

# Android 插件化原理解析（5）：Activity 生命周期管理（下）

原创 2016-05-22 伯乐在线/田维术 安卓应用频道

(点击上方公众号，可快速关注)

来源：伯乐在线专栏作者 - weishu

链接：<http://android.jobbole.com/83168/>

[点击 → 了解如何加入专栏作者](#)

[接上文](#)

## 僵尸还魂——拦截Callback从恢复真身

行百里者半九十。现在我们的startActivity启动一个没有显式声明的Activity已经不会抛异常了，但是要真正正确地把TargetActivity启动起来，还有一些事情要做。其中最重要的一点是，我们用替身StubActivity临时换了TargetActivity，肯定需要在『合适的』时候替换回来；接下来我们就完成这个过程。

在AMS进程里面我们是没有办法换回来的，因此我们要等AMS把控制权交给App所在进程，也就是上面那个『Activity启动过程简图』的第三步。AMS进程转移到App进程也是通过Binder调用完成的，承载这个功能的Binder对象是IAplicationThread；在App进程它是Server端，在Server端接受Binder远程调用的是Binder线程池，Binder线程池通过Handler将消息转发给App的主线程；（我这里不厌其烦地叙述Binder调用过程，希望读者不要反感，其一加深印象，其二懂Binder真的很重要）我们可以在这个Handler里面将替身恢复成真身。

这里不打算讲述Handler 的原理，我们简单看一下Handler是如何处理接收到的Message的，如果我们能拦截这个Message的接收过程，就有可能完成替身恢复工作；Handler类的dispatchMessage如下：

```
public void dispatchMessage(Message msg) {  
    if (msg.callback != null) {  
        handleCallback(msg);  
    } else {  
        if (mCallback != null) {  
            mCallback.handleMessage(msg);  
        } else if (mSource != null) {  
            mSource.handleMessage(msg);  
        }  
    }  
}
```

```

if (mCallback.handleMessage(msg)) {
    return;
}

}

handleMessage(msg);

}
}

```

从这个方法可以看出来，Handler类消息分发的过程如下：

1. 如果传递的Message本身就有callback，那么直接使用Message对象的callback方法；
2. 如果Handler类的成员变量mCallback存在，那么首先执行这个mCallback回调；
3. 如果mCallback的回调返回true，那么表示消息已经成功处理；直接结束。
4. 如果mCallback的回调返回false，那么表示消息没有处理完毕，会继续使用Handler类的handleMessage方法处理消息。

那么，ActivityThread中的Handler类H是如何实现的呢？H的部分源码如下：

```

public void handleMessage(Message msg) {
    if (DEBUG_MESSAGES) Slog.v(TAG, ">> handling: " + codeToString(msg.what));
    switch (msg.what) {
        case LAUNCH_ACTIVITY: {
            Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "activityStart");
            ActivityClientRecord r = (ActivityClientRecord)msg.obj;

            r.packageInfo = getPackageInfoNoCheck(
                r.activityInfo.applicationInfo, r.compatInfo);
            handleLaunchActivity(r, null);
            Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
        } break;
        case RELAUNCH_ACTIVITY: {
            Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "activityRestart");
            ActivityClientRecord r = (ActivityClientRecord)msg.obj;
            handleRelaunchActivity(r);
        }
    }
}

```

```
Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
```

```
// 以下略
}
}
```

可以看到H类仅仅重载了handleMessage方法；通过dispatchMessage的消息分发过程得知，我们可以拦截这一过程：把这个H类的mCallback替换为我们的自定义实现，这样dispatchMessage就会首先使用这个自定义的mCallback，然后看情况使用H重载的handleMessage。

这个Handler.Callback是一个接口，我们可以使用动态代理或者普通代理完成Hook，这里我们使用普通的静态代理方式；创建一个自定义的Callback类：

```
/* package */ class ActivityThreadHandlerCallback implements Handler.Callback {

    Handler mBase;

    public ActivityThreadHandlerCallback(Handler base) {
        mBase = base;
    }

    @Override
    public boolean handleMessage(Message msg) {

        switch (msg.what) {
            // ActivityThread里面 "LAUNCH_ACTIVITY" 这个字段的值是100
            // 本来使用反射的方式获取最好，这里为了简便直接使用硬编码
            case 100:
                handleLaunchActivity(msg);
                break;
        }

        mBase.handleMessage(msg);
        return true;
    }
}
```

```

private void handleLaunchActivity(Message msg) {
    // 这里简单起见,直接取出TargetActivity;

    Object obj = msg.obj;
    // 根据源码:
    // 这个对象是 ActivityClientRecord 类型
    // 我们修改它的intent字段为我们原来保存的即可.

    /* switch (msg.what) {
        /     case LAUNCH_ACTIVITY: {
        /         Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER,
        "activityStart");
        /         final ActivityClientRecord r = (ActivityClientRecord) msg.obj;
        /
        /         r.packageInfo = getPackageInfoNoCheck(
        /             r.activityInfo.applicationInfo, r.compatInfo);
        /         handleLaunchActivity(r, null);
        */
    }

    try {
        // 把替身恢复成真身
        Field intent = obj.getClass().getDeclaredField("intent");
        intent.setAccessible(true);
        Intent raw = (Intent) intent.get(obj);

        Intent target = raw.getParcelableExtra(HookHelper.EXTRA_TARGET_INTENT);
        raw.setComponent(target.getComponent());

    } catch (NoSuchFieldException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    }
}
}

```

这个Callback类的使命很简单：把替身StubActivity恢复成真身TargetActivity；有了这个自定义的Callback之后我们需要把ActivityThread里面处理消息的Handler类H的的

mCallback修改为自定义callback类的对象：

```
// 先获取到当前的ActivityThread对象
Class<?> activityThreadClass = Class.forName("android.app.ActivityThread");
Field currentActivityThreadField = activityThreadClass.getDeclaredField("sCurrentActivityThread");
currentActivityThreadField.setAccessible(true);
Object currentActivityThread = currentActivityThreadField.get(null);

// 由于ActivityThread一个进程只有一个,我们获取这个对象的mH
Field mHField = activityThreadClass.getDeclaredField("mH");
mHField.setAccessible(true);
Handler mH = (Handler) mHField.get(currentActivityThread);

// 设置它的回调, 根据源码:
// 我们自己给他设置一个回调,就会替代之前的回调;

//     public void dispatchMessage(Message msg) {
//         if (msg.callback != null) {
//             handleCallback(msg);
//         } else {
//             if (mCallback != null) {
//                 if (mCallback.handleMessage(msg)) {
//                     return;
//                 }
//             }
//             handleMessage(msg);
//         }
//     }

Field mCallBackField = Handler.class.getDeclaredField("mCallback");
mCallBackField.setAccessible(true);

mCallBackField.set(mH, new ActivityThreadHandlerCallback(mH));
```

到这里，我们已经成功地绕过AMS，完成了『启动没有在AndroidManifest.xml中显式声明的Activity』的过程；瞒天过海，这种玩弄系统与股掌之中的快感你们能体会到吗？

## 僵尸or活人？——能正确收到生命周期回调吗

虽然我们完成了『启动没有在AndroidManifest.xml中显式声明的Activity』，但是启动的TargetActivity是否有自己的生命周期呢，我们还需要额外的处理过程吗？

实际上TargetActivity已经是一个有血有肉的Activity了：它具有自己正常的生命周期；可以运行Demo代码验证一下。

这个过程是如何完成的呢？我们以onDestroy为例简要分析一下：

从Activity的finish方法开始跟踪，最终会通过ActivityManagerNative到AMS然后接着通过ApplicationThread到ActivityThread，然后通过H转发消息到ActivityThread的handleDestroyActivity，接着这个方法把任务交给performDestroyActivity完成。

在真正分析这个方法之前，需要说明一点的是：不知读者是否感受得到，App进程与AMS交互几乎都是这么一种模式，几个角色 ActivityManagerNative, ApplicationThread, ActivityThread以及Handler类H分工明确，读者可以按照这几个角色的功能分析AMS的任何调用过程，屡试不爽；这也是我的初衷——希望分析插件框架的过程中能帮助深入理解Android Framework。

好了继续分析performDestroyActivity，关键代码如下：

```
ActivityClientRecord r = mActivities.get(token);  
  
// ...略  
  
mInstrumentation.callActivityOnDestroy(r.activity);
```

这里通过mActivities拿到了一个ActivityClientRecord，然后直接把这个record里面的Activity交给Instrument类完成了onDestroy的调用。

在我们这个demo的场景下，r.activity是TargetActivity还是StubActivity？按理说，由于我们欺骗了AMS，AMS应该只知道StubActivity的存在，它压根儿就不知道TargetActivity是什么，为什么它能正确完成对TargetActivity生命周期的回调呢？

一切的秘密在token里面。AMS与ActivityThread之间对于Activity的生命周期的交互，并没有直接使用Activity对象进行交互，而是使用一个token来标识，这个token是binder对象，因此可以方便地跨进程传递。Activity里面有一个成员变量mToken代表的就是它，token可以唯一地标识一个Activity对象，它在Activity的attach方法里面初始化；

在AMS处理Activity的任务栈的时候，使用这个token标记Activity，因此在我们的demo里面，AMS进程里面的token对应的是StubActivity，也就是AMS还在傻乎乎地操作 StubActivity（关于这一点，你可以dump出任务栈的信息，可以观察到dump出的确实是 StubActivity）。但是在我们App进程里面，token对应的却是TargetActivity！因此，在 ActivityThread执行回调的时候，能正确地回调到TargetActivity相应的方法。

为什么App进程里面，token对应的是TargetActivity呢？

回到代码，ActivityClientRecord是在mActivities里面取出来的，确实是根据token取；那么这个token是什么时候添加进去的呢？我们看performLaunchActivity就完全明白了：它通过classloader加载了TargetActivity，然后完成一切操作之后把这个activity添加进了 mActivities！另外，在这个方法里面我们还能看到对Activity.attach方法的调用，它传递给了新创建的Activity一个token对象，而这个token是在ActivityClientRecord构造函数里面初始化的。

至此我们已经可以确认，通过这种方式启动的Activity有它自己完整而独立的生命周期！

## 小节

本文讲述了『启动一个并没有在AndroidManifest.xml中显示声明的Activity』的解决办法，我们成功地绕过了Android的这个限制，这个是插件Activity管理技术的基础；但是要做到启动一个插件Activity问题远没有这么简单。

首先，在Android中，Activity有不同的启动模式；我们声明了一个替身StubActivity，肯定没有满足所有的要求；因此，我们需要在AndroidManifest.xml中声明一系列的有不同 launchMode的Activity，还需要完成替身与真正Activity launchMode的匹配过程；这样才能完成启动各种类型Activity的需求，关于这一点，在 DroidPlugin 的 com.morgoo.droidplugin.stub包下面可以找到。

另外，每启动一个插件的Activity都需要一个StubActivity，但是AndroidManifest.xml中肯定只能声明有限个，如果一直startActivity而不finish的话，那么理论上就需要无限个 StubActivity；这个问题该如何解决呢？事实上，这个问题在技术上没有好的解决办法。但是，如果你的App startActivity了十几次，而没有finish任何一个Activity，这样在Activity的回退栈里面有十几个Activity，用户难道按back十几次回到主页吗？有这种需求说明你的产品设计有问题；一个App一级页面，二级页面..到五六级的页面已经影响体验了，所以，每种LaunchMode声明十个StubActivity绝对能满足需求了。

最后，在本文所述例子中，TargetActivity与StubActivity存在于同一个Apk，因此系统的

ClassLoader能够成功加载并创建TargetActivity的实例。但是在实际的插件系统中，要启动的目标Activity肯定存在于一个单独的文件中，系统默认的ClassLoader无法加载插件中的Activity类——系统压根儿就不知道要加载的插件在哪，谈何加载？因此还有一个很重要的问题需要处理：

我们要完成插件系统中类的加载，这可以通过自定义ClassLoader实现。解决了『启动没有在AndroidManifest.xml中显式声明的，并且存在于外部文件中的Activity』的问题，插件系统对于Activity的管理才算得上是一个完全体。篇幅所限，欲知后事如何，请听下回分解！

喜欢就点个zan吧～持续更新，请关注github项目 [understand-plugin-framework](#)和我的博客！

## 专栏作者简介（[点击 → 加入专栏作者](#)）

weishu : Android开发，Haskell爬坑ing

微信扫一扫 支付



向田维术 [\*\*术] 赞助

支持我写出更好的文章 ^\_^

¥1.00

打赏支持作者写出更多好文章，谢谢！

# 安卓应用频道

专注分享安卓应用相关内容



微信号：AndroidPD



长按识别二维码关注

---

伯乐在线 旗下微信公众号

商务合作QQ：2302462408

---