虚拟多开的技术原理

我们知道一个 Android App 的正常启动、运行需要几样基本的要素，即独立的进程环境、独立的运行时上下文环境、各个组件的生命周期管理以及独立的运行时文件系统。为了实现应用的多开，毫无疑问宿主应用也必须具备创建这四种核心环境的功能才能满足插件应用的正常运行，好比创建了一个中间桥梁打通操作系统与插件应用之间的交互，这里引入一句计算机领域非常有名的一句话：

“All problems in computer science can be solved by another level of indirection”。

——*David* Wheeler

翻译过来的意思是计算机科学中的大多数问题都可以通过引入一层中间层解决。这句话几乎概括了计算机系统体系结构的设计要点，很多体系结构基本都是从上到下按照严格的层次结构设计的，比如操作系统、网络协议栈、虚拟化以及各种应用层的中间件等。甚至很多硬件结构都是按照这种层次的理念组织和设计的。好了言归正传，回归到我们的主题上来，图 2 我们给出了应用虚拟多开的一个基本架构原型。

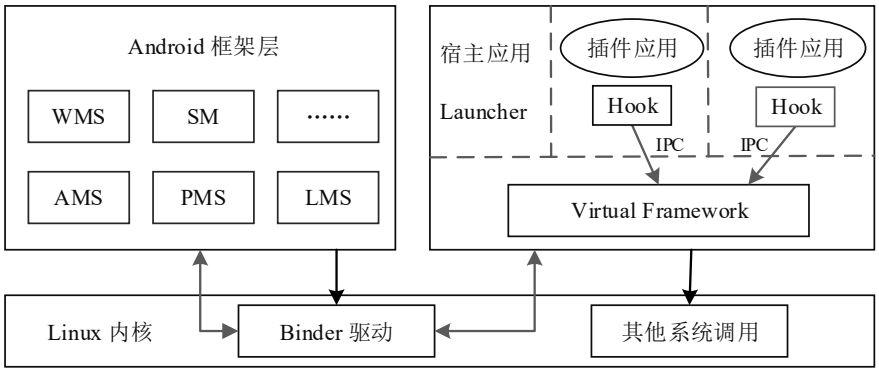

图 2.2 Android 应用虚拟化技术的原型架构

从图中可以看出，宿主应用主要由三个模块组成，分别是 Launcher、Hook、Virtual Framework。我们将以一个插件 APK 是如何被启动运行起来为主线，讲解宿主应用各个功能模块的作用；

首先，Launcher 模块主要负责虚拟安装、启动插件应用，功能类似于原生 Android 系统的桌面应用，虚拟安装的过程和应用在原生 Android 系统上安装的过程类似，起先对插件应用的 APK 文件进行解析，得到插件应用的 Mainifest.xml 文件中注册的组件、权限、资源文件等。当用户点击要启动的插件应用时，Launcher 模块起先启动一个在宿主应用的 Mainifest.xml 中预注册 stub 组件，我们以开源框架 VirtualApp 为例，stub 组件的注册形式如下所示：

```
<provider
    android:name="StubContentProvider$C1"
    android:authorities="${applicationId}.virtual_stub_20"
    android:exported="false"
    android:process=":p1" /> 系统会启动一个新进程来运行该组件
```

该 stub 组件使用了 android:process=":p1"来标识，表明组件在启动组件之前会向系统服务 System Server 发送进程孵化请求，请求的最终处理是由 Zygote 进程进行处理，Zygote 进程会 fork 出一个完整的带有虚拟机运行实例的进程返回。进程启动之后会执行 ActivityThread 的 main 函数进行消息循环和处理阶段，然后 AMS 再通过 IPC 调用执行 bindApplication 操作创建和执行 Application 的初始化（我们启动带有 android:process=":p1"标签的组件时，都会创建一个新的进程并再一次执行 Application 的初始化），注意这里的 Application 是宿主应用定义的 Application，并不是插件应用的 Application，不具有要加载运行插件应用的执行上下文环境，但是该 Application 是宿主应用定义的，在执行 attachBaseContext 的初始化过程中进行判断，如果当前进程的标识是以":px"标识结尾的，表明当前是新启动的进程是要用来运行插件应用的进程，进一步对新启动的进程注入 Hook 代码（包括 Dex 层对系统服务调用的 hook, Native 层对 libc.so 中关键系统调用的 hook），Hook 模块是之后拦截 APK 与系统服务的交互实现虚拟化的关键。故事讲到这里，我们发现，此时已经启动了用于插件应用运行的独立进程。此时还不具备运行插件应用的上下文环境，那么该怎么办呢？

此时，插件应用的 MainActivity 还没有启动起来，系统会向 AMS 发起启动 Intent 请求，我们知道所有和系统服务交互的 Binder 调用都被 hook 了，hook 函数拦截启动请求，替换 Intent 中的启动目标组件 MainActivity 为宿主预注册的 StubActivity，然后再将篡改了的 Intent 转发给真正的 AMS，AMS 接受到请求之后进行一系列的初始化、任务栈管理、ActivityRecord 创建、窗体 token 的生成等操作，最后通过 Binder 代理对象通知 ActivityThread 创建执行真正的 MainActivity，这里是 MainActivity 的首次启动，因此会再一次触发 hook 代码逻辑，hook 代码会判断是否已经创建了插件应用的上下文对象 Application，如果没有创建，则通过系统提供的 LoadedApk 对象创建插件应用的 Application 上下文对象，构造子程序独立的类加载器。然后在将 MainActivity 的执行上下文 Context 与新创建的 Application 进行绑定和 attach，这样子程序插件应用就具备了独立的进程和独立的执行上下文。然后再通过新创建的类加载器创建和加载 MainActivity，并且回调其 onCreate()事件函数，到这里插件应用就已经具备独立上下文环境了。

那么到这里，组件是启动起来了，与之配套的相关资源也是可以加载了，但重要的是生命周期怎么维护和管理的呢？没有生命的组件是僵死的，不具备各项事件的响应处理能力。因此模拟生命周期的管理尤为重要。在运行的过程中每一个插件应用的组件都有一个由宿主预注册好的 stub 组件与之对应，插件应用的组件每个生命周期状态都会通过 IPC 调用告知系统服务自己的状态，由于系统服务不认识插件组件的身份（因为没有真正安装，系统不知道有该组件的存在），所以会通过中间层的 hook 替换为与之对应的 stub 组件身份和系统交互。再者有些系统消息（比如广播事件、回退事件等）需要告知插件组件，系统还是以 stub

的身份发出 IPC 消息，中间层的 hook 会先收到消息，然后将消息传递给与之对应的插件组件，由插件组件执行最终的事件响应或触发回调函数。可以简单的理解为让一个原本没有生命周期的组件依附在一个具有正常生命周期的组件上运行，有点借尸还魂的味道。具体的细节非常的复杂，有兴趣的可以参照 Android 系统源码中对各个组件生命周期的管理模块，看懂了这些代码，相信 Andriod Framework 的任何模块都难不到你。

最后一点是文件系统的虚拟化，插件应用要在宿主应用环境中正常运行那么必须对其运行时文件系统做处理，举个简单的例子假设有个 APP 的包名为 com.example.demo，如果该 APP 运行在真实的 Android 操作系统上时，系统会分配数据目录给该应用，一般以/data/data/com.example.demo/files/为文件目录前缀，应用可在代码中根据该目录结构特征读写文件。但是，当其运行在宿主应用（假设宿主的私有文件目录/data/data/com.host.demo/）构建的环境中时，由于 Android 应用级别沙箱的限制，如果插件应用还是按照安装在原生系统上的规则读写文件，那么很可能会触发权限访问异常或者读写不到目标文件，造成插件应用不能正常运行。因此，多数宿主引擎实现在 Native 层通过 IO 重定向技术实现应用运行时文件系统的虚拟化，具体的讲主要是通过在所有和文件系统操作相关的系统调用处（vfork()、_open()、mkdir()等系统调用函数）实现拦截和挂钩，这些函数都封装在 Libc.so 动态库中。

已有对 native 层 so 的 Hook 技术分为两种，一种是 Hook ELF 全局偏移表（GOT 表），然而 Hook GOT 表功能是很有限的，只能 Hook 外部导入函数，无法 Hook 局部函数。另外一种 Hook 方式为 Inline Hook 技术，其原理就是通过替换需要 Hook 的原函数起始地址处的指令为跳转指令，使得调用原函数变成跳转到自定义的钩子函数，通常在执行钩子程序之前还会保留原函数的调用接口和一些寄存器参数。基本的过程如图 3 所示。

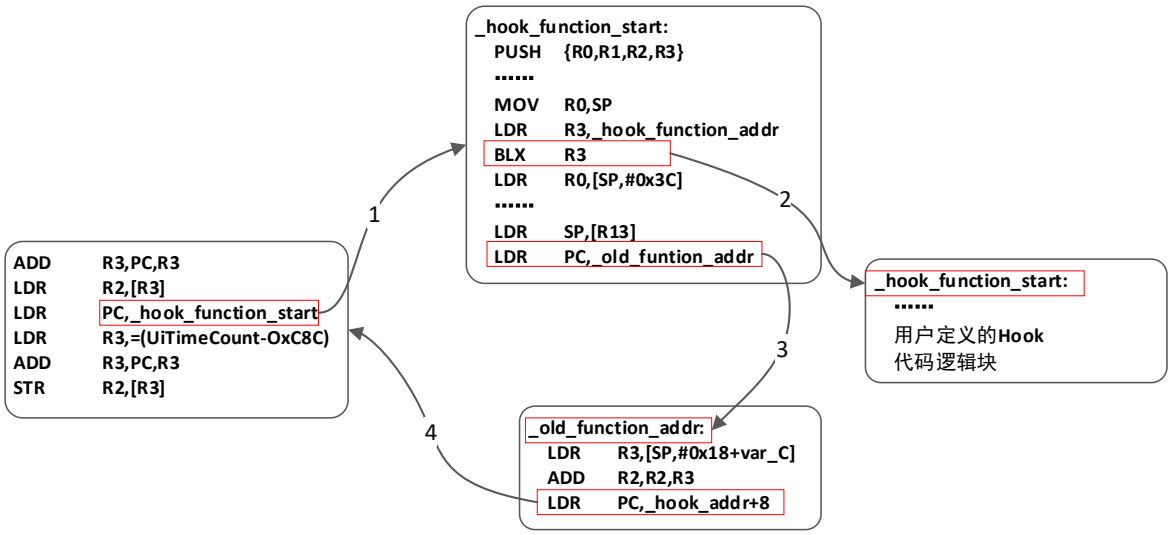

图 3 Inline Hook 技术实现文件 I/O 重定向的基本原理

与 Hook 表 GOT 相比，内联 Hook 具有更好的通用性，几乎可以对任何本地函数进行挂钩。虚拟化应用多开技术就是通过拦截文件操作相关的底层函数，在其陷入内核系统调用之前注入钩子程序，钩子程序通过更改相应文件描述中的路径描述为重定向之后的文件目录，以此来实现文件 I/O 重定向操作。宿主利用文件 I/O 重定向技术将子程序当前访问的运行时文件目录重定向到

/data/data/com.host.demo /virtual/0/data/data/com.example.demo/下，每个插件的运行目录都被重定向这样一个单独的目录下进行读写操作。实现了非侵入式、无感知的应用程序文件系统的虚拟化。