Android ActivityManagerService 在系统低内存时对应用生死的控制

deepwaterooo

September 13, 2022

Contents

1 ActivityManagerService 恢复 App 现场机制

- 改天早上把这个看一遍https://blog.csdn.net/weixin_34090562/article/details/93166086?spm=1001.2101.3001.6650.2&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-2-93166086-blog-76046447.pc_relevant_aa&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-2-93166pc_relevant_aa&utm_relevant_index=3 感觉是个透透的高手写得，逻辑清晰透彻
- 假设，一个应用被后台杀死，再次从最近的任务列表唤起 App 时候，系统是如何处理的呢?

1.1 Android 框架层（AMS）如何知道 App 被杀死了

- 首先来看第一个问题，系统如何知道 Application 被杀死了，Android 使用了 Linux 的 oomKiller 机制，只是简单的做了个变种，采用分等级的 LowmemoryKiller，但这个其实是内核层面，LowmemoryKiller 杀死进程后，如何向用户空间发送通知，并告诉框架层的 ActivityMangerService 呢？只有 AMS 在知道 App 或者 Activity 是否被杀死后，AMS（ActivityMangerService）才能正确的走重建或者唤起流程，比如，APP 死了，但是由于存在需要复活的 Service，那么这个时候，进程需要重新启动，这个时候怎么处理的，那么 AMS 究竟是在什么时候知道 App 或者 Activity 被后台杀死了呢？我们先看一下从最近的任务列表进行唤起的时候，究竟发生了什么。

1.1.1 从最近的任务列表或者 Icon 再次唤起 App 流程

- 在系统源码 systemUi 的包里，有个 RecentActivity，这个其实就是最近的任务列表的入口，而其呈现界面是通过 RecentsPanelView 来展现的，点击最近的 App 其执行代码如下：

```
public void handleOnClick(View view) {
    ViewHolder holder = (ViewHolder)view.getTag();
    TaskDescription ad = holder.taskDescription;
    final Context context = view.getContext();
    final ActivityManager am = (ActivityManager)
        context.getSystemService(Context.ACTIVITY_SERVICE);
    Bitmap bm = holder.thumbnailViewImageBitmap;
    // 关键点 1  如果 TaskDescription 没有被主动关闭，正常关闭，ad.taskId 就是 >=0
    if (ad.taskId >= 0) {
        // This is an active task; it should just go to the foreground.
        am.moveTaskToFront(ad.taskId, ActivityManager.MOVE_TASK_WITH_HOME, opts);
    } else {
        Intent intent = ad.intent;
        intent.addFlags(Intent.FLAG_ACTIVITY_LAUNCHED_FROM_HISTORY
                        | Intent.FLAG_ACTIVITY_TASK_ON_HOME
                        | Intent.FLAG_ACTIVITY_NEW_TASK);
        try {
            context.startActivityAsUser(
                intent, opts,
                new UserHandle(UserHandle.USER_CURRENT));
        }
    }
}
```

- 在上面的代码里面，**有个判断 ad.taskId >= 0，如果满足这个条件，就通过 moveTaskToFront 唤起 APP** ,那么 ad.taskId 是如何获取的?recent 包里面有各类 RecentTasksLoader，这个类就是用来加载最近任务列表的一个 Loader，看一下它的源码，主要看一下加载：

```
@Override
protected Void doInBackground(Void params) {
    // We load in two stages: first, we update progress with just the first screenful
    // of items. Then, we update with the rest of the items
    final int origPri = Process.getThreadPriority(Process.myTid());
    Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
    final PackageManager pm = mContext.getPackageManager();
    final ActivityManager am = (ActivityManager)
        mContext.getSystemService(Context.ACTIVITY_SERVICE);

    final List<ActivityManager.RecentTaskInfo> recentTasks =
        am.getRecentTasks(MAX_TASKS, ActivityManager.RECENT_IGNORE_UNAVAILABLE);

    TaskDescription item = createTaskDescription(recentInfo.id,
                                                 recentInfo.persistentId, recentInfo.baseIntent,
                                                 recentInfo.origActivity, recentInfo.description);
}
```

- 可以看到，其实就是通过 ActivityManger 的 getRecentTasks 向 AMS 请求最近的任务信息，然后通过 createTaskDescription 创建 TaskDescription，这里传递的 recentInfo.id 其实就是 TaskDescription 的 taskId，来看一下它的意义：

```
public List<ActivityManager.RecentTaskInfo> getRecentTasks(int maxNum, int flags, int userId) {
    IPackageManager pm = AppGlobals.getPackageManager();

    final int N = mRecentTasks.size();
    for (int i=0; i<N && maxNum > 0; i++) {
        TaskRecord tr = mRecentTasks.get(i);
        if (i == 0
            || ((flags&ActivityManager.RECENT_WITH_EXCLUDED) != 0)
            || (tr.intent == null)
            || ((tr.intent.getFlags() & Intent.FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS) == 0)) {
            ActivityManager.RecentTaskInfo rti = new ActivityManager.RecentTaskInfo();
            rti.id = tr.numActivities > 0 ? tr.taskId : -1;
            rti.persistentId = tr.taskId;
            rti.baseIntent = new Intent(
                tr.intent != null ? tr.intent : tr.affinityIntent);
            if (!detailed)
                rti.baseIntent.replaceExtras((Bundle)null);
```

- 可以看出 RecentTaskInfo 的 id 是由 TaskRecord 决定的，如果 TaskRecord 中 numActivities > 0 就去 TaskRecord 的 Id，否则就取-1，这里的 numActivities 其实就是 TaskRecode 中记

录的 ActivityRecord 的数目，更具体的细节可以自行查看 ActivityManagerService 及 ActivityStack，那么这里就容易解释了，只要是存活的 APP，或者被 LowmemoryKiller 杀死的 APP，其 AMS 的 ActivityRecord 是完整保存的，这也是恢复的依据。对于 RecentActivity 获取的数据其实就是 AMS 中的翻版，它也是不知道将要唤起的 APP 是否是存活的，只要 TaskRecord 告诉 RecentActivity 是存货的，那么久直接走唤起流程。也就是通过 ActivityManager 的 moveTaskToFront 唤起 App，至于后续的工作，就完全交给 AMS 来处理。现看一下到这里的流程图：

1.2 整个 APP 被后台杀死的情况下 AMS 是如何恢复现场的

- AMS 与客户端的通信是通过 Binder 来进行的，并且通信是”全双工“的，且互为客户端跟服务器，也就说 AMS 向客户端发命令的时候，AMS 是客户端，反之亦然。注意 Binder 有个讣告的功能的：如果基于 Binder 通信的服务端（S）如果挂掉了，客户端（C）能够收到 Binder 驱动发送的一份讣告，告知客户端 Binder 服务挂了，可以把 Binder 驱动看作是第三方不死邮政机构，专门向客户端发偶像死亡通知。对于 APP 被异常杀死的情况下，这份讣告是发送给 AMS 的，AMS 在收到通知后，就会针对 APP 被异常杀死的情况作出整理，这里牵扯到 Binder 驱动的代码有兴趣可以自己翻一下。之类直接冲讣告接受后端处理逻辑来分析，在 AMS 源码中，入口其实就是 appDiedLocked.

```
final void appDiedLocked(ProcessRecord app, int pid,
                         IApplicationThread thread) {
    if (app.pid == pid && app.thread != null &&
        app.thread.asBinder() == thread.asBinder()) {
        boolean doLowMem = app.instrumentationClass == null;
        // 关键点 1
        handleAppDiedLocked(app, false, true);
        // 如果是被后台杀了，怎么处理
        // 关键点 2
        if (doLowMem) {
            boolean haveBg = false;
            for (int i=mLruProcesses.size()-1; i>=0; i--) {
                ProcessRecord rec = mLruProcesses.get(i);
                if (rec.thread != null && rec.setAdj >= ProcessList.HIDDEN_APP_MIN_ADJ) {
                    haveBg = true;
                    break;
                }
            }
            if (!haveBg) {
                // <!--如果被 LowmemoryKiller 杀了，就说明内存紧张，这个时候就会通知其他后台 APP，小心了，赶紧释放资源-->
                EventLog.writeEvent(EventLogTags.AM_LOW_MEMORY, mLruProcesses.size());
                long now = SystemClock.uptimeMillis();
                for (int i=mLruProcesses.size()-1; i>=0; i--) {
                    ProcessRecord rec = mLruProcesses.get(i);
                    if (rec != app && rec.thread != null && (rec.lastLowMemory+GC_MIN_INTERVAL) <= now) {
                        if (rec.setAdj <= ProcessList.HEAVY_WEIGHT_APP_ADJ) {
                            rec.lastRequestedGc = 0;
                        } else {
                            rec.lastRequestedGc = rec.lastLowMemory;
                        }
                        rec.reportLowMemory = true;
                        rec.lastLowMemory = now;
                        mProcessesToGc.remove(rec);
                        addProcessToGcListLocked(rec);
                    }
                }
                mHandler.sendEmptyMessage(REPORT_MEM_USAGE);
                // <!--缩减资源-->
                scheduleAppGcsLocked();
            }
        }
    }
}
```

- 先看关键点 1：在进程被杀死后，AMS 端要选择性清理进程相关信息，清理后，再根据是不是内存低引起的后台杀死，决定是不是需要清理其他后台进程。接着看 handleAppDiedLocked 如何清理的，这里有重建时的依据：ActivityRecord 不清理，但是为它设置个 APP 未绑定的标识

```
private final void handleAppDiedLocked(ProcessRecord app,
                                       boolean restarting, boolean allowRestart) {
    // 关键点 1
    cleanUpApplicationRecordLocked(app, restarting, allowRestart, -1);
    // 关键点 2
    // Remove this application's activities from active lists.
    boolean hasVisibleActivities = mMainStack.removeHistoryRecordsForAppLocked(app);
    app.activities.clear();
    // 关键点 3
    if (!restarting) {
        if (!mMainStack.resumeTopActivityLocked(null)) {
            // If there was nothing to resume, and we are not already
            // restarting this process, but there is a visible activity that
            // is hosted by the process  then make sure all visible
            // activities are running, taking care of restarting this
            // process.
            if (hasVisibleActivities)
                mMainStack.ensureActivitiesVisibleLocked(null, 0);
        }
    }
}
```

- 看关键点 1，cleanUpApplicationRecordLocked，主要负责清理一些 Providers，receivers，service 之类的信息，并且在清理过程中根据配置的一些信息决定是否需要重建进程并启动，关键点 2 就是关系到唤起流程的判断，关键点 3，主要是被杀的进程是否是当前前台进程，如果是，需要重建，并立即显示：先简单看 cleanUpApplicationRecordLocked 的清理流程

```
private final void cleanUpApplicationRecordLocked(ProcessRecord app,
                                                  boolean restarting, boolean allowRestart, int index) {
    // <!--清理 service-->
    mServices.killServicesLocked(app, allowRestart);
    boolean restart = false;
    // <!--清理 Providers.-->
    if (!app.pubProviders.isEmpty()) {
        Iterator<ContentProviderRecord> it = app.pubProviders.values().iterator();
        while (it.hasNext()) {
            ContentProviderRecord cpr = it.next();
            app.pubProviders.clear();
        }
        // <!--清理 receivers.-->
        // Unregister any receivers.
        if (app.receivers.size() > 0) {
            Iterator<ReceiverList> it = app.receivers.iterator();
            while (it.hasNext())
                removeReceiverLocked(it.next());
            app.receivers.clear();
        }
        // 关键点 1，进程是够需要重启，
        if (restart && !app.isolated) {
            // We have components that still need to be running in the
            // process, so re-launch it.
            mProcessNames.put(app.processName, app.uid, app);
            startProcessLocked(app, "restart", app.processName);
        }
    }
```

- 从关键点 1 就能知道，这里是隐藏了进程是否需要重启的逻辑，比如一个 Service 设置了 START_STICKY，被杀后，就需要重新唤起，这里也是流氓软件肆虐的原因。再接着看 mMainStack.removeHistoryRecordsForAppLocked(app)，对于直观理解 APP 重建，这句代码处于核心的地位，

```
boolean removeHistoryRecordsForAppLocked(ProcessRecord app) {
    while (i > 0) {
        i--;
        ActivityRecord r = (ActivityRecord)mHistory.get(i);
        if (r.app == app) {
            boolean remove;
            // <!--关键点 1-->
            if ((!r.haveState && !r.stateNotNeeded) || r.finishing) {
                remove = true;
            } else if (r.launchCount > 2 &&
                        remove = true;
                        } else {
                    // 一般来讲，走 false
```

```
                remove = false;
            }
        // <!--关键点 2-->
        if (remove) {
            removeActivityFromHistoryLocked(r);
        } else {
            r.app = null;
        }
        return hasVisibleActivities;
    }
}
}
```

- 在 Activity 跳转的时候，准确的说，在 stopActivity 之前，会保存 Activity 的现场，这样在 AMS 端 r.haveState==true，也就是说，其 ActivityRecord 不会被从 ActivityStack 中移除，同时 ActivityRecord 的 app 字段被置空，这里在恢复的时候，是决定 resume 还是重建的关键。接着往下看 moveTaskToFrontLocked，这个函数在 ActivityStack 中，主要管理 ActivityRecord 栈的，所有 start 的 Activity 都在 ActivityStack 中保留一个 ActivityRecord，这个也是 AMS 管理 Activiyt 的一个依据，最终 moveTaskToFrontLocked 会调用 resumeTopActivityLocked 来唤起 Activity，AMS 获取即将 resume 的 Activity 信息的方式主要是通过 ActivityRecord，它并不知道 Activity 本身是否存活，获取之后，AMS 在唤醒 Activity 的环节才知道 App 或者 Activity 被杀死，具体看一下 resumeTopActivityLocked 源码：

```
final boolean resumeTopActivityLocked(ActivityRecord prev, Bundle options) {
    // 关键点 1
    if (next.app != null && next.app.thread != null) {
    } else {
        // Whoops, need to restart this activity!
        startSpecificActivityLocked(next, true, true);
    }
    return true;
}
```

- 看关键点 1 的判断条件，由于已经将 ActivityRecord 的 app 字段置空，AMS 就知道了这个 APP 或者 Activity 被异常杀死过，因此，就会走 startSpecificActivityLocked 进行重建。其实仔细想想很简单，对于主动调用 finish 的，AMS 并不会清理掉 ActivitRecord，在唤起 APP 的时候，如果 AMS 检测到 APP 还存活，就走 scheduleResumeActivity 进行唤起上一个 Activity，但是如果 APP 或者 Activity 被异常杀死过，那么 AMS 就通过 startSpecificActivityLocked 再次将 APP 重建，并且将最后的 Activity 重建。

1.3 APP 存活，但是 Activity 被后台杀死的情况下 AMS 是如何恢复现场的

- 还有一种可能，APP 没有被 kill，但是 Activity 被 Kill 掉了，这个时候会怎么样。首先，Activity 的管理是一定通过 AMS 的，Activity 的 kill 一定是是 AMS 操刀的，是有记录的，严格来说，这种情况并不属于后台杀死，因为这属于 AMS 正常的管理，在可控范围，比如打开了开发者模式中的“不保留活动”，这个时候，虽然会杀死 Activity，但是仍然保留了 ActivitRecord，所以再唤醒，或者回退的的时候仍然有迹可循，看一下 ActivityStack 的 Destroy 回调代码，

```
final boolean destroyActivityLocked(ActivityRecord r,
                                    boolean removeFromApp, boolean oomAdj, String reason) {
    if (hadApp) {
        boolean skipDestroy = false;
        try {
            // 关键代码 1
            r.app.thread.scheduleDestroyActivity(r.appToken, r.finishing,
                                                 r.configChangeFlags);
            if (r.finishing && !skipDestroy) {
                if (DEBUG_STATES) Slog.v(TAG, "Moving to DESTROYING: " + r
                                         + " (destroy requested)");
                r.state = ActivityState.DESTROYING;
                Message msg = mHandler.obtainMessage(DESTROY_TIMEOUT_MSG);
                msg.obj = r;
                mHandler.sendMessageDelayed(msg, DESTROY_TIMEOUT);
            } else {
                // 关键代码 2
                r.state = ActivityState.DESTROYED;
                if (DEBUG_APP) Slog.v(TAG, "Clearing app during destroy for activity " + r);
```

```
                r.app = null;
            }
        }
        return removedFromHistory;
    }
}
```

- 这里有两个关键啊你单，1 是告诉客户端的 AcvitityThread 清除 Activity，2 是标记如果 AMS 自己非正常关闭的 Activity，就将 ActivityRecord 的 state 设置为 ActivityState.DESTROYED，并且清空它的 ProcessRecord 引用：r.app = null。这里是唤醒时候的一个重要标志，通过这里 AMS 就能知道 Activity 被自己异常关闭了，设置 ActivityState.DESTROYED 是为了让避免后面的清空逻辑。

```
final void activityDestroyed(IBinder token) {
    synchronized (mService) {
        final long origId = Binder.clearCallingIdentity();
        try {
            ActivityRecord r = ActivityRecord.forToken(token);
            if (r != null) {
                mHandler.removeMessages(DESTROY_TIMEOUT_MSG, r);
            }
            int index = indexOfActivityLocked(r);
            if (index >= 0) {
                // 1  <!--这里会是否从 history 列表移除 ActivityRecord-->
                if (r.state == ActivityState.DESTROYING) {
                    cleanUpActivityLocked(r, true, false);
                    removeActivityFromHistoryLocked(r);
                }
            }
            resumeTopActivityLocked(null);
        } finally {
            Binder.restoreCallingIdentity(origId);
        }
    }
}
```

- 看代码关键点 1，只有 r.state == ActivityState.DESTROYING 的时候，才会移除 ActivityRecord，但是对于不非正常 finish 的 Activity，其状态是不会被设置成 ActivityState.DESTROYING，是直接跳过了 ActivityState.DESTROYING，被设置成了 ActivityState.DESTROYED，所以不会 removeActivityFromHistoryLocked，也就是保留了 ActivityRecord 现场，好像也是依靠异常来区分是否是正常的结束掉 Activity。这种情况下是如何启动 Activity 的呢？通过上面两点分析，就知道了两个关键点
  - ActivityRecord 没有动 HistoryRecord 列表中移除
  - ActivityRecord 的 ProcessRecord 字段被置空，r.app = null
- 这样就保证了在 resumeTopActivityLocked 的时候，走 startSpecificActivityLocked 分支

```
final boolean resumeTopActivityLocked(ActivityRecord prev, Bundle options) {
    if (next.app != null && next.app.thread != null) {
    } else {
        // Whoops, need to restart this activity!
        startSpecificActivityLocked(next, true, true);
    }
    return true;
}
```

- 到这里，AMS 就知道了这个 APP 或者 Activity 是不是被异常杀死过，从而，决定是走 resume 流程还是 restore 流程。

1.4 App 被杀前的场景是如何保存：新 Activity 启动跟旧 Activity 的保存

- App 现场的保存流程相对是比较简单的，入口基本就是 startActivity 的时候，只要是界面的跳转基本都牵扯到 Activity 的切换跟当前 Activity 场景的保存：先画个简单的图形，开偏里面讲 FragmentActivity 的时候，简单说了一些 onSaveInstance 的执行时机，这里详细看一下 AMS 是如何管理这些跳转以及场景保存的，模拟场景：Activity A 启动 Activity B 的时候，这个时候 A 不可见，可能会被销毁，需要保存 A 的现场，这个流程是什么样的：简述如下

```
ActivityA startActivity ActivityB
ActivityA pause
ActivityB create
ActivityB start
ActivityB resume
ActivityA onSaveInstance
ActivityA stop
```

- 流程大概是如下样子：
- 现在我们通过源码一步一步跟一下，看看 AMS 在新 Activity 启动跟旧 Activity 的保存的时候，到底做了什么：跳过简单的 startActivity, 直接去 AMS 中去看

1.4.1 ActivityManagerService

```
public final int startActivityAsUser(IApplicationThread caller, String callingPackage,
                                     Intent intent, String resolvedType, IBinder resultTo,
                                     String resultWho, int requestCode, int startFlags,
                                     String profileFile, ParcelFileDescriptor profileFd, Bundle options, int userId) {
    enforceNotIsolatedCaller("startActivity");
    return mMainStack.startActivityMayWait(caller, -1, callingPackage, intent, resolvedType,
                                           resultTo, resultWho, requestCode, startFlags, profileFile, profileFd,
                                           null, null, options, userId);
}
```

1.4.2 ActivityStack

```
final int startActivityMayWait(IApplicationThread caller, int callingUid,
                               int res = startActivityLocked(caller, intent, resolvedType,
                                                             aInfo, resultTo, resultWho, requestCode, callingPid, callingUi
                                                             callingPackage, startFlags, options, componentSpecified, null)
```

- 这里通过 startActivityMayWait 启动新的 APP，或者新 Activity，这里只看简单的，至于从桌面启动 App 的流程，可以去参考更详细的文章，比如老罗的 startActivity 流程，大概就是新建 ActivityRecord，ProcessRecord 之类，并加入 AMS 中相应的堆栈等，resumeTopActivityLocked 是界面切换的统一入口，第一次进来的时候，由于 ActivityA 还在没有 pause，因此需要先暂停 ActivityA，这些完成后，

1.4.3 ActivityStack

```
final boolean resumeTopActivityLocked(ActivityRecord prev, Bundle options) {
    // <!--必须将当前 Resume 的 Activity 设置为 pause 然后 stop 才能继续-->
    // We need to start pausing the current activity so the top one
    // can be resumed
    if (mResumedActivity != null) {
        if (next.app != null && next.app.thread != null) {
            mService.updateLruProcessLocked(next.app, false);
        }
        startPausingLocked(userLeaving, false);
        return true;
    }
}
```

- 其实这里就是暂停 ActivityA，AMS 通过 Binder 告诉 ActivityThread 需要暂停的 ActivityA，ActivityThread 完成后再通过 Binder 通知 AMS，AMS 会开始 resume ActivityB，

```
private final void startPausingLocked(boolean userLeaving, boolean uiSleeping) {
    if (prev.app != null && prev.app.thread != null) {
        try {
            prev.app.thread.schedulePauseActivity(prev.appToken, prev.finishing,
                                                  userLeaving, prev.configChangeFlags);
        }
    }
}
```

1.4.4 ActivityThread

```
private void handlePauseActivity(IBinder token, boolean finished,
                                 boolean userLeaving, int configChanges) {
    ActivityClientRecord r = mActivities.get(token);
    if (r != null) {
        performPauseActivity(token, finished, r.isPreHoneycomb());
        // Tell the activity manager we have paused.
        try {
            ActivityManagerNative.getDefault().activityPaused(token);
        } catch (RemoteException ex) {
        }
    }
}
```

- AMS 收到 ActivityA 发送过来的 pause 消息之后，就会唤起 ActivityB，入口还是 resumeTopActivityLocked，唤醒 B，之后还会 A 给进一步 stop 掉，这个时候就牵扯到现场的保存，

1.4.5 ActivityStack

```
private final void completePauseLocked() {
    if (!mService.isSleeping()) {
        resumeTopActivityLocked(prev);
    } else {

    }
}
```

- ActivityB 如何启动的，本文不关心，只看 ActivityA 如何保存现场的，ActivityB 起来后，会通过 ActivityStack 的 stopActivityLocked 去 stop ActivityA，

```
private final void stopActivityLocked(ActivityRecord r) {
    if (mMainStack) {
        r.app.thread.scheduleStopActivity(r.appToken, r.visible, r.configChangeFlags);
    }
}
```

- 回看 APP 端，看一下 ActivityThread 中的调用：首先通过 callActivityOnSaveInstanceState，将现场保存到 Bundle 中去，

```
private void performStopActivityInner(ActivityClientRecord r,
                                      StopInfo info, boolean keepShown, boolean saveState) {
    // Next have the activity save its current state and managed dialogs
    if (!r.activity.mFinished && saveState) {
        if (r.state == null) {
            state = new Bundle();
            state.setAllowFds(false);
            mInstrumentation.callActivityOnSaveInstanceState(r.activity, state);
            r.state = state;
        }
    }
}
```

- 之后，通过 ActivityManagerNative.getDefault().activityStopped，通知 AMS Stop 动作完成，在通知的时候，还会将保存的现场数据带过去。

```
private static class StopInfo implements Runnable {
    ActivityClientRecord activity;
    Bundle state;
    Bitmap thumbnail;
    CharSequence description;
    @Override public void run() {
        // Tell activity manager we have been stopped.
        try {
            ActivityManagerNative.getDefault().activityStopped(
                activity.token, state, thumbnail, description);
        } catch (RemoteException ex) {
        }
    }
}
```

- 通过上面流程，AMS 不仅启动了新的 Activity，同时也将上一个 Activity 的现场进行了保存，及时由于种种原因上一个 Actiivity 被杀死，在回退，或者重新唤醒的过程中 AMS 也能知道如何唤起 Activiyt，并恢复。
- 现在解决两个问题，1、如何保存现场，2、AMS 怎么判断知道 APP 或者 Activity 是否被异常杀死，那么就剩下最后一个问题了，AMS 如何恢复被异常杀死的 APP 或者 Activity 呢。

1.5 Activity 或者 Application 恢复流程

1.5.1 Application 被后台杀死

- 其实在讲解 AMS 怎么判断知道 APP 或者 Activity 是否被异常杀死的时候，就已经涉及了恢复的逻辑，也知道了一旦 AMS 知道了 APP 被后台杀死了，那就不是正常的 resuem 流程了，而是要重新 laucher，先来看一下整个 APP 被干掉的会怎么处理，看 resumeTopActivityLocked 部分，从上面的分析已知，这种场景下，会因为 Binder 通信抛异常走异常分支，如下：

```
final boolean resumeTopActivityLocked(ActivityRecord prev, Bundle options) {
    if (next.app != null && next.app.thread != null) {
        if (DEBUG_SWITCH)
            Slog.v(TAG, "Resume running: " + next);
        try {
        } catch (Exception e) {
            // Whoops, need to restart this activity!
            // 这里是知道整个 app 被杀死的
            Slog.i(TAG, "Restarting because process died: " + next);
            next.state = lastState;
            mResumedActivity = lastResumedActivity;
            Slog.i(TAG, "Restarting because process died: " + next);

            startSpecificActivityLocked(next, true, false);
            return true;
        }
    }
}
```

- 从上面的代码可以知道，其实就是走 startSpecificActivityLocked，这根第一次从桌面唤起 APP 没多大区别，只是有一点需要注意，那就是这种时候启动的 Activity 是有上一次的现场数据传递过得去的，因为上次在退到后台的时候，所有 Activity 界面的现场都是被保存了，并且传递到 AMS 中去的，那么这次的恢复启动就会将这些数据返回给 ActivityThread，再来仔细看一下 performLaunchActivity 里面关于恢复的特殊处理代码：

```
private Activity performLaunchActivity(ActivityClientRecord r, Intent customIntent) {
    ActivityInfo aInfo = r.activityInfo;
    Activity activity = null;
    try {
        java.lang.ClassLoader cl = r.packageInfo.getClassLoader();
        activity = mInstrumentation.newActivity(
            cl, component.getClassName(), r.intent);
        StrictMode.incrementExpectedActivityCount(activity.getClass());
        r.intent.setExtrasClassLoader(cl);
        if (r.state != null) {
            r.state.setClassLoader(cl);
        }
    } catch (Exception e) {
    }
    try {
        Application app = r.packageInfo.makeApplication(false, mInstrumentation);
        关键点 1
            mInstrumentation.callActivityOnCreate(activity, r.state);
        r.activity = activity;
        r.stopped = true;
        if (!r.activity.mFinished) {
            activity.performStart();
            r.stopped = false;
        }
        // 关键点 1
        if (!r.activity.mFinished) {
            if (r.state != null) {
                mInstrumentation.callActivityOnRestoreInstanceState(activity, r.state);
            }
```

```
        }
        if (!r.activity.mFinished) {
            activity.mCalled = false;
            mInstrumentation.callActivityOnPostCreate(activity, r.state);

        }
    }
}
```

- 看一下关键点 1 跟 2，先看关键点 1，mInstrumentation.callActivityOnCreate 会回调 Actiivyt 的 onCreate，这个函数里面其实主要针对 FragmentActivity 做一些 Fragment 恢复的工作，ActivityClientRecord 中的 r.state 是 AMS 传给 APP 用来恢复现场的，正常启动的时候，这些都是 null。再来看关键点 2 ，在 r.state != null 非空的时候执行 mInstrumentation.callActivityOnRestoreInstanceState，这个函数默认主要就是针对 Window 做一些恢复工作，比如 ViewPager 恢复之前的显示位置等，也可以用来恢复用户保存数据。

1.5.2 Application 没有被后台杀死

- 打开开发者模式”不保留活动 “，就是这种场景，在上面的分析中，知道，AMS 主动异常杀死 Activity 的时候，将 AcitivityRecord 的 app 字段置空，因此 resumeTopActivityLocked 同整个 APP 被杀死不同，会走下面的分支

```
final boolean resumeTopActivityLocked(ActivityRecord prev, Bundle options) {
    if (next.app != null && next.app.thread != null) {
    } else {
        关键点 1 只是重启Activity，可见这里其实是知道的，进程并没死，
            // Whoops, need to restart this activity!
            startSpecificActivityLocked(next, true, true);
    }
    return true;
}
```

- 虽然不太一样，但是同样走 startSpecificActivityLocked 流程，只是不新建 APP 进程，其余的都是一样的，不再讲解。
- 主动清除最近任务跟异常杀死的区别：ActivityStack 是否正常清除
- 恢复的时候，为什么是倒序恢复：因为这是 ActivityStack 中的 HistoryRecord 中栈的顺序，严格按照 AMS 端来
- 一句话概括 Android 后台杀死恢复原理：Application 进程被 Kill，但现场被 AMS 保存，AMS 能根据保存恢复 Application 现场