基于动态代理 Android 插件化研究与实现

Research and Implementation of Android Proxy Based on Dynamic Agent

陈先跃　王大全　（杭州电子科技大学计算机图像研究所，浙江 杭州 310018）

摘要：随着 Android 项目日益复杂、项目开发困难、用户体验差等问题不断加深，Android 插件化技术的研究迅速发展。目前成熟的插件化[1]方案采用应用安装的方式，但是它存在数据交换速度慢的缺点。而非安装形式的插件化方案存在组件没有生命的问题。针对上述问题，采用非安装形式的插件化方案，通过在应用的配置文件中预设组件，在目标组件进行验证时用动态代理方式替换为预设组件通过验证，最后通过替换为目标组件的方式对 Android 系统底层的安全机制进行欺骗，解决插件运行时出现的组件没有生命周期问题，提高数据交换速度。

关键词：插件化，Android，动态代理

Abstract:This paper uses the non-installable plug-in scheme,by pre-setting the component in the application configuration file,replacing the default component with the dynamic proxy by the verification of the target component, and finally by replacing the target component.The way the security system on the bottom of the Android system to deceive,to solve the plug-in runtime components do not have life-cycle problems,improve data exchange speed.

Keywords:plug-in,Android,dynamic proxy

Android 插件化技术的出现，究其原因是 Android 项目越来越巨大。而这种应用巨大化带来了方法数越界[2]、编译速度缓慢、代码耦合度增加、项目管理困难等一系列开发问题，和应用占用空间过大、下载流量过多、安装更新速度太慢等一系列用户体验问题。在目前热门的 Html5 技术支持的快速的开发环境中，插件化技术以其操作性能更优秀和用户体验更良好，让 Native App 保有一席之地。目前 Android 插件化包括两种形式，插件安装方式和插件免安装方式。插件安装方式是相当于将应用安装到系统中，然后通过包名去加载插件到 Context（应用上下文）中，可以方便地进行插件资源的加载，但是由于插件之间的 ClassLoader（类加载器）并不相同，数据只能通过序列化和反序列化的方式进行交换，造成了数据传递效率差的问题。插件免安装方式则是将应用和宿主之间使用同一个 ClassLoader 加载，增加了数据传输速度，但是由于 Context 并不相同以及 Android 系统底层的安全机制，应用会出现未注册或没有生命周期等情况。

1　动态代理的基本原理

动态代理[3]是一种面向接口的技术，它引入让通常由 stub 提供的信息在运行时可以被系统发现。在 Java 的开发中提供了一个动态代理类，在 Java.lang.reflect 包下，其中定义了一个接口 InvocationHandler，它是整个动态代理的核心，对被代理接口上任意方法的调用都将被传递给代理对象 InvocationHandler 接口反射方法 Object:invoke（Object obj;Method method,Object[] args）。Obj 是被代理的类，method 是被代理的方法，args 为该方法的参数数组。通过实现 InvocationHandler 的子类就是一个代理对象的最终执行类。Proxy 就是 Java 中的动态代理类，通过 newProxyInstance（ClassLoader loader, Class<?>[] interfaces, InvocationHandler h）可以为一个或者多个类动态地生成实现类，其中，loader 就是类加载器，interfaces 是真实类的全部接口函数。在实现过程中，Proxy 可以被看作是分发器，把每个接口上的调用分发到它们的实现类中，如图 1。当代理对象被类型转换为成其中某个的接口时，那么它就是该接口的一个实例。这种机制使开发人员只要指定一组接口及委托类对象就动态地获得代理类，并且只需要写一次处理逻辑和 Proxy 中逻辑，减少了代码的编写，修改时只需改动一次。

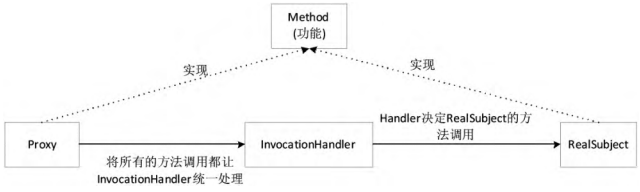

图 1　动态代理原理图

通过创建代理对象，然后把原始对象进行替换为自己的代理对象，就可以在创建代理对象中修改参数，替换返回值，实现对 Java API 中的代码功能的替换。Android API 同样是使用 Java 语言编写，因此动态代理能够进行 Android 中的代码方法进行替换。

2　动态代理方式实现插件化

2.1 资源加载和代码动态加载

在 Android 系统中资源的加载是通过 Resources 类中 AssetManager 中的隐式方法 addAssetPath 设置加载路径。因此可以通过反射 AssetManager 修改 addAssertPath 完成资源的加载。其代码如下：

```
AssetManager assetManager=AssetManager.class.newInstance ();
Method addAssetPath = AssetManager.class.getMethod ("addAssetPath",tring.class);
addAssetPath.invoke(assetManager, path);
```

代码动态加载[4-5]则是通过 DexClassLoader 类，DexClassLoader pluginClassLoader = new DexClassLoader (dexPath, optimizedDirectory, libraryPath, parentClassLoader)；然后就可以通过 loadclass 加载某个类。

2.2 动态代理组件生命周期获取

免安装插件没有生命周期的主要原因是在 Android 系统中存在沙箱机制，使得必须要在 AMS 中进行验证是否在配置文件 AndroidManifest 中注册，若没由注册则无法启动组件。因此本文提出在配置文件预先注册组件，当目标组件被调用时，在验证之前使用动态代理的方式，将目标组件保存并替换为预设的组件，在验证完成后，将保存过的目标组件重新换回主线程，使其避过验证，获得生命周期。

在 Android 开发中大部分应用是由 Activity 组件构成，少量

应用使用到 Service 组件，极少的应用使用到 Content Provider 和 Broadcast Receiver，因此本文 Activity 组件为例实现插件的生命周期。Activity 的启动简要包括三个部分：第一个部分在 App 进程里调用 startActivity()函数进行启动；第二个部分通过 Binder 机制实现 App 进行和 AMS 进程直接的通信，调用 AMS 中的 startActivity，进行一系列的验证后调用，在 App 线程中新建 ActivityThread，并不断与 AMS 通信，完成创建和管理，最后在 App 进程中创建。其动态代理插件化的步骤如下：

1）预设 Activity。在 Manifest 中预设多种类型的 Activity 为后面的不同目标 Activity 提供不同的验证实例。

2）调用 startActivity。用户启动目标 Activity，函数调用正式开始。

3）将目标 Activity 替换为预设 Activity。通过动态代理 ActivityManagerNative，修改 startActivity 方法，先查找目标 Activity 的匹配的预设 Activity，保存目标 Activity 的信息，对目标 Activity 进行替换。

4）预设 Activity 经过与 AMS 的验证，通过后，创建进程。

5）拦截 Callback，换回目标 Activity。在 App 进程里它是 Server 端，Binder 线程池在 Server 端接受 Binder 远程调用，它通过 Handler 把消息转发给 App 的主线程；可以在这个 Handler 里面将换回 TargetActivity。

通过以上步骤目标 Activity 获得生命周期，正常运行。

3 实验结果及分析

本文通过主要对动态代理免安装插件化方式的 App、安装方式的插件化 App 和一体式 App 进行测试比较。首先针对设计测试环境，为保证测试的准确性，文本为三种方式的 App 设计相同的 Activity，其中包含 Button、Text、ImageView、EditText 等一系列控件。采用编写 MonkeyRunner 的测试脚本，对三种方式进行性能测试对比。

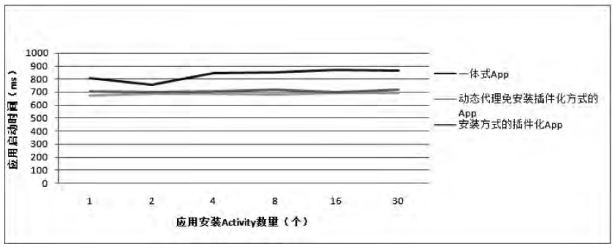

图 2 应用启动分析图

图 2 是应用启动时间与 Activity 数量的关系，可以看两个插件化方式的 App 的启动时间明显比较快。

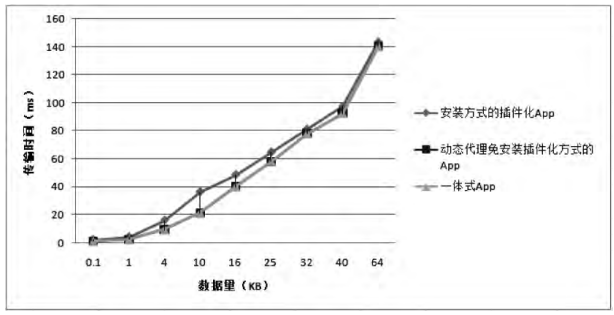

图 3 应用传输速度分析图

图 3 是数据传输时间和数据量之间的关系，安装方式的插件化 App 在小数据量时，明显慢于其他两种，而一体式的速度略优于插件式。

参考文献

[1]张茗越.Android 插件化与云服务相结合的软件开发方法[D].哈尔滨：哈尔滨理工大学，2016

[2]任玉刚.Android 开发艺术探索[M].北京：电子工业出版社，2015

[3]Sheinis J, Baldwin M, Sherkin A. Application internationalization using dynamic proxies: US 20050050548 A1[P]. 2005

[4]钱宇虹.Java 动态加载与插件开发研究[J].中小企业管理与科技，2015(30)：156–157

[5]姚伟涛.APK(Android Package) plug–in management method: CN, CN 102915247 A[P]. 2013

[收稿日期：2017.2.22]

（上接第 98 页）

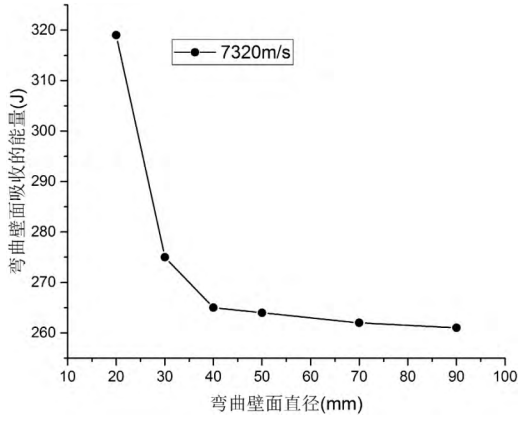

图 7 速度为 7320m/s 时的能量吸收曲线

的能量具有相同的变化趋势，即在同一速度下，随着弯曲壁面直径不断增大，壁面所吸收的能量在不断地减小。而根据孔洞的变化规律，由于在同一撞击速度下，撞击所形成的孔洞大小随着弯曲壁面直径的增大而逐渐地减小，该变化规律与被吸收的能量变化呈现出一致的变化趋势，所以可以认为，被吸收的能量转化为了对壁面的破坏能，即弯曲壁面被破坏的越严重，则其对能量的吸收也就越多。

参考文献

[1]管公顺，庞宝君，哈跃，等.铝球弹丸超高速正撞击铝 Whipple 防护结构舱壁的损伤分析[J].兵工学报，2007，28(1)

[2]M. Takaffoli, M. Papini. Material deformation and removal due to single particle impacts on ductile materials using smoothed particle hydrodynamics.[J].Wear 274–275 (2012) 50–59

[3]G.Kay.Failure modeling of titanium 6Al –4V and aluminum 2024 –T3 with the Johnson –Cook material model [S].Tech. Rep.DOT/FAA/AR –03/57.US department of Transportation, Federal Aviation Administration, September, 2003

[4]王猛，张立佼.薄壁管壳高速正撞击穿孔特性的数值研究[J].航空学报，2015.10.12：3876–3884

[5]贾光辉，黄海，胡震东.超高速撞击数值仿真结果分析[J].爆炸与冲击，2005，25(1)

[6]阎晓军，张玉珠，聂景旭，等.超高速碰撞下 Whipple 防护结构的数值模拟[J].宇航学报，2002，23(5)：81–84

[7]管公顺.航天器空间碎片防护结构超高速撞击特性研究[D].哈尔滨：哈尔滨工业大学，2006.12

[8]贾光辉，黄海，胡震东.超高速撞击数值仿真结果分析[J].爆炸与冲击，2005，25(1)

[收稿日期：2017.3.3]