

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。

2. 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：

```
Lynnma@Lynnma-Vostro-3668:~/software/android/Sdk/build-tools/23.0.2$ aapt l /home/lynnma/Desktop/JDMALL-V8.3.4.apk
classes6.dex
builddef.lst
classes.dex
com/sina/weibo/sdk/net/cacert_com.cer
com/sina/weibo/sdk/net/cacert_cn.cer
com/jingdong/common/readme.md
com/jingdong/common/utils/PersonalDesCommonUtils.kt
classes3.dex
classes4.dex
AndroidManifest.xml
lib/armeabi-v7a/libcom.jd.lib.jdpaycommon.so
lib/armeabi-v7a/libyoga.so
lib/armeabi-v7a/libjdface.so
lib/armeabi-v7a/libSecurity.so
lib/armeabi-v7a/libphccommon-lib.so
lib/armeabi-v7a/libmlsdkhelper.so
```

简单归类如下：

文件目录	描述
lib/	存放so文件, 可能会有armeabi, armeabi-v7a, arm64-v8a, x86, x86_64, mips, 大多数情况下只需要支持armeabi与x86的框架即可, 如果非必须可以考虑只保留armeabi
res/	存放编译后的资源文件, 例如:drawable, layout, anim等等
assets/	应用程序的资源, res资源会映射到R.txt文件中, 每一个资源都有属于自己独一无二的ID, 而assets文件可以直接通过访问文件地址来使用AssetManager进行访问
META-INF/	该文件夹一般存放已经签名信息, 用来保证apk包的完整性和系统安全. 没有签名的app是不被系统认可且无法安装到手机中
classes(n).dex	classes文件是Java编译后的生成的字节码文件
resources.arsc	编译后二进制资源文件索引, 记录了资源文件(res目录中文件)和资源文件ID的映射关系, 保证程序运行时可以根据资源ID获取到形影的资源
AndroidManifest.xml	Android应用的配置清单文件, 它描述了每个应用的name, version, permission, 以及Android四大组件注册等内容

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

3. 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。

3.1. 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:
```

3.2. 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
```

3.3. 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
```

3.4. 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

```

3.5. 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.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

3.6. 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。

3.7. 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。

3.7.1. 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

```

3.7.2. 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

4. 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。