聊聊陈旧的插件化

Drummor 鸿洋 2024-05-24 08:35 北京

本文作者

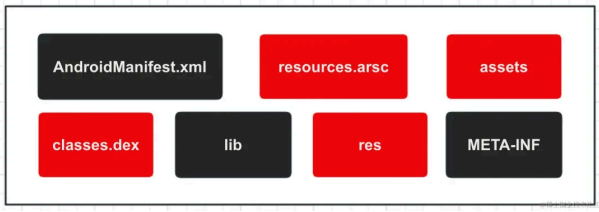
作者：**Drummor**

链接：
https://juejin.cn/post/7359102751907758118
本文由作者授权发布。

前言

之前写过一篇给自己填了坑，聊聊陈旧的插件化，还有小伙伴催更，连夜干了一篇。话不多说开始。

https://juejin.cn/post/7283087306604314636



先大概了解下整个Apk的构成。

- **AndroidManifest.xml**：用于描述APK包的组件和权限信息。
- **classes.dex**：这是一个包含Java字节码文件的文件。
- **resources.arsc**：会把资源打包编译成该格式，被为每份资源分配一个ID。
- **lib 目录**：这个目录中包含了应用程序的所有本地代码库文件。（so文件）
- **assets 目录**：文件和资源，该目录下文件不会分配资源ID。
- **res 目录**：资源文件会被分配资源ID存储在`resources.arsc`里，包括布局文件、字符串文件、颜色文件等。
- **META-INF 目录**：包含了APK的签名信息。

本文着重是从加载 Dex也就是类和资源(`resources.arsc`/res)的方向展开讨论。

1 插件类加载

插件化的三个核心问题的第一个，类的动态加载和使用。具体一些就是加载未安装等 Apk 中的类执行其方法、读取获取其属性属性等。解决插件化中如何动态加载【可执行】代码这一核心问题。

1.1、 类的动态加载

- 首先自定义 ClassLoader，使用自定义 ClassLoader 我们可以加载指定的不在宿主内的类文件。这个特性是Java语言的特性。
- 通过自定义的 ClassLoader 我们可以加载我们想要的指定的类。

继承 DexClassLoader，DexClassLoader 的构造方法有四个参数。

```
public DexClassLoader(String dexPath,
            String optimizedDirectory,
            String librarySearchPath,
            ClassLoader parent) {
```

- **dexPath** : apk、aar、jar 等包含 [class] 的文件路径，该 ClassLoader 执行类加载动作时，会在该路径下查找类文件。
- **optimizedDirectory** : 用于设置 DexClassLoader 对`.dex`优化后的存储位置，该参数在 Android8.0 以上(api26)以上失效。
- **librarySearchPath**: 本地库文件(so)的存储位置。关于so文件的动态加载这里暂时不展开。
- **parent**：设置该 ClassLoader 的父 ClassLoader。众所周知，ClassLoader 有双亲委派机制，即加载动作先交给父 ClassLoader 加载，父ClassLoader不加载时才自身来加载。可能有疑问为啥叫【双亲】委派，可能是翻译水平的问题，parent 翻译成了双亲。

动手写，如下其实没那么复杂不是么。

```
public class ApkClassLoader extends DexClassLoader {
  public ApkClassLoader(String dexPath,
              String optimizedDirectory,
              String librarySearchPath,
              ClassLoader hostClassLoader) {
    super(dexPath, optimizedDirectory, librarySearchPath, hostClassLoader);
  }
}
```

1.2、 插件逻辑使用

有了锤子我们得找钉子哇，用定义好的 ClassLoader 去动态加载 Apk 中的 [class]。

制作一个 Apk ，使用全包类名加载指定的类。

- 有了 插件中的 Class 我们就可以创建对象，执行其静态方法，创建对象等诸多操作。

> - 展开说下 interface 在里面起到的作用。
> - 以及 interface class 由谁负责加载

> - ApkclassLoader 与宿主 ClassLoader 的父子关系。

把一个类加载起来之后，就可以利用反射，调用方法、获取属性、创建实例了。但在实际的工程里这种方式还是比较野路子。往往更为正规的方式，我们会规定好必要的接口让插件来实现。这也符合软件工程设计里的【依赖倒置原则】即不依赖具体实现，而是依赖抽象类或者接口。具体的实现可变，抽象/接口不常变。

举个例子，我们规定一种加法操作，然后在插件 Apk 里实现这个加法操作。对于使用方来说依赖 `PluginArithmetic` 这个接口，实现由我们的插件承接。

- **声明**：定义一个算法接口内声明一个加法方法。

```
interface IPluginArithmetic {
  //加法
  fun plus(num: Int, addNum: Int): Int
}
```

- **实现**： 在插件中对接口做具体的实现

```
@Keep
class IPluginArithmeticImpl : IPluginArithmetic {
  override fun plus(num: Int, addNum: Int): Int {
    return num + addNum
  }
}
```

- **发现&使用**: 宿主中获取类创建实例调用方法。

```
private fun invokeIncreaseByPlugin() {
  val arithmeticClzName = "com.sample.sample_plugin.PluginArithmeticImpl"
  val arithmeticClzNameImpl: Class<*> = apkClassLoader.loadClass(arithmeticClzName)
  val arithmetic: IPluginArithmetic =
    getInterface(IPluginArithmetic::class.java, arithmeticClzNameImpl)
  val result = arithmetic.plus(1, 1)
  Toast.makeText(this.applicationContext, "result:${result}", Toast.LENGTH_SHORT).show()
}
```

1.3、ClassLoader 的分与合

宿主和插件的类加载共用一个ClassLoader还是分开用不同的ClassLoader，市面上大部分插件化方案都是采用插件用单独 ClassLoader。两种方案各有优劣，以下是他们的一些特点。

	优点	缺点	适用场景
合并 classlaoder	宿主与插件之间方便相互访问;减少重复的冗余类，降低包体	引入兼容问题，宿主与插件依赖的library库需要做版本控制。插件/宿主的发版，不灵活	插件需要大量使用宿主的能力，本身存在比较大耦合
插件独立classloader	插件更独立，对宿主的依赖小，发版更灵活；ClassLoader天然隔离，无依赖库版本不兼容问题。	宿主与插件之间通信有限制需要引入通信机制。宿主/插件有相同的类影响包体	插件相对独立，与宿主共用依赖库少，通信接口单一

从降低维护成本和开发灵活度的角度出发，个人比较倾向分开的方式。下面就独立ClassLoader这个方向举两个Case详细的说明阐述。

1.3.1、通信接口谁加载

在插件中和使用方（宿主）中都使用了 `IPluginArithmetic` 这一接口类，我们的插件和宿主是不同的 ClassLoader，那这个 `IPluginArithmetic` 类由哪个 ClassLoader 负载加载呢？

这里似乎没有标准答案，我们这个例子，这种情形下，接口类是相对固定的，让 `HostClassLaoder` 加载比较合适。反过来如果让插件加载这个类，那么插件的Classloader作为`HostClassLoader`的父类，是不合理的。

实现宿主Classloader加载这个插件类的方式，插件 ClassLoader 的 Parent classeLoader 设置为宿主的 classloader，ClassLoader 双亲委派是让其父ClassLoader 先加载。

1.3.2、相同类谁加载

延伸出来，对于同一个类（全包类名相同），宿主与插件之间，插件与插件之间如果相同怎么办?

互不打扰，各自用各自的。这也是独立ClassLoader的重要作用。

再有一个问题，如果我们 B 插件要使用 A 插件中的某段逻辑，假设他们的也是通过接口方式该怎么做呢？这里有一个依赖的概念，B 插件使用A插件，那么B插件在逻辑上仰仗"依赖"A插件的实现。

依赖的实现，是可以通过我们自定义的ClassLoader传入要依赖的插件所属的 ClassLoader，优先让被依赖插件的ClassLoader先加载。当然如果有其他的考量需要引入白名单机制，在白名单里的类让被依赖的插件ClassLoader具备优先加载权利。

2 资源的插件化

正常的 Apk 安装到系统后，我们APP能够正常通过 Resource 对象了使用资源了，无论在在 Xml 中使用`@drawable/xxxx` 还是在代码，使用诸如 `R.layout.xxx` 。

但对于一个插件 Apk 是没有安装到系统的，我们如何能够正常的使用到插件中的资源呢？解决这个问题的方案我们姑且称之为 **资源的插件化**。

如何使用插件包中的资源，插件包可能会用到宿主的资源。

创建Resource有两种方式：

2.1、创建 Resource对象

如何创建一个插件的Resource，加载Resource的资源呢。

系统有公开的API。

主要有两种方式：通过`AssetManager`

```
@Deprecated
public Resources(AssetManager assets, DisplayMetrics metrics, Configuration config) {
   this(null);
   mResourcesImpl = new ResourcesImpl(assets, metrics, config, new DisplayAdjustments());
}
```

`AssetManager`没有公开的API的构建，我们可以通过反射的方式创建。

```
protected AssetManager createAssetManager(File apk) throws Exception {
   AssetManager am = AssetManager.class.newInstance();
   Reflector.with(am).method("addAssetPath", String.class).call(apk.getAbsolutePath());
   return am;
}
```

Resources 构造需要的另外两个参数，直接使用宿主 Resource 的 DisplayMetrics 和 Configuration 。

这样就构建出了插件的 Resource 对象。

插件可能会用到宿主的资源，主要是系统有可能从宿主 Manifest 中获取 app icon 或者logo 的资源ID，然后直接向插件的 Resources 对象查询这些资源。为了解决这个问题，需要让插件的 Resource 包含宿主的 Resource。还是刚刚那个套路构建一个 MixResource，复写Resource的方法，在资源获取不到时，再尝试从宿主Resource中获取，方法比较多，以获取字符串方法为例。

```
override fun getString(id: Int): String {
   return try {
      pluginResource.getString(id)
   } catch (e: Exception) {
      hostResources.getString(id)
   }
```

2.2、无 hook 方式

2.1 章节中，为了使用 Resource 构造方法所需的 `AssetManager` 对象，创建 `AssetManager` 使用反射的方式。系统还提供了另外一种方式创建 Resource 对象 ，如下：

```
 # PackageManager
@NonNull
public abstract Resources getResourcesForApplication(@NonNull ApplicationInfo app)
     throws NameNotFoundException;
```

使用该方法，我们不需要 AssetManager 对象，就可以创建 Resource 对象，而 `AssetsManager` 对象，可以直接在 Resource 对象中获取。

```
val packageManager = hostAppContext.packageManager
val hostApplicationInfo = hostAppContext.applicationInfo
val pluginApplicationInfo = ApplicationInfo()
pluginApplicationInfo.packageName = hostApplicationInfo.packageName
pluginApplicationInfo.uid = hostApplicationInfo.uid
val pluginResource = packageManager.getResourcesForApplication(pluginApplicationInfo)
```

2.3、资源冲突

2.3.1 、Resources ID 的构成

```
[0xPPTTEEEE]
```

appt对资源打包时，每个资源都对应一个唯一的资源 id，资源 id 是一个 8 位的 16 进制 int 值 0xPPTTEEEE：

- PP： 前两位是 PackageId 字段，系统资源是 01，正常打包的PP字段值为7f。
- TT： 中间两位是 TypeId 字段，表示资源的类型
- EEEE： 最后四位是 EntryId 字段。相同PP且相同TT时不会有相同的EEE。

2.3.2 、资源冲突问题

插件包和宿主是独立打包的，就会存在插件中和宿主中生成了相同的资源ID，插件先加载自己的再去加载宿主的，如果存在相同的ID，可能加载不同类型甚至不存在的资源。

2.3.3 、解决方案

这里介绍以下官方插件化的案（AAB）解决资源冲突的方案。

```
//Resources.java
public String sourceDir;
public String publicSourceDir;
public String[] sharedLibraryFiles;
```

是把宿主路径和插件包路径都给Resources，以下是Resources的代码，sourceDir和sharedLibraryFiles制定的路径资源都会进入到 Resources 中。再结合 aapt2 提供的参数配置我们可以指定资源ID的PP字段，避免与宿主资源产生冲突。但是这里有个需要注意的点：

- `additionalParameters` 指定参数在Android O （26）及以上不能指定小于 0x7F 的 package-id。Android O （26） 以下不能指定大于0x7F以下的点。
- 指定小于0x7f的package-id时，需要追加 `--allow-reserved-package-id` 参数。示例如下：

```
aaptOptions {
   additionalParameters "--package-id", "0x7E", "--allow-reserved-package-id"
}
```

2.3.4、 其他方案

- 修改 aapt 生成 ResoruceID、或者插件包生成后修改插件包重新改造 ResoruceID。市面上开源库常用方案这里不展开。
- 字节方案，看到字节团队博文通过修改插件包获取资源的方式。核心原理是利用 `Resources#getIdentifier`方法可以通过资源名称获取资源ID，结合字节码编辑讲直接使用

ID方式转换为通过名称获取资源的方式。

2.4 、资源插件化小结

- 构建插件 Resource 有两种方式，一种通过反射创建`AssetManager`，然后公开的Resource构造方法创建得到。
- 另外一种是通过公开的 Resource 构造方案是通过`packageManager.getResourcesForApplication(...)`创建。
- 资源 ID 冲突的问题也有方案解决。一种是无 hook 的通过AAPT2设置PP字段方案，另外是一些hook的方式修改插件包绕开。
- 我们这里没有讨论资源合并或者分开的问题。上述方案插件是能够使用宿主资源的。宿主使用插件的资源，是可以在宿主中获取到插件的 Resource 对象然后进行自由的控制使用。

3 练习

> Talk is cheap. Show me the code.

- 动态加载插件，执行其中的方法。
- 加载插件中的View，添加到指定的布局中。

代码已经上传到：
https://github.com/drummor/PluginSampleApplication

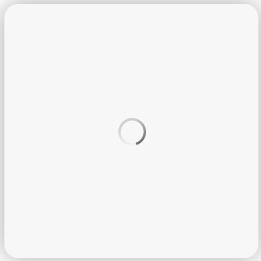
最后推荐一下我做的网站，玩Android: *wanandroid.com* ，包含详尽的知识体系、好用的工具，还有本公众号文章合集，欢迎体验和收藏！

推荐阅读：

Android崩在so里面，怎么定位Native堆栈呢?

RecyclerView面试宝典：7大高频问题解析

2024 Google I/O Android 相关内容



扫一扫 关注我的公众号
如果你想要跟大家分享你的文章，欢迎投稿~

┏(^ 0 ^)┛明天见!

阅读原文

喜欢此内容的人还喜欢

Flutter鸿蒙终端一体化-天下一统
鸿洋

WO靠！鸿蒙的自由流转还能这么玩?
鸿洋

Android 描边动画实现母亲节祝福效果
鸿洋