# Android APK资源分析之Python实现

# 作者

马彦彦 京东零售-技术与数据中台-共享技术部-移动技术组 软件开发工程师岗

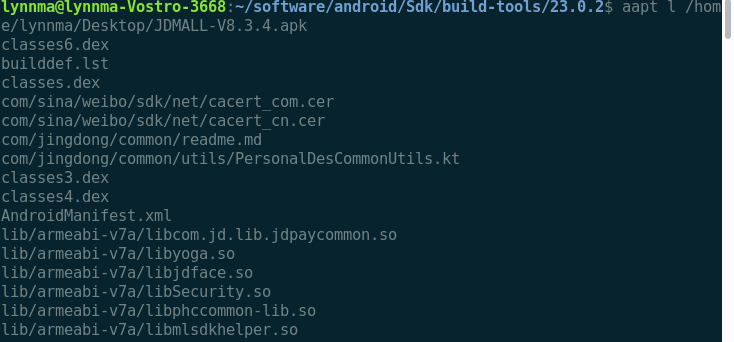
1. **背景**

随着业务的快速迭代增长，京东主站APP不断加入新的代码、图片资源、第三方SDK、React Native等等，直接导致APK体积不断增加。APK体积增长也会带来诸多问题，例如：推广费用增加，用户下载意愿降低，流量费用增加，下载及安装成功率降低，甚至可能会影响用户留存率；应用市场限制，Google Play规定安装包上限为100M。所以APK瘦身已经是迫在眉睫的事情。在尝试瘦身过程中，我们借鉴了很多业界其他同行的方案，比如资源混淆、图片压缩/转码等，同时针对自己需求发现了一些新的技巧。本文主要讲解如何使用Python对APK进行分析，统计基础数据，分析可优化的资源，为应用瘦身提供数据支持。

分析APK的前提条件就是要充分了解APK组成，所以下文将首先简单介绍APK组成。

1. **APK文件目录**

APK是一个压缩包，使用aapt l file.apk命令可以查看APK下所有文件，如下:



简单归类如下 ：



当然还会有一些其他文件，例如org/，src/，aidl等等文件或文件夹，这些资源是Java Resources，具体详情可以了解下APK打包流程。

在充分了解APK组成部分后，下面来介绍下APK扫描实现主要工作

1. **APK分析主要工作**

分析APK主要分为以下部分:

1. 下载APK以及mapping文件
2. AAPT获取APK信息
3. 读取APK在操作系统中大小(apk\_file\_size)以及APK真正大小(apk\_download\_size)
4. 还原混淆后资源ID
5. 根据文件MD5判断重复资源文件
6. 读取DEX头文件获取classes.dex的class\_numbers和references\_methods
7. 获取非alpha通道图以及图片尺寸，遍历出非透明通道图
8. ZipFile解压APK的so文件并读取so文件内容，还原so混淆的资源ID，so中非透明通道图
9. res/下无用资源

以上工作主要使用Python进行实现，Python断点续传下载APK以及Mapping文件之后解压文件,为之后分析APK做好准备。这里关于Python下载实现不再赘述。

* 1. **AAPT获取APK信息**

通过AAPT命令可以获取APK的package\_name，version\_name，version\_code，launch\_activity，min\_sdk\_version，target\_sdk\_version，application\_label等信息

具体实现如下

def get\_apk\_base\_info(self):

# 获取apk包的基本信息

p = subprocess.Popen(self.aapt\_path + **" dump** *badging* **%s"** % self.apkPath, stdout=subprocess.PIPE,  
 stderr=subprocess.PIPE,  
 stdin=subprocess.PIPE, shell=True)  
 (output, err) = p.communicate()  
 package\_match = *re*.compile(*"package: name='(\S+)' versionCode='(\d+)' versionName='(\S+)'"*).match(output.decode(*)*)  
 if not package\_match:  
 raise Exception(**"can't get package,versioncode,version"**)  
 package\_name = package\_match.group(1)  
 version\_code = package\_match.group(2)  
 version\_name = package\_match.group(3)  
 launch\_activity\_match = re.*compile*(**"***launchable***-activity: name='(\S+)'"**).search(output.decode())  
 if not launch\_activity\_match:  
 raise Exception(**"can't get launch\_activity"**)  
 launch\_activity = launch\_activity\_match.group(1)  
 *sdk\_version\_match* = re.compile(**"sdkVersion:'(\S+)'"**).search(output.decode())  
 if not sdk\_version\_match:  
 raise Exception(**"can't get min\_sdk\_version"**)  
 min\_sdk\_version = sdk\_version\_match.group(1)  
 *target\_sdk\_version\_match* = re.compile(**"targetSdkVersion:'(\S+)'"**).search(output.decode())  
 if not target\_sdk\_version\_match:  
 raise Exception(**"can't get target\_sdk\_version"**)  
 target\_sdk\_version = target\_sdk\_version\_match.group(1)  
 application\_label\_match = re.compile(*"application-label:'([\u4e00-\u9fa5\_a-zA-Z0-9-\S]+)'"*).search(output.decode*(*))  
 if not application\_label\_match:  
 raise Exception(**"can't get application\_label"**)  
 application\_label = application\_label\_match.group(1)  
 return package\_name, version\_name, version\_code*,*launch\_activity*,*min\_sdk\_version*,*target\_sdk\_version*,application\_label*

* 1. **apk\_file\_size & apk\_download\_size**

apk\_file\_size是APK在操作系统中占据存储空间，可以通过os模块直接获取； apk\_download\_size是APK内实际大小，可以ZipFile获取每个文件压缩大小，实现如下：

def get\_apk\_size(self):

# 得到apk的文件大小  
 *size* = round(os.path.getsize(self.apkPath) / (1024 \* 1000), 2)  
 # return str(size) + "M"  
 return os.path.getsize(self.apkPath)

def get\_apk\_download\_size(apk\_file\_name):  
 # 获取apk\_download\_size  
 zip\_file = zipfile.ZipFile(apk\_file\_name, **'r'**)  
 zip\_infos = zip\_file.infolist()  
 download\_size = 0  
 for index in range(len(zip\_infos)):  
 zip\_info = zip\_infos[index]  
 download\_size += zip\_info.compress\_size  
 return download\_size

* 1. **ZipFile读取APK文件**

许多人多使用apktool.jar解压APK，然后遍历APK文件夹，该方法可以解决除apk\_download\_size大部分功能，介于以上获取apk\_download\_size使用ZipFile读取APK文件，这里同样采用ZipFile读取APK内容。且将APK文件作为压缩文件直接使用ZipFile进行读取压缩文件内容，该方法可以免去解压APK流程，一定程度上提高遍历速度。

def \_\_get\_files\_from\_apk(apk\_file\_name, apk\_name\_without\_suffix, mapping\_name\_without\_suffix):

# 读取混淆文件  
 proguard\_map = reproguard.read\_proguard\_apk(mapping\_name\_without\_suffix)  
 zip\_file = zipfile.ZipFile(apk\_file\_name, **'r'**)

# 获取APK文件内所有文件列表  
 file\_name\_list = zip\_file.namelist()  
 # 遍历APK文件下文件列表  
 for index in range(len(file\_name\_list)):  
 file\_name = str(file\_name\_list[index])  
 # 还原混淆文件  
 if proguard\_map:  
 entry\_name = str(reproguard.replace\_path\_id(file\_name, proguard\_map)) if(**"/"** in file\_name) else file\_name  
 else:  
 entry\_name = file\_name

# 获取文件MD5值  
 md5\_str = md5.get\_md5\_value(file\_name)  
 parent\_dir = entry\_name[:parent\_index] if parent\_index >= 0 else **""**

# 根据文件名获取文件的ZipInfo(压缩文件)zip\_info = zip\_file.getinfo(file\_name)  
 file\_info = FileInfo(  
 path=file\_name,  
 entry\_name=entry\_name,  
 md5\_str=md5\_str,  
 compress\_size=zip\_info.compress\_size,  
 file\_type=file\_type,  
 zip\_file=zip\_info  
 )  
 if **"so"** == file\_type and **"***libcom***.jd.lib"** in entry\_name:  
 # aura插件分析  
 ......

elif **"assets/***jdreact***/"** in entry\_name:  
 # React Native分析

......  
 elif **"dex"** == file\_type:  
 # dex分析方法类,获取APK中class数+references methods数

......  
 elif file\_util.is\_image(entry\_name):

# 如果是图片文件,就要分析图片的尺寸,以及判断是否是 非透明通道图

.....  
 zip\_file.close()  
 return apk\_file\_list, aura\_bundles, dex\_files, react\_modules

* 1. **解析DEX文件**

以上内容可以获取APK中大部分内容，但是要获取APK中涉及的class数目，以及methods数目，显然以上分析均不能满足条件。这也是需要了解DEX结构并分析DEX文件的原因所在。DEX文件作为Android APK的组成部分，是Android的Java代码经过编译生成class文件，在经过dx命令生成的，它包含了APK的源码，反编译时最主要就是对这个文件进行反编译。首先简单了解下DEX文件格式。

**DEX格式**

| **名称** | **格式** | **说明** |
| --- | --- | --- |
| header | header\_item | DEX文件头部，记录整个DEX文件的相关属性 |
| string\_ids | string\_id\_item[] | 字符串数据索引.记录了每个字符串在数据区的偏移量 |
| type\_ids | type\_id\_item[] | 类型数据索引.是DEX文件引用的所有类型(类,数组或原始类型)的字符串索引 |
| proto\_ids | proto\_id\_item[] | 方法原型索引.记录了方法声明的字符串(指向string\_ids),返回类型(指向type\_ids),参数列表(指向typeList) |
| field\_ids | field\_id\_item[] | 字段数据索引.是DEX文件引用的所有字段索引,记录了所属类,字段类型(指向type\_ids)和字段名(指向string\_ids) |
| method\_ids | method\_id\_item[] | 方法索引.记录了所属类名,定义类型(指向type\_ids),方法名称(指向string\_ids),方法原型(指向proto\_ids) |
| class\_defs | class\_def\_item[] | 类定义数据索引,记录了指定类各类信息,包括8各部分,类的类型,访问标志,父类类型,实现接口,源文件,注解,class\_data |
| method\_handles | method\_handle\_item[] | 方法句柄列表 |
| data | ubyte[] | 数据区,上面所有表格的支持数据 |
| link\_data | ubyte[] | 静态链接文件中使用数据 |

从DEX文件格式中我们看到有string、field、class、method等标识符列表，唯独没有我们想要了解的class、methods、field、string数量及其所在偏移量。这时我们看到一个header组成部分，经了解header组成如下:

|  |  |  |  |
| --- | --- | --- | --- |
| **字段名称** | **偏移值** | **长度** | **说明** |
| magic | 0x0 | 8 | 魔数字段，值为"DEX\n035\0" |
| checksum | 0x8 | 4 | 校验码 |
| signature | 0xc | 20 | sha-1签名 |
| file\_size | 0x20 | 4 | DEX文件总长度 |
| header\_size | 0x24 | 4 | 文件头长度，009版本=0x5c,035版本=0x70 |
| endian\_tag | 0x28 | 4 | 标示字节顺序的常量 |
| link\_size | 0x2c | 4 | 链接段的大小，如果为0就是静态链接 |
| link\_off | 0x30 | 4 | 链接段的开始位置 |
| map\_off | 0x34 | 4 | map数据基址 |
| string\_ids\_size | 0x38 | 4 | 字符串列表中字符串个数 |
| string\_ids\_off | 0x3c | 4 | 字符串列表基址 |
| type\_ids\_size | 0x40 | 4 | 类列表里的类型个数 |
| type\_ids\_off | 0x44 | 4 | 类列表基址 |
| proto\_ids\_size | 0x48 | 4 | 原型列表里面的原型个数 |
| proto\_ids\_off | 0x4c | 4 | 原型列表基址 |
| field\_ids\_size | 0x50 | 4 | 字段个数 |
| field\_ids\_off | 0x54 | 4 | 字段列表基址 |
| method\_ids\_size | 0x58 | 4 | 方法个数 |
| method\_ids\_off | 0x5c | 4 | 方法列表基址 |
| class\_defs\_size | 0x60 | 4 | 类定义标中类的个数 |
| class\_defs\_off | 0x64 | 4 | 类定义列表基址 |
| data\_size | 0x68 | 4 | 数据段的大小，必须4k对齐 |
| data\_off | 0x6c | 4 | 数据段基址 |

其中有method\_ids\_size和class\_defs\_size，这两个数据就是所需要得到class数量和references methods数量，另外还有一些其他string\_ids数目以及偏移量，type\_ids数量以及偏移量等等。

所以读取DEX header即可得到DEX中定义的class方法数以及引用方法数。Android 虚拟机也是通过引用方法数(references methods)进行分包。

references methods包括第三方引用方法+自定义方法数。可以作为分析DEX的标准，具体实现如下:

def ReadDexHeader\_(self, file\_dir):

# 以二进制形式读取文件  
 f = open(file\_dir, 'rb')  
 m = mmap.mmap(f.fileno(), 0, access=mmap.ACCESS\_READ)  
 self.mmap = m  
 ....省略DEX部分内容读取

# dex中用到的所有的字符串内容的大小  
 string\_ids\_size = struct.unpack(**'<L'**, m[0x38:0x3C])[0]  
 # dex中用到的所有的字符串内容的偏移值  
 string\_ids\_off = struct.unpack(**'<L'**, m[0x3C:0x40])[0]  
 # dex中的类型数据结构的大小  
 type\_ids\_size = struct.unpack(**'<L'**, m[0x40:0x44])[0]  
 # dex中的类型数据结构的偏移值  
 type\_ids\_off = struct.unpack(**'<L'**, m[0x44:0x48])[0]  
 # dex中的原型数据信息数据结构的大小  
 proto\_ids\_size = struct.unpack(**'<L'**, m[0x48:0x4C])[0]  
 # dex中的原型数据信息数据结构的偏移值  
 proto\_ids\_off = struct.unpack(**'<L'**, m[0x4C:0x50])[0]  
 # dex中的字段信息数据结构的大小  
 field\_ids\_size = struct.unpack(**'<L'**, m[0x50:0x54])[0]  
 # dex中的字段信息数据结构的偏移值  
 field\_ids\_off = struct.unpack(**'<L'**, m[0x54:0x58])[0]  
 # dex中的方法信息数据结构的大小  
 method\_ids\_size = struct.unpack(**'<L'**, m[0x58:0x5C])[0]  
 # dex中的方法信息数据结构的偏移值  
 method\_ids\_off = struct.unpack(**'<L'**, m[0x5C:0x60])[0]  
 # dex中的类信息数据结构的大小  
 class\_defs\_size = struct.unpack(**'<L'**, m[0x60:0x64])[0]  
 # dex中的类信息数据结构的偏移值  
 class\_defs\_off = struct.unpack(**'<L'**, m[0x64:0x68])[0]  
 # dex中数据区域的结构信息的大小  
 data\_size = struct.unpack(**'<L'**, m[0x68:0x6C])[0]  
 # dex中数据区域的结构信息的偏移值  
 data\_off = struct.unpack(**'<L'**, m[0x6C:0x70])[0]  
 # 此变量用于存放读取到的dex头部  
 header\_data = dict({})  
 header\_data[**'string\_ids\_size'**] = string\_ids\_size  
 header\_data[**'string\_ids\_off'**] = string\_ids\_off  
 header\_data[**'type\_ids\_size'**] = type\_ids\_size  
 header\_data[**'type\_ids\_off'**] = type\_ids\_off  
 header\_data[**'proto\_ids\_size'**] = proto\_ids\_size  
 header\_data[**'proto\_ids\_off'**] = proto\_ids\_off  
 header\_data[**'field\_ids\_size'**] = field\_ids\_size  
 header\_data[**'field\_ids\_off'**] = field\_ids\_off  
 header\_data[**'method\_ids\_size'**] = method\_ids\_size  
 header\_data[**'method\_ids\_off'**] = method\_ids\_off  
 header\_data[**'class\_defs\_size'**] = class\_defs\_size  
 header\_data[**'class\_defs\_off'**] = class\_defs\_off  
 header\_data[**'data\_size'**] = data\_size  
 header\_data[**'data\_off'**] = data\_off  
 self.header = header\_data

* 1. **获取非alpha通道图以及图片尺寸**

在计算机图形学中, 每一张图片都是由一个或多个数据通道构成。例如：RGB图像就是有3个通道构成，分别为R、G、B。而透明通道在一定程度上增强视觉感染力。本文的非alpha通道图是图片大小>10kb & 非.9.png&没有alpha通道的PNG图。该过程可以在一定程度上避免使用大图

from PIL import Image

try:  
 image\_bytes = io.BytesIO(zip\_file.read(file\_name))  
 img = Image.open(image\_bytes)  
 # print(img.format) # JPEG  
 # 图片尺寸,单位像素  
 image\_size = img.size # (801, 1200)  
 filename\_without\_suffix, image\_type = file\_util.get\_file\_type(entry\_name)  
 # mode图片模式,有9种，分别为1，L，P，RGB，RGBA(有alpha通道)，CMYK，YCbCr，I，F  
 if img.mode != "RGBA":  
 if image\_type == ".png" and not filename\_without\_suffix.endswith(".9") \  
 and zip\_info.compress\_size >= 10\*1024:  
 non\_alpha = True  
 except OSError:  
 pass  
 finally:  
 file\_info.image\_size = image\_size  
 file\_info.non\_alpha = non\_alpha  
 apk\_file\_list.append(file\_info)  
 continue

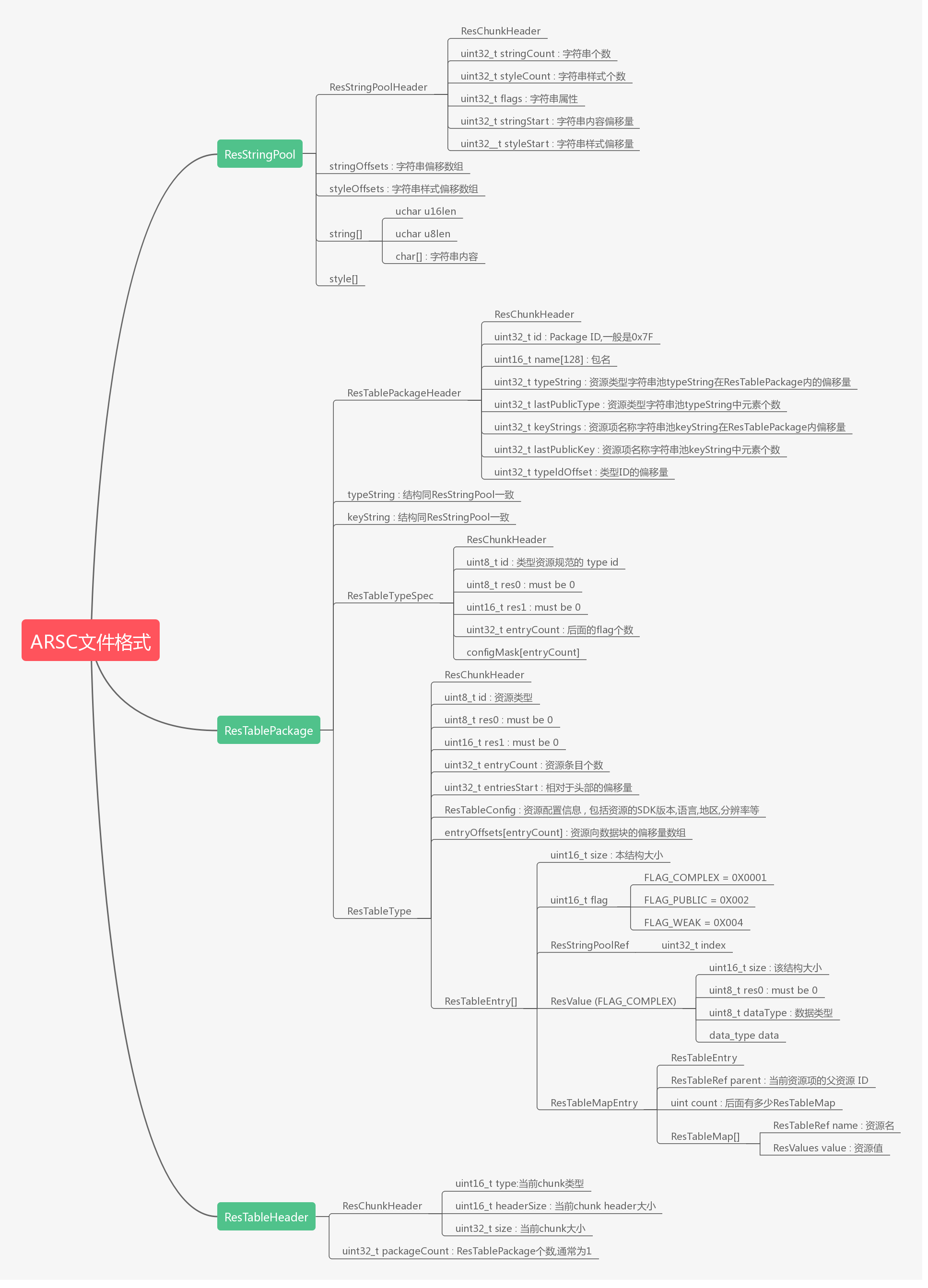
其中img.size和img.mode是核心代码，可以获取图片尺寸和图片模式

* 1. **重复资源**

重复资源的获取是通过对比资源文件的MD5值，一个APP中存在不同名称的图片具有相同的MD5值，那么这些图片便重复需要删除只保留一份即可，实现方法不再赘述。

* 1. **无用资源**

以上分析可以满足APP资源对比，分析资源增减情况需求，更加直观分析APP中大小增长过快模块。但在APK瘦身中还有一个更行而有效方法——无用资源，无用资源包含res目录下资源以及assets目录下资源。在分析这两块无用资源的首要工作：了解AAPT打包APK资源方法。在AAPT打包资源文件中，项目中的AndroidManifest.xml文件和布局文件XML都会被编译，然后生成相应的R.java，存放在APP的res目录下的资源在打包前会被编译成二进制文件，并且为每一个该类文件赋予一个resource id。对于该类资源的访问，应用层则是通过resource id进行访问。Android应用在编译过程中AAPT工具会对资源文件进行编译，并生成一个resource.arsc文件，而resource.arsc文件其实就是一个文件索引表，所有res目录下资源都会在这个文件中，所以分析res/无用资源首要就是要简单了解下resources.arsc文件。



在我们使用apktool进行反编译后，res/values/public.xml就是分析resources.arsc而来的。了解resource.arsc文件，可以编写Python代码进行分析，获取对应resource文件。当然使用apktool亦可反编译出相同文件。

### res目录无用资源

AAPT将res/目录下资源文件经过混淆直接保存至r目录下，且res资源可以被values目录下xml、非values目录下xml、AndroidManifest.xml、java代码中被引用，所以要分析该目录下的无用资源，需要步骤如下：

1. 解析R.txt，并将R.txt中resources\_ids作为全部资源set<resource\_id> unusedset列表中
2. 分析resources.arsc文件，得到被引用资源文件列表

该部分可以分为2部分：values资源扫描；非values资源(像layout、animation、drawable等目录下xml文件)扫描。这是因为values资源可以引用图片资源，而非values资源不仅可以引用图片资源且可以被引用

1. values文件夹下文件直接引用资源列表+manifest.xml文件中引用资源列表，均放入set<resource\_id> value\_references\_set列表中
2. 非values文件夹，且是xml文件中引用资源列表，放入map<resource\_id,set<resource\_id>> non\_values\_references\_map

3. 分析DEX转码为smali代码中直接引用的资源，放入set<resource\_id>code\_ref\_set列表中

4. 将以上所有set中数据合并至一个set中references\_ref\_set列表中

5. unusedset删除references\_ref\_set中数据

6. unusedset删除 shared\_res\_public.xml中标注的自定义so库(即aura)中引用的资源

7. unusedset删除需要被忽略的数据

以上7个步骤即可得到无用资源列表，当然会涉及到资源文件还原等工作，详情请参照上文。

核心实现代码如下:

def read\_resource\_txt\_file(mapping\_name):

resource\_txt\_path = mapping\_name + "/AndroidJD-Phone/hotfix/R.txt"  
 # R.txt解析结果  
 resource\_def\_map = dict({})  
 unused\_res\_set = set({})  
 try:  
 r\_txt\_file = open(resource\_txt\_path, "r")  
 line = r\_txt\_file.readline()  
 while line:  
 columns = line.split(" ")  
 if len(columns )>= 4:  
 resource\_name = "R."+columns[1]+"."+columns[2]  
 if not columns[0].endswith("[]") and columns[3].startswith("0x"):  
 if columns[3].startswith("0x01"):  
 print("ignore system resource %s", resource\_name)  
 else:  
 res\_id = \_\_parse\_resource\_id(columns[3].strip())  
 if res\_id:  
 resource\_def\_map[res\_id] = resource\_name  
 if not ignore\_resource(resource\_name):  
 unused\_res\_set.add(resource\_name)  
 else:  
 # print("ignore resource %s", resource\_name)

# styleable资源读取  
 ........

line = r\_txt\_file.readline()  
 except Exception as e:  
 raise Exception(resource\_txt\_path+**" file Error,"**, e)  
 finally:  
 return resource\_def\_map, styleable\_map, unused\_res\_set

# 遍历*smali*代码,获取引用资源的res\_id  
def read\_smali\_files(*smali*\_path, resource\_def\_map, styleable\_map, r\_class\_proguard\_map):  
 resource\_def\_set = set({})  
 try:  
 if file\_util.is\_readable(smali\_path):  
 *smali*\_file = open(smali\_path, **"r"**)  
 *smali*\_line = smali\_file.readline().strip()  
 while smali\_line:  
 line = smali\_line.lstrip(**'**\t**'**)  
 if line and line.strip().startswith(**"const "**):  
 columns = line.split(**","**)  
 if len(columns) == 2:  
 res\_id = \_\_parse\_resource\_id(columns[1].strip())  
 if res\_id and (res\_id in resource\_def\_map.keys()):  
 # print(resource\_def\_map[res\_id])  
 resource\_def\_set.add(resource\_def\_map[res\_id])  
 elif line.startswith(**"***sget* **"**):

# styleable资源引用在smali代码中呈现  
 .....

smali\_line: = smali\_file.readline()  
 except Exception as e:  
 raise Exception(**"read\_***smali***\_files Error,"**, e)  
 return resource\_def\_set

# 遍历res/目录下xml文件  
def decode\_resources(apk\_path\_dir, res\_guard\_map):  
 non\_value\_reference\_map = dict({})  
 resource\_res\_used\_set = set({})  
 if not apk\_path\_dir.endswith(**"/"**):  
 apk\_path\_dir += **"/"** res\_dir = apk\_path\_dir+**"res/"** if not os.path.exists(res\_dir):  
 res\_dir = apk\_path\_dir + **'r/'** for paths, sub\_paths, files in os.walk(res\_dir):  
 path\_dirs = paths.split(**"/"**)  
 resource\_dir = path\_dirs[len(path\_dirs)-1]  
 res\_type = str(resource\_dir).split(**'-'**)[0]  
 if res\_type and **"values"** in res\_type:  
 for file in files:  
 # 解析values下xml文件,并返回引用资源ID列表  
 value\_references\_set = xml\_decoder(os.path.join(paths, file), res\_guard\_map)  
 for resource in value\_references\_set:  
 resource\_res\_used\_set.add(resource)  
 elif res\_type:  
 for file in files:  
 if file\_util.is\_legal\_file(os.path.join(paths, file)) and file.endswith(**".xml"**) \  
 and not ignore\_resource(os.path.join(paths, file)):  
 # 非values xml引用res资源  
 res\_used\_set = xml\_decoder(os.path.join(paths, file), res\_guard\_map)  
 resource = **"R."** + str(res\_type) + **"."** + file[:file.rfind(**"."**)]  
 if resource in res\_guard\_map.keys():  
 before\_resource = res\_guard\_map[resource].split(**"R."**+res\_type+**"."**)[1].replace(**'.'**, **'\_'**)  
 res\_guard\_resource = **"R."**+res\_type+**"."** + before\_resource  
 resource = res\_guard\_resource  
 if resource in non\_value\_reference\_map.keys():  
 reference\_set = non\_value\_reference\_map.get(resource)  
 for item in res\_used\_set:  
 reference\_set.add(item)  
 non\_value\_reference\_map[resource] = reference\_set  
 else:  
 non\_value\_reference\_map[resource] = res\_used\_set

# AndroidManifest.xml文件读取,并返回引用的资源ID列表  
 manifest\_path = apk\_path\_dir + **"AndroidManifest.xml"** if not file\_util.is\_legal\_file(manifest\_path):  
 logging.warning(**"File %s is illegal!"** % manifest\_path)  
 return  
 manifest\_ref\_set = xml\_decoder(manifest\_path, res\_guard\_map)  
 for reference in manifest\_ref\_set:  
 resource\_res\_used\_set.add(reference)  
 return resource\_res\_used\_set, non\_value\_reference\_map

### assets目录无用资源分析

经过分析AAPT打包APK流程可知，assets/下文件直接被保留至APK中，所以DEX可以直接使用名称进行访问

assets/目录下无用资源分析流程：

1. 查找assets目录下所有文件，并将文件路径保存至set<asset\_id> assets\_path\_set中
2. 遍历smali代码，查找代码中引用的assets资源，并将查找结果放入set<asset\_id> asset\_ref\_set中
3. assets\_path\_set去除asset\_ref\_set剩余的资源，都是未被使用的

核心代码：

def find\_asset\_file(asset\_dir):

if asset\_dir and os.path.exists(asset\_dir) and os.path.isdir(asset\_dir):  
 for paths, sub\_paths, files in os.walk(asset\_dir):  
 for file in files:  
 asset\_file\_sub\_dir = paths.split(asset\_dir)[1]  
 # assets目录下图片资源均放入unused\_asset\_set  
 unused\_asset\_set.add(os.path.join(asset\_file\_sub\_dir, file))  
  
  
# 遍历*smali*代码,获取引用资源的res\_id  
def read\_smali\_files(*smali*\_path):  
 if file\_util.is\_readable(smali\_path):  
 smali\_file = open(smali\_path, "r")  
 smali\_line = smali\_file.readline()  
 while smali\_line:  
 line = smali\_line.strip()  
 if line and line.startswith("const"):  
 # smali代码中引用资源ID  
 do something...  
 elif line.startswith("sget"):  
 # smali代码中引用资源ID数组,例如int[] styleable  
 do something...  
 elif line.startswith("const-string"):  
 # 查找smali代码中引用asset资源  
 columns = line.split(",")  
 if len(columns) == 2:  
 asset\_file\_name = columns[1].strip()  
 if asset\_file\_name:  
 for path in unused\_res\_set:  
 if path.endswith(asset\_file\_name):  
 unused\_res\_set.remove(path)  
 *smali*\_line = smali\_file.readline()

注意:无用资源分析，只会分析Java代码中使用的资源，像React Native代码中引用资源将不会被扫描到，后期如果允许在对相关部分进行研究

1. **结语**

使用Python我们快速实现了Android APK 的资源分析，为应用瘦身提供准确的数据支持，同时也开发了图片压缩、资源混淆、资源托管线上等工具。目前分析平台已搭建完成并进入内测阶段，期望更高效地为应用瘦身提供支持，让APK体积将到极致，降低应用分发成本，提升转化率及用户体验。