动态加载介绍

2016年博客

2024.1.5

技术背景

案例1:

一些重要的节日，APP启动时会将原有的启动图换成新的图片。

案例2:

在安卓市场可能拒绝上架含有广告的应用。通过在服务器配置一个开关，审安卓市场审核通过后，打开服务器端的广告开关，规避审核。

道高一尺魔高一丈。安卓市场开始扫描APK里面的Manifest甚至dex文件，查看开发者的APK包里是否有广告的代码，如果有就有可能审核不通过。

能否在用户运行APP的时候，再从服务器下载广告的代码，运行，再显示广告呢?

——动态加载:

程序运行的时候，加载并执行一些程序自身原本不存在的可执行文件。

动态加载

1.应用在运行的时候通过加载一些**本地不存在**的可执行文件实现一些特定的功能;

2.这些可执行文件是**可以替换**的;

3.更换静态资源（比如换启动图、换主题、或者用服务器参数开关控制广告的隐藏现实等）**不属于** 动态加载;

4.Android中动态加载的**核心思想是动态调用外部的dex文件**，极端的情况下，Android APK自身带有的Dex文件只是一个程序的入口（或者说空壳），所有的功能都通过从服务器下载最新的Dex文件完成;

动态加载的类型

1. 动态加载so库；

- Android的NDK中，可以动态加载.so库并通过JNI调用其封装好的方法。
- 用于对性能有需求的工作（比如T9搜索、Bitmap的解码、图片高斯模糊处理）。
- so库相比中dex文件反编译得到代码更难被破解，so库也可以被用于安全领域。
- 一般情况把so库一并打包在APK内部，但是so库也可以从外部存储文件加载。

2. 动态加载dex/jar/apk文件；

- 使用ClassLoader动态加载dex/jar/apk
- Android app运行就是执行dex文件里的代码。动态加载可以在app运行时加载外部的dex文件。下载新的dex并替换原有的dex就可以不安装新APK文件改变代码。

使用动态加载技术，一般来说会使得工作变得复杂，开发方式不是官方推荐的，不是目前主流的Android开发方式，目前只有在天朝才有比较深入的研究和应用，如一些SDK组件项目和BAT的项目上，Github上的相关开源项目基本是国人在维护。

动态加载大致过程

出于安全考虑，Android并不允许直接加载手机外部存储这类noexec（不可执行）存储路径上的可执行文件。

对于外部的可执行文件，调用前都要先拷贝到data/packagename/内部储存文件路径，**确保库不会被第三方应用恶意修改或拦截**，然后再加载到当前运行环境执行。

动态加载流程：

1.把可执行文件（.so/dex/jar/apk）拷贝到应用APP内部存储；

2.加载可执行文件；

3.调用具体的方法执行；

动态加载SO—普通方式

https://github.com/kaedea/android-dynamical-loading

以一个“图片高斯模糊”的功能为例，使用开源的高斯模糊项目

https://github.com/kikoso/android-stackblur

1.编译出SO库

2.把SO库复制到项目中

3.使用System类的loadLibrary()方法加载名为"stackblur"的共享库。

loadLibrary()方法是Java提供的用于加载本地共享库的方法。通过传递共享库的名称作为参数，它会在运行时加载该共享库。

动态加载SO—普通方式

```
// load so file from internal directory
        try {
            System.loadLibrary("stackblur");
            // 使用System类的loadLibrary()方法加载名为"stackblur"的共享库。
            // loadLibrary()方法是Java提供的用于加载本地共享库的方法。
            // 通过传递共享库的名称作为参数，它会在运行时加载该共享库。
            NativeBlurProcess.isLoadLibraryOk.set(true);
            Log.i("MainActivity", "loadLibrary success!");
        } catch (Throwable throwable) {
            Log.i("MainActivity", "loadLibrary error!" + throwable);
        }
```

. 加载成功后就可以直接使用Native方法了

```
public class NativeBlurProcess {
    public static AtomicBoolean isLoadLibraryOk = new AtomicBoolean(false);
    //native method
    private static native void functionToBlur(Bitmap bitmapOut,
        int radius, int threadCount, int threadIndex, int round);
    }
```

动态加载SO—两种load方法

system类有load和loadLibrary两个方法：

```
public static void load(String pathName) {
    Runtime.getRuntime().load(pathName, VMStack.getCallingClassLoader());
}
public static void loadLibrary(String libName) {
    Runtime.getRuntime().loadLibrary(libName,
                                     VMStack.getCallingClassLoader());
}
```

loadLibrary—ClassLoader.findLibrary—DexPathList—Runtime — doLoad
loadLibrary方法一步步找到完整的SO库路径，再加载目标SO库。

load方法直接给出完整的SO路径并加载。

动态加载SO—从外部存储

那能不能直接加载外部存储上面的SO库？把SO库拷贝到SD卡上面试试。

> java.lang.UnsatisfiedLinkError: dlopen failed: couldn't map "/storage/emulated/0/libstackblur.so" segment 1: Permission denied
>

Google开发者论坛：

SD卡等外部存储路径是一种可拆卸的（mounted）不可执行（noexec）的储存媒介，不能直接用来作为可执行文件的运行目录，使用前应该把可执行文件复制到APP内部存储再运行。

动态加载dex/jar/apk--ClassLoader

对于Java，运行程序也就是运行类（编译得到的class文件），其中起关键作用的是ClassLoader。一个运行中的APP不只有一个ClassLoader。

- Boot类型的ClassLoader实例

在Android系统启动的时候会创建一个Boot类型的ClassLoader实例，用于加载系统Framework层级的类，APP启动的时候也会把这个Boot类型的ClassLoader传进来。

关于 BootClassLoader：

> - 根类加载器：BootClassLoader 是系统的根类加载器，它是类加载器层次结构中的最顶层。

> - 加载核心类和资源：负责加载安卓框架的系统类和资源。

> - 引导类路径：使用一组指定了系统类和库的位置的预定义的引导类路径。

> - 不可更改性：在系统启动时创建保持不变。无法直接创建或修改。

动态加载dex/jar/apk--ClassLoader

- APP自己的ClassLoader实例

APP自己的类保存在APK的dex文件里面。当APP启动时，系统会创建一个专门用于加载应用程序的类的类加载器实例，称为应用程序类加载器（Application Class Loader）。

一个运行的APP至少有2个ClassLoader。

动态加载dex/jar/apk--ClassLoader

- 创建自定义的ClassLoader实例

创建一个ClassLoader实例需要使用现有的ClassLoader实例作为Parent。因此，整个Android系统里所有的ClassLoader实例都会被一棵树关联起来，这也是ClassLoader的 **双亲代理模型**。

```
/*
     * constructor for the BootClassLoader which needs parent to be null.
     */
    ClassLoader(ClassLoader parentLoader, boolean nullAllowed) {
        if (parentLoader == null && !nullAllowed) {
            throw new NullPointerException("parentLoader == null && !nullAllowed");
        }
        parent = parentLoader;
    }
```

动态加载dex/jar/apk--ClassLoader

- ClassLoader双亲代理模型的特点和作用

在加载一个类的实例时：

1. 会先查询当前ClassLoader实例是否加载过此类，有就返回；
2. 如果没有。查询Parent是否已经加载过此类，如果已经加载过，就直接返回Parent加载的类；
3. 如果继承路线上的ClassLoader都没有加载，才由Child执行类的加载工作；

即，如果一个类被位于树根的ClassLoader加载过，那么在以后整个系统的生命周期内，这个类永远不会被重新加载。

作用：

- 共享功能，Framework层级的类一旦被顶层的ClassLoader加载过就缓存在内存中，不需要重新加载。
- 隔离功能，不同继承路线上的ClassLoader加载的类肯定不是同一个类。避免了用户自己的代码冒充核心类库的类访问核心类库包可见成员的情况。

动态加载dex/jar/apk--ClassLoader

- 使用ClassLoader一些情况

- 新类替换旧类

  新类在旧类前加载，如果已经加载过旧类，ClassLoader会一直使用旧类。

- 新类在旧类后加载

  可以使用一个与加载旧类的ClassLoader**没有树的继承关系**的另一个ClassLoader来加载新类，因为ClassLoader只会检查其Parent有没有加载过当前要加载的类，**如果两个ClassLoader没有继承关系，那么旧类和新类都能被加载。**

- 类型匹配问题

  只有当两个实例的**类名**、**包名**以及**加载其的ClassLoader**都相同，才会被认为是**同一种类型**。可能会出现类型匹配异常。

  同一个Class = 相同的 ClassName + PackageName + ClassLoader

动态加载dex/jar/apk--DexClassLoader和PathClassLoader

ClassLoader是一个抽象类，有两个子类：

- DexClassLoader可以加载jar/apk/dex，可以从SD卡中加载未安装的apk;

```
public class DexClassLoader extends BaseDexClassLoader {
    public DexClassLoader(String dexPath, String optimizedDirectory,
            String libraryPath, ClassLoader parent) {
        super(dexPath, new File(optimizedDirectory), libraryPath, parent);
    }
}
```

dexPath中的dex会被加载到optimizedDirectory中，并创建一个DexFile，如果optimizedDirectory为null，会使用原有的路径创建DexFile对象。

- PathClassLoader只能加载系统中已经安装过的apk中的dex;

动态加载dex/jar/apk—加载类的过程

loadClass方法遍历所有dex文件，调用loadClassBinaryName方法一个个尝试加载指定的类，loadClassBinaryName中调用了Native方法defineClass加载类。

- **自定义ClassLoader**

可以创建自己的类去继承ClassLoader，loadClass方法并不是final类型的，可以重载loadClass方法并改写类的加载逻辑。

ClassLoader双亲代理的实现很大一部分就是在loadClass方法中，可以通过重写loadClass方法避开双亲代理的框架，**重新加载已经加载过的类**，也可以在**加载类的时候注入一些代码**。

动态加载dex/jar/apk—麻烦之处

比起一般的Java程序，在Android中动态加载主要有两个麻烦的问题：

1. 组件类未注册。

   组件类（如Activity、Service等）需要在Manifest文件里面注册后才能工作（系统会检查该组件有没有注册），所以即使动态加载了一个新的组件类，没有注册的话还是无法工作；

2. Resource资源找不到。

   资源需要用对应的R.id注册好，运行时通过ID从Resource实例中获取对应的资源。动态加载的新类的资源ID可能和现有的Resource实例中保存的资源ID对不上。

动态加载—简单模式

加载dex文件--通过反射的方式修改待调用方法为public--调用。

1. 无法使用res目录下的资源，特别是使用XML布局；
2. 无法动态加载新的Activity等组件，因为这些组件需要在Manifest中注册，动态加载无法更改当前APK的Manifest。

可以先把要用到的全部res资源都放到主APK里面，把所有Activity先写进Manifest，**只通过动态加载更新代码，不更新res资源。**使用纯Java代码创建布局绕开XML布局来改动UI界面。使用Fragment代替Activity。

动态加载—代理Activity

宿主APK需要先注册一个空壳Activity代理插件的Activity。
有以下特点：

1. 插件APK可以调用宿主APK里的一些功能；
2. 宿主APK和插件APK都要接入一套指定的接口框架才能实现以上功能；

限制如下：

1. 需要在Manifest注册的功能都无法在插件实现，如应用权限、LaunchMode、静态广播等；
2. 宿主一个代理用的Activity难以满足插件一些特殊的Activity的需求，插件Activity的开发受限于代理Activity；
3. 宿主项目和插件项目的开发都要接入共同的框架，大多时候，插件需要依附宿主才能运行，无法独立运行；

核心在于“**使用宿主的一个代理Activity为插件所有的Activity提供组件工作需要的环境**”。

动态加载—动态创建Activity

核心是“运行时字节码操作”。宿主注册一个不存在的Activity，启动插件的某个Activity时都把想要启动的Activity替换成已注册的Activity，从而使插件的Activity能正常启动。

1.主APK可以启动一个未安装的插件APK;

2.插件APK可以是任意第三方APK，无需接入指定的接口，理所当然也可以独立运行;

- 如何修改需要启动的目标Activity

如何把需要启动的、在Manifest中没有注册的PlugActivity换成有注册的TargetActivity?

自定义类继承ClassLoader，重载loadClass方法改写类的加载逻辑，在需要加载PlugActivity的时候，偷偷把其换成TargetActivity。

动态加载—动态创建Activity

- 遇到的问题

在插件Activity启动另一个Activity的时候怎么办？

启动其他Activity的时候，也采用动态创建Activity的方式

从主项目启动插件MainActivity的时候，启动的其实是动态创建的TargetActivity（extends MainActivity），而Activity启动另一个Activity的时候需要使用“startActivityForResult”方法，所以可以在动态创建TargetActivity的代码中，重写其“startActivityForResult”方法。

动态加载—动态创建Activity存在的问题

- 存在的问题

- 使用同一个注册的Activity，需要在Manifest注册的属性无法做到每个Activity都自定义配置；**（只做应用解耦可以全部自定义？）**
- 无法动态添加权限。插件中的权限，无法动态注册，插件需要的权限需要在宿主中注册;
- 插件的Activity无法开启独立进程，因为需要在Manifest里面注册;
- 对Service的支持可能不稳定。

  用动态创建方式创建Service组件时，Service实例化会调用ContextImpl.getApplicationContext方法获取应用程序的全局 Context。动态加载创建的Service实例的Context会被包装在一个 ContextWrapper 中。