Android 详解 App 保活实现原理

deepwaterooo

August 30, 2022

Contents

1 Android 详解 App 保活实现理原

1.1 概述

- 早期的 Android 系统不完善，导致 App 侧有很多空子可以钻，因此它们有着有着各种各样的姿势进行保活。譬如说在 Android 5.0 以前，App 内部通过 native 方式 fork 出来的进程是不受系统管控的，系统在杀 App 进程的时候，只会去杀 App 启动的 Java 进程；因此诞生了一大批“毒瘤”，他们通过 fork native 进程，在 App 的 Java 进程被杀死的时候通过 am 命令拉起自己从而实现永生。那时候的 Android 可谓是魑魅横行，群魔乱舞；系统根本管不住应用，因此长期以来被人诟病耗电、卡顿。同时，系统的软弱导致了 Xposed 框架、阻止运行、绿色守护、黑域、冰箱等一系列管制系统后台进程的框架和 App 出现。

- 不过，随着 Android 系统的发展，这一切都在往好的方向演变。

- Android 5.0 以上，系统杀进程以 uid 为标识，通过杀死整个进程组来杀进程，因此 native 进程也躲不过系统的法眼。

- Android 6.0 引入了待机模式 (doze)，一旦用户拔下设备的电源插头，并在屏幕关闭后的一段时间内使其保持不活动状态，设备会进入低电耗模式，在该模式下设备会尝试让系统保持休眠状态。

- Android 7.0 加强了之前鸡肋的待机模式（不再要求设备静止状态），同时对开启了 Project Svelte，Project Svelte 是专门用来优化 Android 系统后台的项目，在 Android 7.0 上直接移除了一些隐式广播，App 无法再通过监听这些广播拉起自己。

- Android 8.0 进一步加强了应用后台执行限制：一旦应用进入已缓存状态时，如果没有活动的组件，系统将解除应用具有的所有唤醒锁。另外，系统会限制未在前台运行的应用的某些行为，比如说应用的后台服务的访问受到限制，也无法使用 Mainifest 注册大部分隐式广播。

- Android 9.0 进一步改进了省电模式的功能并加入了应用待机分组，长时间不用的 App 会被打入冷宫；另外，系统监测到应用消耗过多资源时，系统会通知并询问用户是否需要限制该应用的后台活动。
- 然而，道高一尺，魔高一丈。系统在不断演进，保活方法也在不断发展。大约在 4 年前出现过一个 MarsDaemon，这个库通过双进程守护的方式实现保活，一时间风头无两。不过好景不长，进入 Android 8.0 时代之后，这个库就逐渐消亡。
- 一般来说，Android 进程保活分为两个方面：
  - 保持进程不被系统杀死。
  - 进程被系统杀死之后，可以重新复活。
- 随着 Android 系统变得越来越完善，单单通过自己拉活自己逐渐变得不可能了；因此后面的所谓「保活」基本上是两条路：
  - 1. 提升自己进程的优先级，让系统不要轻易弄死自己；
  - 2. App 之间互相结盟，一个兄弟死了其他兄弟把它拉起来。
- 当然，还有一种终极方法，那就是跟各大系统厂商建立 PY 关系，把自己加入系统内存清理的白名单；比如说国民应用微信。当然这条路一般人是没有资格走的。
- 大约一年以前，大神 gityuan 在其博客上公布了 TIM 使用的一种可以称之为「终极永生术」的保活方法；这种方法在当前 Android 内核的实现上可以大大提升进程的存活率。笔者研究了这种保活思路的实现原理，并且提供了一个参考实现 Leoric。接下来就给大家分享一下这个终极保活黑科技的实现原理。

1.2 保活的底层技术原理

- 知己知彼，百战不殆。既然我们想要保活，那么首先得知道我们是怎么死的。一般来说，系统杀进程有两种方法，这两个方法都通过 ActivityManagerService 提供：
  - 1.killBackgroundProcesses
  - 2.forceStopPackage
- 在原生系统上，很多时候杀进程是通过第一种方式，除非用户主动在 App 的设置界面点击「强制停止」。不过国内各厂商以及一加三星等 ROM 现在一般使用第二种方法。第一种方法太过温柔，根本治不住想要搞事情的应用。第二种方法就比较强力了，一般来说被 force-stop 之后，App 就只能乖乖等死了。
- 因此，要实现保活，我们就得知道 force-stop 到底是如何运作的。既然如此，我们就跟踪一下系统的 **forceStopPackage** 这个方法的执行流程：
- 首先是 ActivityManagerService 里面的 forceStopPackage 这方法：

```
public void forceStopPackage(final String packageName, int userId) {

    // .. 权限检查，省略

    long callingId = Binder.clearCallingIdentity();
    try {
        IPackageManager pm = AppGlobals.getPackageManager();
        synchronized(this) {
            int[] users = userId == UserHandle.USER_ALL
                    ? mUserController.getUsers() : new int[] { userId };
            for (int user : users) {

                // 状态判断，省略..

                int pkgUid = -1;
                try {
                    pkgUid = pm.getPackageUid(packageName, MATCH_DEBUG_TRIAGED_MISSING,
```

```
                        user);
                } catch (RemoteException e) {
                }
                if (pkgUid == -1) {
                    Slog.w(TAG, "Invalid packageName: " + packageName);
                    continue;
                }
                try {
                    pm.setPackageStoppedState(packageName, true, user);
                } catch (RemoteException e) {
                } catch (IllegalArgumentException e) {
                    Slog.w(TAG, "Failed trying to unstop package "
                            + packageName + ": " + e);
                }
                if (mUserController.isUserRunning(user, 0)) {
                    // 根据 UID 和包名杀进程
                    forceStopPackageLocked(packageName, pkgUid, "from pid " + callingPid);
                    finishForceStopPackageLocked(packageName, pkgUid);
                }
            }
        }
    } finally {
        Binder.restoreCallingIdentity(callingId);
    }
}
```

- 在这里我们可以知道，系统是通过 uid 为单位 force-stop 进程的，因此不论你是 native 进程还是 Java 进程，force-stop 都会将你统统杀死。我们继续跟踪 forceStopPackageLocked 这个方法：

```
final boolean forceStopPackageLocked(String packageName, int appId,
        boolean callerWillRestart, boolean purgeCache, boolean doit,
        boolean evenPersistent, boolean uninstalling, int userId, String reason) {
    int i;

    // .. 状态判断，省略

    boolean didSomething = mProcessList.killPackageProcessesLocked(packageName, appId, userId,
            ProcessList.INVALID_ADJ, callerWillRestart, true /* allowRestart */, doit,
            evenPersistent, true /* setRemoved */,
            packageName == null ? ("stop user " + userId) : ("stop " + packageName));

    didSomething |=
            mAtmInternal.onForceStopPackage(packageName, doit, evenPersistent, userId);

    // 清理 service
    // 清理 broadcastreceiver
    // 清理 providers
    // 清理其他

    return didSomething;
}
```

- 这个方法实现很清晰：先杀死这个 App 内部的所有进程，然后清理残留在 system_server 内的四大组件信息；我们关心进程是如何被杀死的，因此继续跟踪 killPackageProcessesLocked，这个方法最终会调用到 ProcessList 内部的 removeProcessLocked 方法，removeProcessLocked 会调用 ProcessRecord 的 kill 方法，我们看看这个 kill：

```
void kill(String reason, boolean noisy) {
    if (!killedByAm) {
        Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "kill");
        if (mService != null && (noisy || info.uid == mService.mCurOomAdjUid)) {
            mService.reportUidInfoMessageLocked(TAG,
                    "Killing " + toShortString() + " (adj " + setAdj + "): " + reason,
                    info.uid);
        }
        if (pid > 0) {
            EventLog.writeEvent(EventLogTags.AM_KILL, userId, pid, processName, setAdj, reason);
            Process.killProcessQuiet(pid);
            ProcessList.killProcessGroup(uid, pid);
        } else {
            pendingStart = false;
        }
        if (!mPersistent) {
            killed = true;
```

```
            killedByAm = true;
        }
        Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
    }
}
```

- 这里我们可以看到，首先杀掉了目标进程，然后会以 uid 为单位杀掉目标进程组。如果只杀掉目标进程，那么我们可以通过双进程守护的方式实现保活；关键就在于这个 killProcessGroup，继续跟踪之后发现这是一个 native 方法，它的最终实现在 libprocessgroup 中，代码如下：

```
int killProcessGroup(uid_t uid, int initialPid, int signal) {
    return KillProcessGroup(uid, initialPid, signal, 40 /*retries*/);
}
```

- 注意这里有个奇怪的数字：40。我们继续跟踪：

```
static int KillProcessGroup(uid_t uid, int initialPid, int signal, int retries) {

    // 省略

    int retry = retries;
    int processes;
    while ((processes = DoKillProcessGroupOnce(cgroup, uid, initialPid, signal)) > 0) {
        LOG(VERBOSE) << "Killed " << processes << " processes for processgroup " << initialPid;
        if (retry > 0) {
            std::this_thread::sleep_for(5ms);
            --retry;
        } else {
            break;
        }
    }

    // 省略
}
```

- 瞧瞧我们的系统做了什么骚操作？循环 40 遍不停滴杀进程，每次杀完之后等 5ms，循环完毕之后就算过去了。

- 看到这段代码，我想任何人都会蹦出一个疑问：假设经历连续 40 次的杀进程之后，如果 App 还有进程存在，那不就侥幸逃脱了吗？

1.3 实现方法

- 那么，如何实现这个目的呢？我们看这个关键的 5ms。假设，App 进程在被杀掉之后，能够以足够快的速度（5ms 内）启动一堆新的进程，那么系统在一次循环杀掉老的所有进程之后，sleep 5ms 之后又会遇到一堆新的进程；如此循环 40 次，只要我们每次都能够拉起新的进程，那我们的 App 就能逃过系统的追杀，实现永生。是的，炼狱般的 200ms，只要我们熬过 200ms 就能渡劫成功，得道飞升。不知道大家有没有玩过打地鼠这个游戏，整个过程非常类似，按下去一个又冒出一个，只要每次都能足够快地冒出来，我们就赢了。

- 现在问题的关键就在于：如何在 5ms 内启动一堆新的进程？

- 再回过头来看原来的保活方式，它们拉起进程最开始通过 am 命令，这个命令实际上是一个 java 程序，它会经历启动一个进程然后启动一个 ART 虚拟机，接着获取 ams 的 binder 代理，然后与 ams 进行 binder 同步通信。这个过程实在是太慢了，在这与死神赛跑的 5ms 里，它的速度的确是不敢恭维。

- 自媒体培训

- 后来，MarsDaemon 提出了一种新的方式，它用 binder 引用直接给 ams 发送 Parcel，这个过程相比 am 命令快了很多，从而大大提高了成功率。其实这里还有改进的空间，毕竟这里还是在 Java 层调用，Java 语言在这种实时性要求极高的场合有一个非常令人诟病的特性：垃圾回收（GC）；虽然我们在这 5ms 内直接碰上 gc 引发停顿的可能性非常小，但是由于 GC 的存在，ART 中的 Java 代码存在非常多的 checkpoint；想象一下你现在是一个信使有重要军情

要报告，但是在路上却碰到很多关隘，而且很可能被勒令暂时停止一下，这种情况是不可接受的。因此，最好的方法是通过 native code 给 ams 发送 binder 调用；当然，如果再底层一点，我们甚至可以通过 ioctl 直接给 binder 驱动发送数据进而完成调用，但是这种方法的兼容性比较差，没有用 native 方式省心。

- 通过在 native 层给 ams 发送 binder 消息拉起进程，我们算是解决了「快速拉起进程」这个问题。但是这个还是不够。还是回到打地鼠这个游戏，假设你摁下一个地鼠，会冒起一个新的地鼠，那么你每次都能摁下去最后获取胜利的概率还是比较高的；但如果你每次摁下一个地鼠，其他所有地鼠都能冒出来呢？这个难度系数可是要高多了。如果我们的进程能够在任意一个进程死亡之后，都能让把其他所有进程全部拉起，这样系统就很难杀死我们了。
- 新的黑科技保活中通过 2 个机制来保证进程之间的互相拉起：
  - 1. 2 个进程通过互相监听文件锁的方式，来感知彼此的死亡。
  - 2. 通过 fork 产生子进程，fork 的进程同属一个进程组，一个被杀之后会触发另外一个进程被杀，从而被文件锁感知。
- 具体来说，创建 2 个进程 p1, p2，这两个进程通过文件锁互相关联，一个被杀之后拉起另外一个；同时 p1 经过 2 次 fork 产生孤儿进程 c1，p2 经过 2 次 fork 产生孤儿进程 c2，c1 和 c2 之间建立文件锁关联。这样假设 p1 被杀，那么 p2 会立马感知到，然后 p1 和 c1 同属一个进程组，p1 被杀会触发 c1 被杀，c1 死后 c2 立马感受到从而拉起 p1，因此这四个进程三三之间形成了铁三角，从而保证了存活率。
- 分析到这里，这种方案的大致原理我们已经清晰了。基于以上原理，我写了一个简单的 PoC，代码在这里：https://github.com/tiann/Leoric 有兴趣的可以看一下。
- 上面别人分析和实现的案例已经 fork 到我的仓库，方便学习
- **我想拿这个案例来学习，把里面的每个细节都弄懂了，我的知识结构上下层就可以串起来了，并且可以帮助我更好地理解 IBinder IPC 底层原理**

1.4 改进空间

- 本方案的原理还是比较简单直观的，但是要实现稳定的保活，还需要很多细节要补充；特别是那与死神赛跑的 5ms，需要不计一切代价去优化才能提升成功率。具体来说，就是当前的实现是在 Java 层用 binder 调用的，我们应该在 native 层完成。笔者曾经实现过这个方案，但是这个库本质上是有损用户利益的，因此并不打算公开代码，这里简单提一下实现思路供大家学习：

1.4.1 如何在 native 层进行 binder 通信

- libbinder 是 NDK 公开库，拿到对应头文件，动态链接即可。
- 难点：依赖繁多，剥离头文件是个体力活。

1.4.2 如何组织 binder 通信的数据？

- 通信的数据其实就是二进制流；具体表现就是 (C++/Java) Parcel 对象。native 层没有对应的 Intent Parcel，兼容性差。
- 方案：
  - 1.Java 层创建 Parcel（含 Intent），拿到 Parcel 对象的 mNativePtr(native peer)，传到 Native 层。
  - 2.native 层直接把 mNativePtr 强转为结构体指针。
  - 3.fork 子进程，建立管道，准备传输 parcel 数据。
  - 4. 子进程读管道，拿到二进制流，重组为 parcel。

1.5 如何应对

- 今天我把这个实现原理公开，并且提供 PoC 代码，并不是鼓励大家使用这种方式保活，而是希望各大系统厂商能感知到这种黑科技的存在，推动自己的系统彻底解决这个问题。
- 两年前我就知道了这个方案的存在，不过当时鲜为人知。最近一个月我发现很多 App 都使用了这种方案，把我的 Android 手机折腾的惨不忍睹；毕竟本人手机上安装了将近 800 个 App，假设每个 App 都用这个方案保活，那这系统就没法用了。

1.5.1 系统如何应对

- 如果我们把系统杀进程比喻为斩首，那么这个保活方案的精髓在于能快速长出一个新的头；因此应对之法也很简单，只要我们在斩杀一个进程的时候，让别的进程老老实实呆着别搞事情就 OK 了。具体的实现方法多种多样，不赘述。

1.5.2 用户如何应对

- 在厂商没有推出解决方案之前，用户可以有一些方案来缓解使用这个方案进行保活的流氓 App。这里推荐两个应用给大家：
  - 冰箱
  - Island
- 通过冰箱的冻结和 Island 的深度休眠可以彻底阻止 App 的这种保活行为。当然，如果你喜欢别的这种“冻结”类型的应用，比如小黑屋或者太极的阴阳之门也是可以的。
- 其他不是通过“冻结”这种机制来压制后台的应用理论上对这种保活方案的作用非常有限。

1.6 总结

- 1. 对技术来说，黑科技没有什么黑的，不过是对系统底层原理的深入了解从而反过来对抗系统的一种手段。很多人会说，了解系统底层有什么用，本文应该可以给出一个答案：可以实现别人永远也无法实现的功能，通过技术推动产品，从而产生巨大的商业价值。
- 2. 黑科技虽强，但是它不该存在于这世上。没有规矩，不成方圆。黑科技黑的了一时，黑不了一世。要提升产品的存活率，终归要落到产品本身上面来，尊重用户，提升体验方是正途。
- 以上就是详解 App 保活实现原理的详细内容，更多关于 App 保活实现原理的资料请关注脚本之家其它相关文章！https://www.jb51.net/article/214545.htm