VirtualApp原理解析(4)--双开应用启动过程

原创

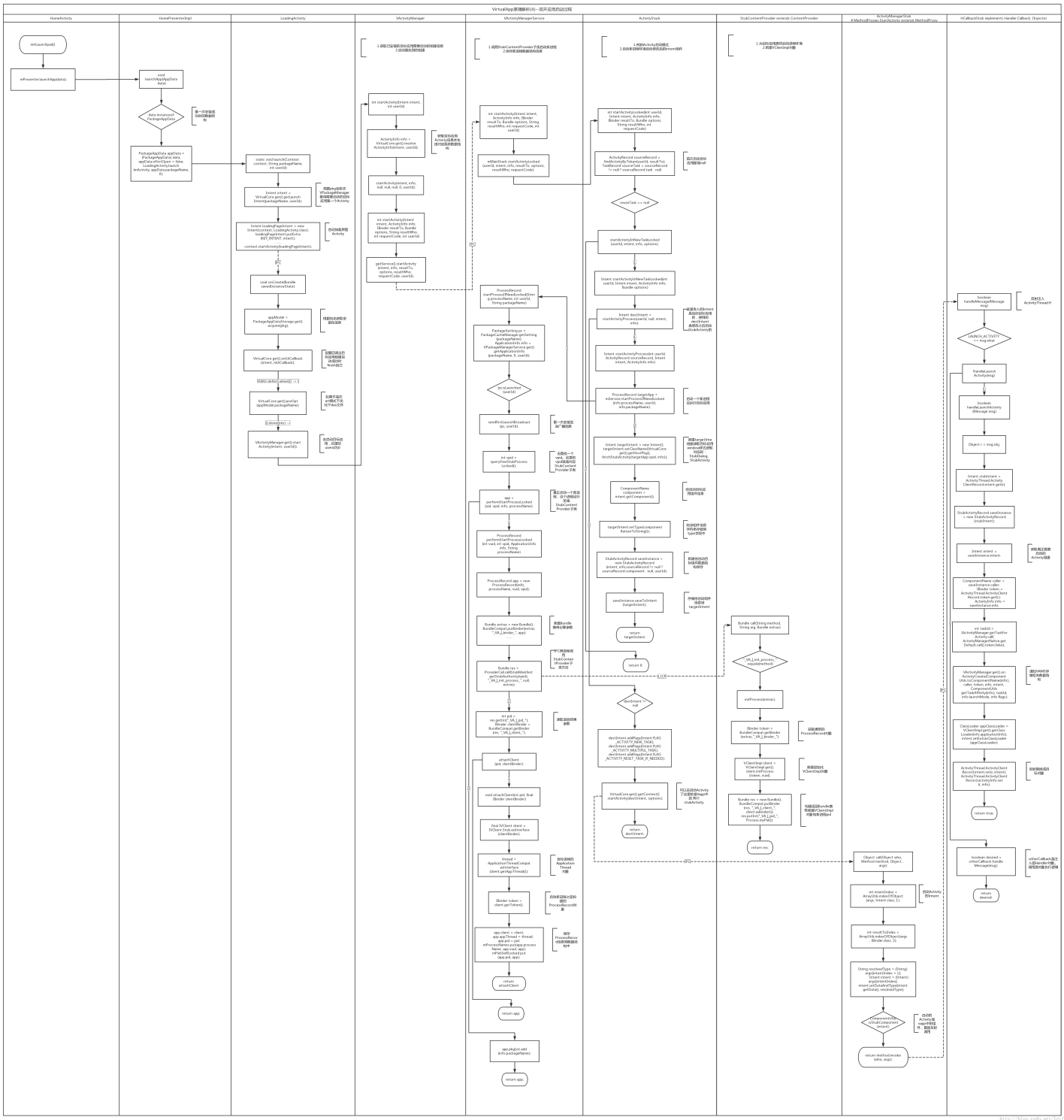
leif_123 于 2017-06-14 14:08:54 发布

分类专栏： 开源框架源码分析 文章标签： VirtualApp 双开应用启动 Activity启动

本文链接：https://blog.csdn.net/leif_/article/details/73223851

VirtualApp原理解析(4)–双开应用启动过程

双开应用启动是从HomeActivity的列表点击事件中开始的。用户点击已安装的双开应用则启动这个双开应用。整个过程流程图如下：



1.用户在HomeActivity中点击对应双开应用启动流程由此开始。

调用mPresenter.launchApp(data)执行启动流程对应实现在HomePresenterImpl中。

在lunchApp()中首先判断传入参数类型，这里要注意：如果该双开应用是第一次安装到virtualapp对应应该是PackageAppData，如果是多次安装多版本则是MultiplePackageAppData。

这里只考虑首次安装则调用LoadingActivity.lunch()静态方法，在该方法中首先调用VirtualCore.get().getLaunchIntent(packageName, userId);根据传入包名去VPMS中查询对应需要启动的第一个Activity的Intent。接着传递获取的intent等参数真正启动LoadingActivity。

2.这里的LoadingActivity真是在启动双开应用时的中间过度界面，具体就是显示一个loading状态。

在onCreate()中先获取传入参数，接着根据传入包名调用PackageAppDataStorage.get().acquire(pkg)获取一个PackageAppData这个东西是根据安装时解析到的数据分装。其实就是一个app的安装信息。接着获取我们传入的需要启动的目标Activity的intent。接着判断如果在非art模式下VirtualCore.get().preOpt(appModel.packageName)优化dex文件。

开始调用VActivityManager.get().startActivity(intent, userId))，执行启动目标组件。

3.在VActivityManager.get().startActivity()中开始处理具体的启动Activity逻辑。

首先调用VirtualCore.get().resolveActivityInfo(intent, userId)返回一个系统API的数据结构ActivityInfo，封装待启动Activity信息。这里的数据获取是根据之前双开应用安装时的信息进行查询分装得到。同是传入的intent已经设置对应的ComponentName。intent已经具备启动能力。

调用startActivity(intent, info, null, null, null, 0, userId);该函数内部通过IPC访问VAMS同名函数。

4.已经通过IPC调用到了后台服务进程中的VAMS。

- 在VAMS中被调用的startActivity()内部有调用到ActivityStack.startActivityLocked()，真正的处理逻辑在这里。

5.在ActivityStack.startActivityLocked()中：

调用findActivityByToken(userId, resultTo)根据token去已经启动的缓存中获取对应的ActivityRecord，因为是第一次启动这里sourceRecord、 sourceTask都是null。

接着根据目标intent设置的flag处理启动模式等。不清楚这里的逻辑可以参考Android源码中的Activity启动处理逻辑。

因为是首次启动reuseTask是null所以走的是startActivityInNewTaskLocked(userId, intent, info, options);函数。 重点来了，就是在这个函数中，首先调用startActivityProcess(userId, null, intent, info);函数。该函数负责为新启动的Activity创建一个新进程。具体如下：

首先调用VAMS.startProcessIfNeedLocked(info.processName, userId, info.packageName)，在该函数中获取对应包名的PackageSetting和ApplicationInfo对象，这两个对象在双开应用安装时创建病保存。接着判断目标进程是否已经创建过，如果存在返回对应的ProcessRecord。首次启动不存在继续调用queryFreeStubProcessLocked();获取现在还未占用的vpid。起内部是一个循环遍历所有ProcessRecord查找还未使用的vpid。接着调用performStartProcessLocked(uid, vpid, info, processName)，在该函数中新建ProcessRecord对象，将传入参数分装到Bundle，接着调用ProviderCall.call()传入需要启动的目标StubContentProvider的url，该url根据vpid获取。启动成功会返回一个Bundle分装新进程pid保存。接着attachClient()设置远程StubContentProvider死亡回调、分装数据到ProcessRecord等。

经过以上步骤已经启动了一个新进程并且得到了Processrecord，接着需要处理传入的目标组件Intent。首先新建一个targetIntent，设置targetIntent.setClassName(VirtualCore.get().getHostPkg(), fetchStubActivity(targetApp.vpid, info))。这里调用到fetchStubActivity(targetApp.vpid, info)函数，该函数根据传入参数判断目标启动Activity组件是否是对话框模式返回对应Activity类名。这里根据Activity样式需要启动VirtualApp中的不同Activity：StubActivity和StubDialog。

接下来保存需要启动的目标双开应用的ComponentName信息到targetIntent的type字段中。

新建一个StubActivityRecord保存信息和生成的targetIntent之后返回这个targetIntent。这里的targetIntent启动目标是VirtualApp中的申明的组件和目标双开应用无关，待启动的双开应用组件信息都保存在其type字段中。

走完startActivityProcess()得到了之前生成的targetIntent，现在新进程有了为targetIntent添加flag等可以去启动这个targetIntent了。调用VirtualCore.get().getContext().startActivity(destIntent, options);

简单总结：

到这里我们需要处理的逻辑已经完成，接下来启动的任务交给系统处理，大致有如下步骤：

一、根据包名获取需要启动双开应用的Activity组件信息保存。

二、判断待启动组件是否需要启动新进程，启动新进程后构建位于新进程的Activity组件信息。

三、启动这个Activity，到这里启动的Activity是VirtualApp中的和双开应用无关。

> **注意：到这里需要重点关注以下问题：**
> 一、首先是VirtualApp的进程启动的管理：在VirtualApp中申明了一些列的StubContentProvider及其子类，该类继承子ContentProvider。这一系列的StubContentProvider由于声明在各自单独进程中，每当启动双开应用组件需要新进程时，其实就是启动当前一个还未启动的StubContentProvider子类，由于其声明在单独进程，这个进程就是双开应用组件运行的进程环境。
> 二、执行到上述步骤有一个循环遍历查找未使用的vpid其实就是从0开始遍历，根据已使用的ProcessRecord对应vpid判断可使用的。
> 三、如何保证启动的组建运行在我们之前启动的目标进程中呢？通过查看代码就是通过ProcessRecord中的vpid查找到对应的Activity组建。相同后缀名的StubContentProvider子类与StubActivity和StubDialog其子类通过vpid一一对应并且声明在同一进程下运行。这样就保证了启动的目标组件运行在我们的新进程中。

6.调用了startActivity()系统处理这个启动请求，大致如：AMS起收到启动请求->处理目标Activity启动模式->暂停当前Activity->启动目标Activity->IPC调用ActivityThread函数将启动消息通过其内部Handler发送一个LAUNCH_ACTIVITY消息。到这里重点来了。

因为在之前VirtualApp刚刚启动是注入了该进程所在ActivityThread中的Handler的Handler.Callback，所以当系统启动一个Activity时在ActivityThread中通过Handler发送LAUNCH_ACTIVITY消息最终会走到VirtualApp中的HCallbackStub.handleMessage()。这里会调用我们自己的handleLaunchActivity()。

首先反射获取位于ActivityThread.ActivityClientRecord中的intent，这个intent可以看做就是我们启动Activity时传入的targetIntent。新建一个StubActivityRecord解析这个intent内的附件信息即我们真正要启动的双开应用的Activity的intent信息。

接着处理一些状态信息反射获取待启动Activity对应的taskId，然后调用到VActivityManager.get().onActivityCreate()通知后台服务进程中的VAMS保存该Activity相关信息。

最后反射设置 ActivityThread.ActivityClientRecord中的intent为双开应用组建的intent，即我们真正需要启动的Activity的intent。同时设置其内部的activityInfo等。

好了至此我们将关键的ActivityThread中的Handler处理的逻辑做了修改替换了待启动的目标Activity的信息为双开应用中的组件信息。做完以上操作接下来就是将修改之后的msg信息交会原本的ActivityThread中的Handler去执行真正的启动逻辑。

至此，是真正开始执行双开应用的Activity。

简单总结：

一、反射注入ActivityThread的Handler对象，该对象负责处理客户端进程所有组件启动工作。我们在lunchActivity的时候将之前步骤保存的双开应用intent取出并反射修改。

二、系统在启动Activity时获取的是我们修改后的双开应用的intent，所以启动的是双开应用。

三、通过以上步骤：首先启动VirtualApp内部组件->系统处理完逻辑->给客户端ActivityThread做进一步处理，修改intent为目标双开应用组件->系统得到intent信息启动对应组件->此时组件就是目标双开应用的组件。可以看出：双开应用<——>VirtualApp<——>Android系统。VirtualApp起了一个中间层的作用“欺上瞒下”。之前hook的一些关键函数比如ActivityManager、PackageManager等都是为目标双开应用准备，当双开应用调用到其中系统函数会做响应包括包名、权限等修改让系统认为这是VirtualApp的调用。双开应用始终运行在VirtualApp内部。

说明：

一、VirtualApp的实现需要非常熟悉Android Framework组件，包括：apk安装流程、四大组件启动流程、反射、动态代理等的深入理解。

二、以上关于VirtualApp的分析到此结束，本人水平有限分析学习的视角、水平一般。文章也只是自己的一些见解存在错误尽情见谅。也感谢开源作者的无私奉献。