机制进行进程的拉活。添加账号和设置同步周期的代码如下：


```

KeepLiveService.java
135  /**
136   * 添加账户，并启用账户同步功能<br/>
137   * 利用账户同步功能进行拉活，适用范围为目前所有版本<br/>
138   * 这里设置的账户同步周期为30秒，可以通过常量SYNC_FREQUENCY修改
139   */
140  private void addAccount() {
141      AccountManager accountManager = (AccountManager) this.getSystemService(Context.ACCOUNT_SERVICE);
142      Account account = null;
143      Account[] accounts = accountManager.getAccountsByType(ACCOUNT_TYPE);
144      if (accounts.length > 0) {
145          account = accounts[0];
146      } else {
147          account = new Account(getString(R.string.account_name), ACCOUNT_TYPE);
148      }
149
150      if (accountManager.addAccountExplicitly(account, null, null)) {
151          //开启同步，并设置同步周期
152          ContentResolver.setIsSyncable(account, CONTENT_AUTHORITY, 1);
153          ContentResolver.setSyncAutomatically(account, CONTENT_AUTHORITY, true);
154          ContentResolver.addPeriodicSync(account, CONTENT_AUTHORITY, new Bundle(), SYNC_FREQUENCY);
155      }
156  }

```

该方案需要在 AndroidManifest 中定义账号授权与同步服务。

```

vice.java  XLBDaemon Manifest
<android:stopWithTask="false" />
<service
    android:name="com.xlb.keplive.KeepLiveService$AuthenticationService"
    android:exported="true"
    android:process=":live" >
    <intent-filter>
        <action android:name="android.accounts.AccountAuthenticator" />
    </intent-filter>

    <meta-data
        android:name="android.accounts.AccountAuthenticator"
        android:resource="@xml/authenticator" />
</service>
<service
    android:name="com.xlb.keplive.KeepLiveService$AccountSyncService"
    android:exported="true"
    android:process=":live" >
    <intent-filter>
        <action android:name="android.content.SyncAdapter" />
    </intent-filter>

    <meta-data
        android:name="android.content.SyncAdapter"
        android:resource="@xml/syncadapter" />
</service>

```

4.6.2. 方案适用范围

该方案适用于所有的 Android 版本，包括被 forestop 掉的进程也可以进行拉活。

最新 Android 版本（Android N）中系统好像对账户同步这里做了变动，该方法不再有效。

5. 其他有效拉活方案

经研究发现还有其他一些系统拉活措施可以使用，但在使用时需要用户授权，用户感知比较强烈。

这些方案包括：

1、利用系统通知管理权限进行拉活

2、利用辅助功能拉活，将应用加入厂商或管理软件白名单。

这些方案需要结合具体产品特性来搞。

上面所有解释这些方案都是考虑的不 Root 的情况。

其他还有一些技术之外的措施，比如说应用内 Push 通道的选择：

1、国外版应用：接入 Google 的 GCM。

2、国内版应用：根据终端不同，在小米手机（包括 MIUI）接入小米推送、华为手机接入华为推送；其他手机可以考虑接入腾讯信鸽或极光推送与小米推送做 A/B Test。