Android应用程序启动流程

ActivityThread

AMS ActivityManagerService

IApplicationThread 接口

ActivityThread.ApplicationThread

通过android:process 让特定组件在不同的进程中运行。

android.app.ActivityThread

mClassLoader

# Android 动态加载技术基础

自定义类加载器ClassLoader

标准Java虚拟机

继承ClassLoader通过defineClass来加载Class

Dalvik虚拟机

VMClassLoader的defineClass方法未实现

从ClassLoader派生了两个类DexClassLoader和PathClassLoader

重载方法findClass

protected Class<?> loadClass(String className,Boolean resolve)

throws ClassNotFoundException{

}

双亲委派模型的类加载器

## DexClassLoader和PathClassLoader的区别？

DexClassLoader需要提供一个可写的outpath路径，用来释放.apk包或者.jar包中的dex文件。换个说法来说，就是PathClassLoader不能主动从zip包中释放出dex，因此只支持直接操作dex格式文件，或者已经安装的apk（因为已经安装的apk在cache中存在缓存的dex文件）。而DexClassLoader可以支持.apk、.jar和.dex文件，并且会在指定的outpath路径释放出dex文件。

另外，PathClassLoader在加载类时调用的是DexFile的loadClassBinaryName，而DexClassLoader调用的是loadClass。因此，在使用PathClassLoader时类全名需要用”/”替换”.”

## 动态加载类实现

1 源码结构

IDynamic.java 动态接口

package com.dynamic.interfaces;

public interface IDynamic{

public void init(Activity activity);

public void showBanner();

}

**Dynamic.java 动态类**

package com.dynamic.impl

public class Dynamic implements IDynamic{

private Activity mActivity;

public void init(Activity activity);

public void showBanner();

}

2 打包动态类

使用Eclipse导出Jar包dynamic.jar 只包括动态类Dynamic.java

dx –dex –output=dynamic\_temp.jar dynamic.jar

变成的dex文件，并打包为jar

3 打包动态类接口文件jar用于程序开发

然后可使用DexClassLoader加载dynamic\_temp.jar

# 类加载器

PathClassLoader是Android应用中的默认加载器

DexClassLoader可以加载任何路径的apk/dex/jar

PathClassLoader只能加载/data/app中的apk，也就是已经安装到手机中的apk。这个也是PathClassLoader作为默认的类加载器的原因，因为一般程序都是安装了，在打开，这时候PathClassLoader就去加载指定的apk(解压成dex，然后在优化成odex)就可以了。

# 应用换肤原理

如何快速获取程序包名

adb shell dumpsys activity top

AssetManager

addAssetPath

# 动态加载Activity 免安装运行程序

1 用DexClassLoader不能直接加载Activity,因为组件有自己的生命周期。如果直接加载，将会只是一个普通的类

LoadedApk.java

public final class LoadedApk{

private ClassLoader mClassLoader;

}

ActivityThread.java

final ArrayMap<String,WeakReference<LoadedApk>> mPackages;

# 双亲委派模型

**什么是类加载器？**

类加载器就是根据指定全限定名称将class文件加载到JVM内存，转为Class对象。

**分类：**

自举类加载器 Bootstrap ClassLoader

其他类加载器

Extension ClassLoader

Application ClassLoader

关键函数

**双亲委派模型工作过程**

如果一个类加载器收到类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器完成。每个类加载器都是如此，只有当父加载器在自己的搜索范围内找不到指定的类时（即ClassNotFoundException），子加载器才会尝试自己去加载。

**为什么需要双亲委派模型？**

在JVM中，判断一个对象是否是某个类型时，如果该对象的实际类型与待比较的类型的类加载器不同，那么会返回false。

**如何实现双亲委派模型？**

loadClass函数来实现

public Class<?> loadClass(String name)throws ClassNotFoundException{

}

protected Class<?> findClass(String name)throws ClassNotFoundException{

}

protected final Class<?> defineClass(String name,byte[] b,int off,int len){

}

## references

<https://blog.csdn.net/huachao1001/article/details/52297075>

<http://www.cnblogs.com/lanxuezaipiao/p/4138511.html>

# References

<https://blog.csdn.net/jiangwei0910410003/article/details/48415225>

<https://juejin.im/entry/57a42c52d342d30057506337>

<http://gityuan.com/2017/04/02/android-application/>

<https://blog.csdn.net/shifuhetudi/article/details/52089562>

<https://blog.csdn.net/jltxgcy/article/details/50540309>

<https://blog.csdn.net/qinjuning/article/details/7310620>

<https://blog.csdn.net/jiangwei0910410003/article/details/48104581>