Android APK Crash重启机制

Last edited by **sunchao** 3 weeks ago

Android APK Crash重启机制

Table of Contents

在介绍死亡重启机制，我们先介绍crash流程，有助于我们更能理解如何死亡如何重启.

1. 应用crash流程

应用都是由Zygote fork孵化而来，分为system_server系统进程和各种应用进程，在这些进程创建之初会设置未捕获异常的处理器，当系统抛出未捕获的异常时，最终都交给异常处理器

- 对于system_server进程：system_server启动过程中由RuntimeInit.java的commonInit方法设置LoggingHandler与KillApplicationHandler，用于处理未捕获异常;
- 对于非system_server进程的应用：进程创建过程中，同样会调用RuntimeInit.java的commonInit方法设置LoggingHandler与KillApplicationHandler。

1.1. Crash异常log

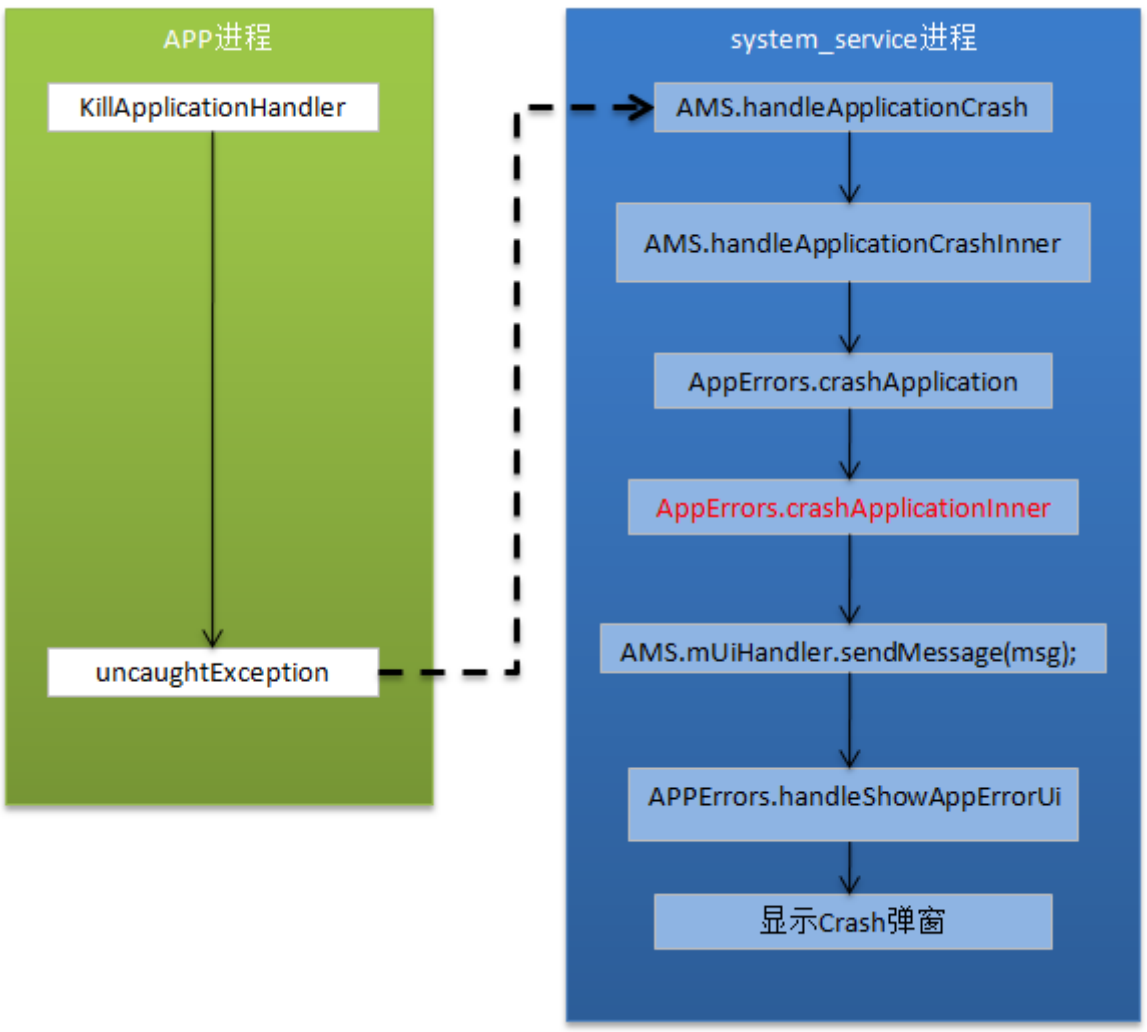
当system进程crash的信息：

- Log会输出开头是*** FATAL EXCEPTION IN SYSTEM PROCESS [线程名]，接着输出发生crash时的调用栈信息.

当app进程crash时的信息：

- log会输出开头是FATAL EXCEPTION: [线程名]， 紧接着 Process: [进程名], PID: [进程id]，最后输出发生crash时的调用栈信息.

1.2. crash异常处理流程



上面流程图中，重点部分就是我标注的红色的crashApplicationInner方法，这方法主要做了：

- 1.引入RescueParty机制（拯救机制），根据Persistent App的crash的重启次数，提高等级，然后提供不同的解决方案。（RescueParty机制android O出来的）.
- 2.处理有IActivityController的情况，如果Controller已经处理错误，则不会显示错误框
- 3.调用makeAppCrashingLocked方法，主要是做了忽略当前的广播接收器的接收，停止进程中的所有activity，contentprovider，限制应用crsh次数等。
- 4.发送SHOW_ERROR_UI_MSG给AMS的mUiHandler，将弹出一个错误对话框，提示用户某进程crash，等待用户选择，5分钟操作等待。
- 5.调用AppErrorResult的get方法，该方法内部调用了wait方法，故为线程等待状态，当用户处理了对话框后会调用AppErrorResult的set方法，该方法内部调用了notifyAll()方法来唤醒线程。
- 6.判断用户点击弹框的结果。根据点击操作的不同，做不同的事。总共有不在提示错误，尝试重启进程，强行结束进程，停止进程并报告错误，这四种结果。

1.3. crash重启次数限制

为了保护手机，提升用户体验，当一个应用多次发生crash的时候，android系统将会做出一些限制.

1，persistent APP ，1分钟crash次数超过或等于两次，则只进行结束当前进程的操作，并且自动重启.但是persistent的APP 引入了RescueParty机制，RescueParty会有五个等级：

```
    private static final int LEVEL_NONE = 0;
    private static final int LEVEL_RESET_SETTINGS_UNTRUSTED_DEFAULTS = 1;
    private static final int LEVEL_RESET_SETTINGS_UNTRUSTED_CHANGES = 2;
    private static final int LEVEL_RESET_SETTINGS_TRUSTED_DEFAULTS = 3;
    private static final int LEVEL_FACTORY_RESET = 4;
```

默认等级是LEVEL_NONE也就是0，最高等级LEVEL_FACTORY_RESET.如果30秒内发生crash次数达到5次，则提升一个等级，并且执行不同的拯救方法，但是只要有一次crash的时间间隔超过30秒，就不会提升等级，保持原来等级，并且重新计算。

等级1对SettingS属性值做修改，重置所有非系统进程对settings的属性值，如果非系统进程或者系统进程存在默认属性值，则将属性设置为默认属性值，否则将删除该属性值。

等级2也是settings属性值做修改，删除所有非系统进程对settings的属性值，如果某个属性是由非系统进程设置的，但是系统进程没有提供该属性的默认值，那么该属性值将被删除，否则该属性将被设置为其默认值。

等级3也是settings属性值做修改，重置所有系统进程默认的Settings属性值，并且删除所有非系统进程的属性值。

等级4就是最高等级，达到等级4将会进入recovery模式，让用户重置data分区。

以上在源码android8.1验证过了。

注：RescueParty机制在 eng版本会被禁用. userdebug版本的usb正在连接中会被禁用.persist.sys.disable_rescue 属性为true也会禁用.

2.非persistent APP ，1分钟crash次数超过或等于两次，当调用者的Intent带有FLAG_FROM_BACKGROUND标志的话，则不会再次启动.（针对的是应用出现crash的情况.如果应用不是crash而是被kill，情况不相同）

举例说明： 我们点击桌面上的非persistent应用出现crash，不管是Sevice Crash还是什么，都不会自动重启，接着我们再次点击，又出现了一次crash， 刚好两次crash的间隔时间在1分钟，那么此时带有FLAG_FROM_BACKGROUND标志的intent的就启动不了。

1.3.1. 对于非persistent APP crash代码分析

当一个非persistent进程的1分钟crash次数超过或等于两次，则不会再重启，除非显示调用。

调用的方法如下：

- ASS.handleAppCrashLocked, 直接结束该应用所有activity
- AMS.removeProcessLocked，杀死该进程以及同一个进程组下的所有进程
- ASS.resumeTopActivitiesLocked，恢复栈顶第一个非finishing状态的activity

非persistent的APP 重启的次数主要发生在makeAppCrashingLocked调用handleAppCrashLocked函数里面发生。

```
boolean handleAppCrashLocked(ProcessRecord app, String reason,
        String shortMsg, String longMsg, String stackTrace, AppErrorDialog.Data data) {
    final long now = SystemClock.uptimeMillis();
    //获取设置,后台进程确认是否需要弹出ANR提示框。
    final boolean showBackground = Settings.Secure.getInt(mContext.getContentResolver(),
            Settings.Secure.ANR_SHOW_BACKGROUND, 0) != 0;
```

```
//获取该进程的状态是否是系统绑定而托管前台服务。
final boolean procIsBoundForeground =
    (app.curProcState == ActivityManager.PROCESS_STATE_BOUND_FOREGROUND_SERVICE);

Long crashTime;
Long crashTimePersistent;
boolean tryAgain = false;

if (!app.isolated) {
    crashTime = mProcessCrashTimes.get(app.info.processName, app.uid);
    crashTimePersistent = mProcessCrashTimesPersistent.get(app.info.processName, app.ui
} else {
    crashTime = crashTimePersistent = null;
}

//运行在当前进程中的所有服务的crash次数间隔小于1分钟就执行加1操作，如果是前台服务，crash次数只要小于1(
//点击dialog弹框中的Open app again重启按钮
for (int i = app.services.size() - 1; i >= 0; i--) {
    ServiceRecord sr = app.services.valueAt(i);
    if (now > sr.restartTime + ProcessList.MIN_CRASH_INTERVAL) {
        sr.crashCount = 1;
    } else {
        sr.crashCount++;
    }
    if (sr.crashCount < mService.mConstants.BOUND_SERVICE_MAX_CRASH_RETRY
            && (sr.isForeground || procIsBoundForeground)) {
        tryAgain = true;
    }
}
//当同一个进程，连续两次crash的时间间隔小于1分钟时，则认为crash太过于频繁
if (crashTime != null && now < crashTime + ProcessList.MIN_CRASH_INTERVAL) {
    EventLog.writeEvent(EventLogTags.AM_PROCESS_CRASHED_TOO_MUCH,
            app.userId, app.info.processName, app.uid);
    //handleAppCrashLocked实际上就是遍历所有该ProcessRecord的所有ActivityRecord，并结束该Acit
    mService.mStackSupervisor.handleAppCrashLocked(app);
    //对于非persistent app，不再重启，除非用户显式的调用
    if (!app.persistent) {
        EventLog.writeEvent(EventLogTags.AM_PROC_BAD, app.userId, app.uid,
                app.info.processName);
        if (!app.isolated) {
            //将非孤立的app加入到mBadProcesses
            mBadProcesses.put(app.info.processName, app.uid,
                    new BadProcessInfo(now, shortMsg, longMsg, stackTrace));
            //因为已经加入到了mBadProcesses了，不会在重启，所以ProcessCrashTimes将会不再统计次数
            mProcessCrashTimes.remove(app.info.processName, app.uid);
        }
        app.bad = true;
        app.removed = true;
        //移除进程的所有服务，假如tryAgain等于true，发送消息，等待唤醒。否则就是移除所有服务
        mService.removeProcessLocked(app, false, tryAgain, "crash");
        //恢复最顶部的Activity
        mService.mStackSupervisor.resumeFocusedStackTopActivityLocked();
        if (!showBackground) {
            return false;
        }
    }
    //恢复最顶部的Activity
    mService.mStackSupervisor.resumeFocusedStackTopActivityLocked();
} else {
    //finishTopRunningActivityLocked最终会调用到activity的pause方法。
    TaskRecord affectedTask =
            mService.mStackSupervisor.finishTopRunningActivityLocked(app, reason);
    if (data != null) {
        data.task = affectedTask;
    }
    if (data != null && crashTimePersistent != null
            && now < crashTimePersistent + ProcessList.MIN_CRASH_INTERVAL) {
        data.repeating = true;
    }
}

if (data != null && tryAgain) {
    data.isRestartableForService = true;
}
```

```
    //当桌面应用crash，并且被三方app所取代，那么需要清空桌面应用的偏爱选项。
    final ArrayList<ActivityRecord> activities = app.activities;
    if (app == mService.mHomeProcess && activities.size() > 0
            && (mService.mHomeProcess.info.flags & ApplicationInfo.FLAG_SYSTEM) == 0) {
        for (int activityNdx = activities.size() - 1; activityNdx >= 0; --activityNdx) {
            final ActivityRecord r = activities.get(activityNdx);
            if (r.isHomeActivity()) {
                Log.i(TAG, "Clearing package preferred activities from " + r.packageName);
                try {
                    ActivityThread.getPackageManager()
                            .clearPackagePreferredActivities(r.packageName);
                } catch (RemoteException c) {
                }
            }
        }
    }
    //非孤立的APP添加进程crash时间点
    if (!app.isolated) {
        mProcessCrashTimes.put(app.info.processName, app.uid, now);
        mProcessCrashTimesPersistent.put(app.info.processName, app.uid, now);
    }
    //当app存在crash的handler，那么交给其处理
    if (app.crashHandler != null) mService.mHandler.post(app.crashHandler);
    return true;
}
```

对于APP Crash的次数1分钟没有两次Crash则只调用finishTopRunningActivityLocked的方法，执行结束栈顶正在运行activity.

对于persistent APP 两次crash次数间隔小于1分钟，只调用了resumeFocusedStackTopActivityLocked的方法，恢复栈顶第一个非finishing状态的activity.

1.4. RescueParty系统拯救机制：

Adroid系统在很多情况下都会进入到一种无法自主恢复的状态下：例如无法开机，常驻系统进程无限crash等等，往往在这些情况下手机已经无法正常使用了,在Android O上加了一个救援的机制就是来解决这些问题的，这个机制叫：RescueParty.

RescueParty的原理大致为：同一个uid的应用发生多次异常，RescueParty会根据该uid记录发生的次数，当次数达到默认次数后会调整拯救的策略。拯救策略等级分为：

1.NONE =0
2.RESET_SETTINGS_UNTRUSTED_DEFAULTS =1
3.RESET_SETTINGS_UNTRUSTED_CHANGES =2
4.RESET_SETTINGS_TRUSTED_DEFAULTS =3
5.FACTORY_RESET =4
默认等级是LEVEL_NONE也就是0，最高等级LEVEL_FACTORY_RESET.如果30秒内发生crash次数达到5次，则提升一个等级，并且执行不同的拯救方法，但是只要有一次crash的时间间隔超过30秒，就不会提升等级，保持原来等级，并且重新计算。

等级1对SettingS属性值做修改，重置所有非系统进程对settings的属性值，如果非系统进程或者系统进程存在默认属性值，则将属性设置为默认属性值，否则将删除该属性值。

等级2也是settings属性值做修改，删除所有非系统进程对settings的属性值，如果某个属性是由非系统进程设置的，但是系统进程没有提供该属性的默认值，那么该属性值将被删除，否则该属性将被设置为其默认值。

等级3也是settings属性值做修改，重置所有系统进程默认的Settings属性值，并且删除所有非系统进程的属性值。

等级4就是最高等级，达到等级4将会进入recovery模式，让用户重置data分区。

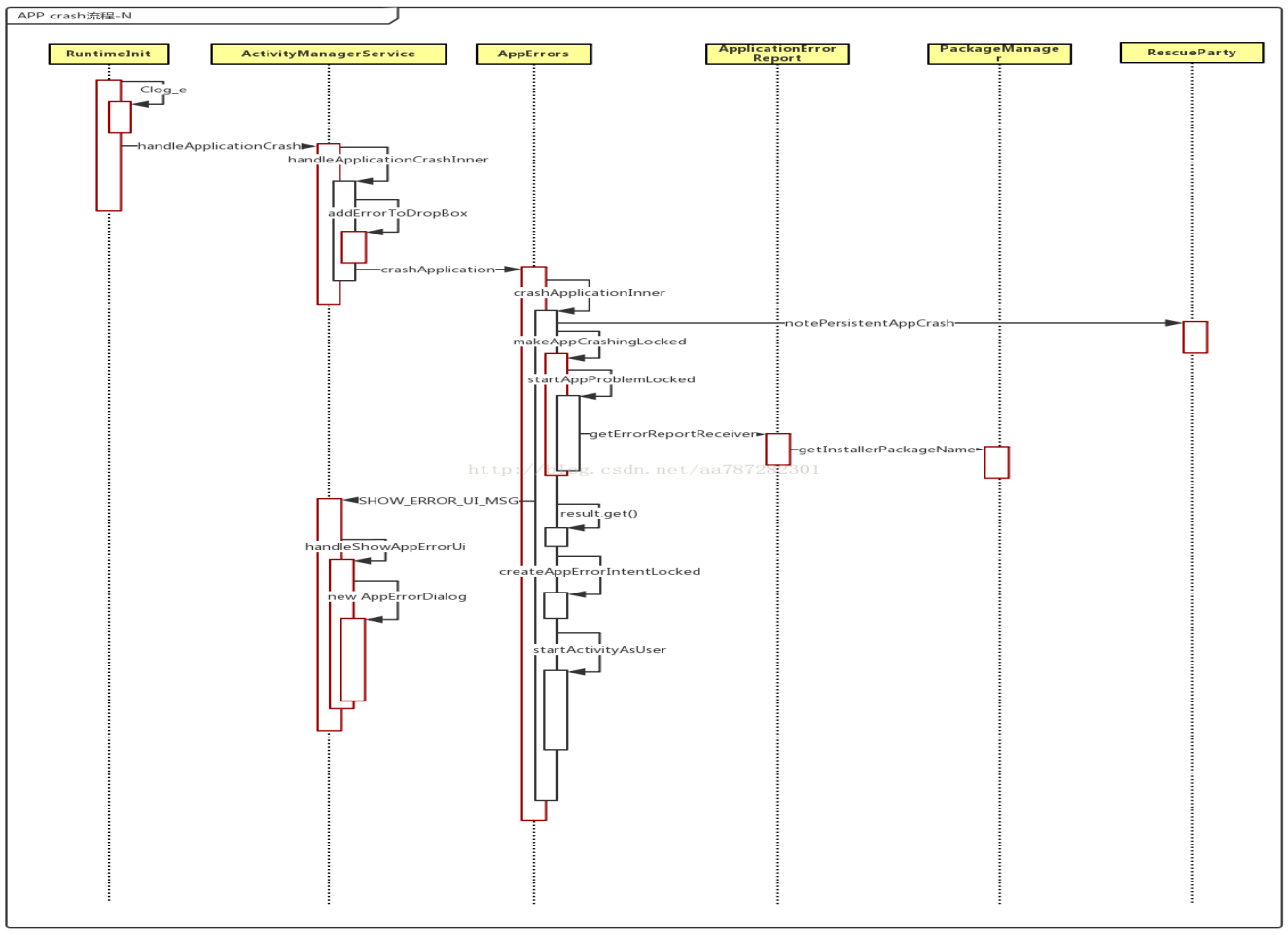
那么哪些场景会造成触发这个机制呢?

- persistent app在 30 秒内崩溃 5 次调整一次级别
- system_server在 5 分钟内重启 5 次调整一次级别

RescueParty机制 在以下情况会被禁用：

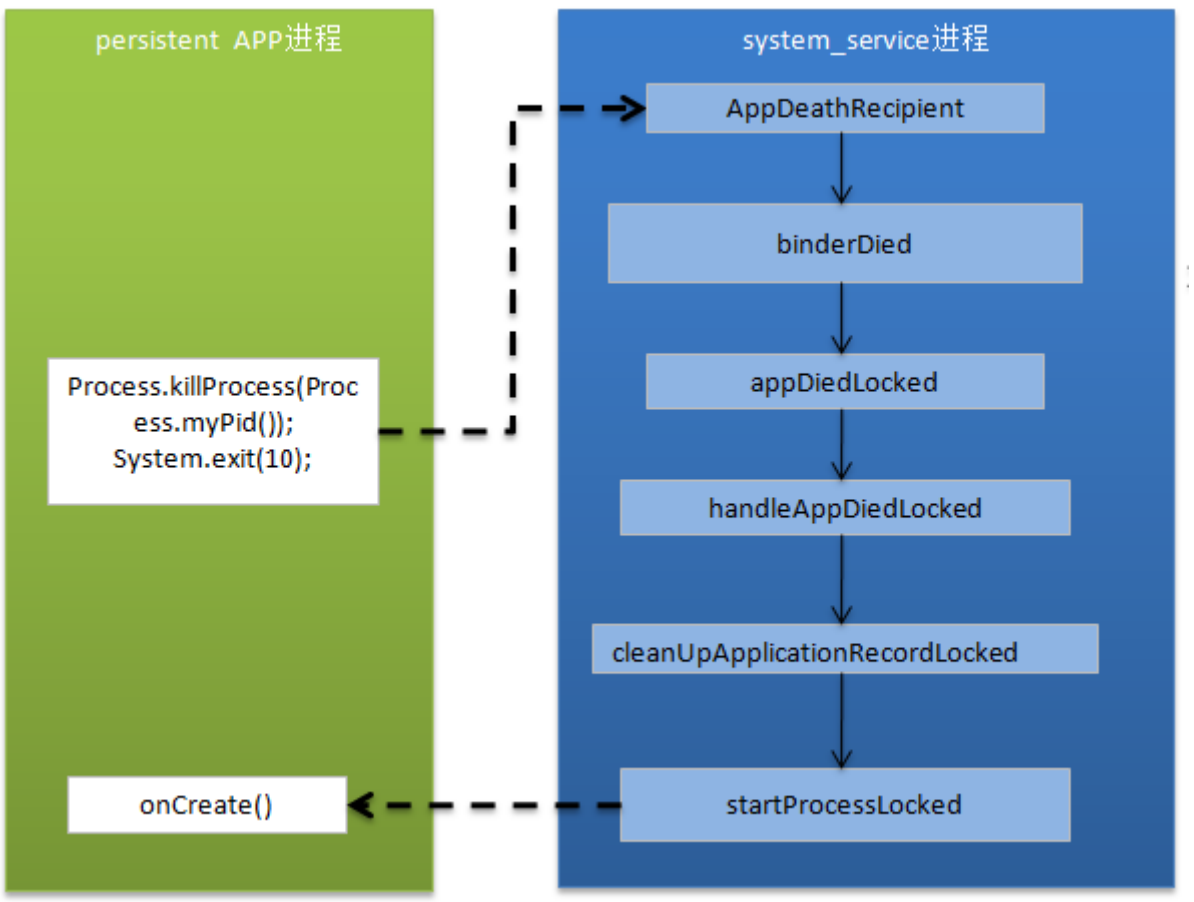
- eng版本会被禁用.
- userdebug版本，并且usb正在连接中.
- persist.sys.disable_rescue 属性为true.

对于persistent app 的拯救机制流程图:



2. 应用重启

流程图



我们知道进程在创建的时候调用了AMS.attachApplicationLocked函数，

```
    private final boolean attachApplicationLocked(IApplicationThread thread,
            int pid) {

       ......
       ......
        try {
            //创建binder死亡通知,
            AppDeathRecipient adr = new AppDeathRecipient(
                    app, pid, thread);
            thread.asBinder().linkToDeath(adr, 0);
            app.deathRecipient = adr;
        } catch (RemoteException e) {
            app.resetPackageList(mProcessStats);
            startProcessLocked(app, "link fail", processName);
            return false;
        }
        .......
        .......
    }
```

```
    private final class AppDeathRecipient implements IBinder.DeathRecipient {
        final ProcessRecord mApp;
        final int mPid;
        final IApplicationThread mAppThread;

        //dinder死亡回调
        @Override
        public void binderDied() {
            if (DEBUG_ALL) Slog.v(
                TAG, "Death received in " + this
                + " for thread " + mAppThread.asBinder());
            synchronized(ActivityManagerService.this) {
                appDiedLocked(mApp, mPid, mAppThread, true);
            }
        }
    }
```

AppDeathRecipient实现了IBinder.DeathRecipient接口，当binder服务端挂了之后，便会通过binder的DeathRecipient来通知AMS进行相应的清理收尾工作。crash的进程会被kill掉，那么当该进程被杀，则会回调到binderDied()方法。

```
    final void appDiedLocked(ProcessRecord app, int pid, IApplicationThread thread,
            boolean fromBinderDied) {

            .......
            .......

            EventLog.writeEvent(EventLogTags.AM_PROC_DIED, app.userId, app.pid, app.proce
ssName,
                    app.setAdj, app.setProcState);
            //清除AM中的死的进程，和进程中的所有链接
            handleAppDiedLocked(app, false, true);

            if (doOomAdj) {
                updateOomAdjLocked();
            }
            if (doLowMem) {
                doLowMemReportIfNeededLocked(app);
            }
        } else if (app.pid != pid) {

            EventLog.writeEvent(EventLogTags.AM_PROC_DIED, app.userId, app.pid, app.proce
ssName);
        } else if (DEBUG_PROCESSES) {

        }
    }

     private final void handleAppDiedLocked(ProcessRecord app,
            boolean restarting, boolean allowRestart) {
        int pid = app.pid;
        //该方法清理应用程序service, BroadcastReceiver, ContentProvider，process相关信息，重新
启动persistent APP等
        boolean kept = cleanUpApplicationRecordLocked(app, restarting, allowRestart, -1,
                false /*replacingPid*/);
    }

    private final boolean cleanUpApplicationRecordLocked(ProcessRecord app,
        boolean restarting, boolean allowRestart, int index, boolean replacingPid) {
        ......
        ......
        //清理service信息
        mServices.killServicesLocked(app, allowRestart);
        ....
        ...
        //清理ContentProvider
        for (int i = app.pubProviders.size() - 1; i >= 0; i--) {
            ContentProviderRecord cpr = app.pubProviders.valueAt(i);
            final boolean always = app.bad || !allowRestart;
            boolean inLaunching = removeDyingProviderLocked(app, cpr, always);
            if ((inLaunching || always) && cpr.hasConnectionOrHandle()) {
                // We left the provider in the launching list, need to
                // restart it.
```

```
                restart = true;
            }

            cpr.provider = null;
            cpr.proc = null;
        }
        app.pubProviders.clear();
        .....
        ......
        // 取消注册的广播接收器
        for (int i = app.receivers.size() - 1; i >= 0; i--) {
            removeReceiverLocked(app.receivers.valueAt(i));
        }
        app.receivers.clear();
        ......
        ......
        if (!app.persistent || app.isolated) {
            if (DEBUG_PROCESSES || DEBUG_CLEANUP) Slog.v(TAG_CLEANUP,
                    "Removing non-persistent process during cleanup: " + app);
            if (!replacingPid) {
                removeProcessNameLocked(app.processName, app.uid, app);
            }
            if (mHeavyWeightProcess == app) {
                mHandler.sendMessage(mHandler.obtainMessage(CANCEL_HEAVY_NOTIFICATION_MS
G,
                        mHeavyWeightProcess.userId, 0));
                mHeavyWeightProcess = null;
            }
        } else if (!app.removed) {
             //把persistent APP添加到正在启动的持久应用程序的列表中
            if (mPersistentStartingProcesses.indexOf(app) < 0) {
                mPersistentStartingProcesses.add(app);
                restart = true;
            }
        }
        ....
        ....

        if (restart && !app.isolated) {
            // We have components that still need to be running in the
            // process, so re-launch it.
            if (index < 0) {
                ProcessList.remove(app.pid);
            }
            //将要运行的应用程序添加到mProcessNames中
            addProcessNameLocked(app);
            //重新启动persistent APP
            startProcessLocked(app, "restart", app.processName);
            return true;
        } else if (app.pid > 0 && app.pid != MY_PID) {
        ......
            }



    }
```

3. 总结

1.发生crash所在的进程，在创建之初便准备好了 LoggingHandler，KillApplicationHandler监听，用来处理未捕获的异常。

LoggingHandler 负责输出相关异常Log，KillApplicationHandler用来处理crash相关事件。

2.调用当前进程中的AMS.handleApplicationCrash；经过binder ipc机制，传递到system_server进程

3.获取进程名字,从mProcessNames查找到目标进程的ProcessRecord对象；并将进程crash信息输出到目录/data/system/dropbox

4.执行AppErrors.crashApplicationInner ，首先对于persistent APP使用RescueParty机制进行检查，根据发生crash的时间和次数进行调整等级，执行不同的方法。

5.再执行makeAppCrashingLocked方法，忽略当前的广播接收器，停止当前进程中所有activity中的WMS的冻结屏幕消息，并执行相关一些屏幕相关操作。

6.执行handleAppCrashLocked的方法

当1分钟内同一进程连续crash两次时，且非persistent进程，则直接结束该应用所有activity，并杀死该进程以及同一个进程组下的所有进程。然后再恢复栈顶第一个非finishing状态的activity;

当1分钟内同一进程连续crash两次时，且persistent进程，，则只执行恢复栈顶第一个非finishing状态的activity;

当1分钟内同一进程未发生连续crash两次时，则执行结束栈顶正在运行activity的流程。

7.通过mUiHandler发送消息SHOW_ERROR_MSG，弹出crash对话框等待用户选择，5分钟操作等待，根据不同的操作，执行不同的方法；到此，system_server进程执行完毕。

8.crash进程开始执行杀掉当前进程的操作，当crash进程被杀，通过binder死亡通知，告知system_server进程来执行appDiedLocked()， 最后，执行清理应用相关的activity/service/ContentProvider/receiver等组件信息，如果是persistent APP 则通过binde的死亡通知binderDied调用startProcessLocked重新启动进程。