实例分析.dex 文件格式 -- 以 hello.dex 为例

月蓝
2014.02.26

前言

最近在学习 dex 的文件格式，阅读非虫大大的《Android 软件安全与逆向分析》里 dex 的章节内容，其实信息量蛮大的。读过一遍后，觉得对 dex 的文件格式的学习还少点什么。下一章 odex 内容看了几行后，这种不安全的感觉愈发变得强烈起来，会想起“空中楼阁”的寓言。决定自己写一个可执行的小程序，编译成.dex 格式，分析过一遍后再往下进行书的内容。觉得这东西总归是基础功，属于必须熟悉的那一类。

分析的过程尽量不去翻书，尽量模拟原生的分析过程。一些结构的定义，多有参考 Android Open Source Project 上的在线文档。

参考的官网文档：
Dalvik Executable Format ： http://source.android.com/devices/tech/dalvik/dex-format.html

一方面反刍已经学到的东西，一方面把分析的过程里记录的东西整理出来。感觉信息量巨大，不记录下来用脑袋肯定记忆不住。最初看书的时候是用笔在纸上写写画画的，后来借助了 office 表格做标记，所以也有些图表出来。

结束的时候，把文字、代码和表格 ，所有东西整理出来以后，发现有 19 页纸张的大小 。若发帖子的话 ，长度肯定会比李咏的脸都长，这发帖后怎么阅读啊。最后决定整理成文档做附件。.dex 格式的数据结构和描述知识 Android Open Source Project 都公开出来了，分析思路非虫大大书里面描述的也很详细，没有新添加的东西。按照自己对 .dex 格式和官方对应文档的理解，推理汇整了一下，作为学习.dex 格式的一个小结。文档的目录如下 ：

Table of Contents

第一部分：创造一个可供分析的 Hello.dex

Windows 操作系统也有五花八门的应用 ，可是想当年学习编程语言的第一个程序，却是在 Turbo C 蓝漆漆的界面里笨拙地敲一个 Hello World 程序 ，然后又在黑漆漆的命令行里打印出的一行输出结果 ，“Hello World”。

Android 上也能实现类似的效果 ，可以使用 dalvikvm 执行一个运行于命令行的 Hello World 。

简要的过程如下：
写代码 Hello.java
编译成 Android Dalvik Virtual Machine 的可执行文件 Hello.dex
使用 ADB 运行测试
内容主要参考 Android 4.4 源码里 dalvik / docs / hello-world.html 文件 。

1. 测试环境

Ubuntu 12.04 64-bit
JDK 1.6
安装 Android SDK，并安装 Android SDK build_tools , 最新版是 19.0.1
具有 Root 权限的 ADB shell

2. java 源码和编译方法

创建 java 源文件 ，内容如下
代码:

```
public class Hello
{
  public static void main(String[] argc)
  {
    System.out.println("Hello, Android!\n");
  }
}
```

在当前工作路径下 ， 编译方法如下 ：
（1）编译成 java class 文件
执行命令 ： javac Hello.java
编译完成后 ，目录下生成 Hello.class 文件 。可以使用命令 java Hello 来测试下 ，会输出代码中的 “Hello， Android ! ” 的字符串 。
（2） 编译成 dex 文件

编译工具在 Android SDK 的路径如下 ，其中 19.0.1 是 Android SDK build_tools 的版本 ，请按照在本地安装的 build_tools 版本来 。建议该路径加载到 PATH 路径下 ，否则引用 dx 工具时需要使用绝对路径 。

./build-tools/19.0.1/dx
执行命令 ： dx --dex --output=Hello.dex Hello.class
编译正常会生成 Hello.dex 文件 。

3. 使用 ADB 运行测试

测试命令和输出结果如下 ：

```
$ adb root
$ adb push Hello.dex  /sdcard/
$ adb shell
root@maguro:/ # dalvikvm -cp /sdcard/Hello.dex Hello
Hello, Android!
```

4. 重要说明

（1）测试环境使用真机和 Android 虚拟机都可以的 。核心的命令是
dalvikvm -cp /sdcard/Hello.dex Hello
-cp 是 class path 的缩写 ，后面的 Hello 是要运行的 Class 的名称 。网上有描述说输入 dalvikvm --help 可以看到 dalvikvm 的帮助文档 ，但是在 Android4.4 的官方模拟器和自己的手机上测试都提示找不到 Class 路径 ，在 Android 老的版本 （ 4.3) 上测试还是有输出的 。
（2）因为命令在执行时 ， dalvikvm 会在 /data/dalvik-cache/ 目录下创建 .dex 文件 ，因此要求 ADB 的执行 Shell 对目录 /data/dalvik-cache/ 有读、写和执行的权限 ，否则无法达到预期效果 。
// added stop

对于生成的 Hello.dex 在 linux 下使用 vi -b Hello.dex 命令打开它，此命令必须添加-b 选项，否则在输入下一个命令时，会有丢失行信息的提示。

然后在命令模式下输入 :%!xxd 命令就可以转化为 2 进制的表示方式。

第二部分：分析过程

1. dex 整个文件的布局

逻辑上，可以把 dex 文件分成 3 个区，文件头、索引区和数据区,索引区的 ids 后缀为 identifiers 的缩写,意思是某件东西的识别码。

1. header // 文件头

2. string_ids // string 索引
3. type_ids // type 索引
4. proto_ids // method prototype 索引
5. field_ids // field 索引
6. method_ids // method 索引

7. class_defs // class 定义
8. data // 数据
9. link_data // 静态连接文件

2. header

dex 文件里的 header。除了描述.dex 文件的文件信息外，还有文件里其它各个区域的索引。header 对应成结构体类型，逻辑上的描述我用结构体 header_item 来理解它。先给出结构体里面用到的数据类型 ubyte 和 uint 的解释，然后再是结构体的描述，后面对各种结构描述的时候也是用的这种方法。

ubyte 8-bit unsinged int
uint 32-bit unsigned int, little-endian;

```
struct header_item
{
    ubyte[8]    magic;
    unit        checksum;
    ubyte[20]   siganature;
    uint        file_size;
    uint        header_size;
    unit        endian_tag;
    uint        link_size;
    uint        link_off;
    uint        map_off;
```

```
    uint        string_ids_size;
    uint        string_ids_off;
    uint        type_ids_size;
    uint        type_ids_off;
    uint        proto_ids_size;
    uint        proto_ids_off;
    uint        method_ids_size;
    uint        method_ids_off;
    uint        class_defs_size;
    uint        class_defs_off;
    uint        data_size;
    uint        data_off;
}
```

里面的元素里，除了蓝色字体标志的，都是一对一对以_size 和_off 为后缀，_size 表示对应区块里元素的个数或大小，off 表示距离文件起始位置的偏移量,是 offset 的略写。蓝色色字体标志的，多是描述该.dex 文件信息的内容.

Header 的大小固定为 0x70,偏移地址从 0x00 到 0x70,提取信息如下：

```
0000000: 6465 780a 3033 3500 175e 36c1 b501 e2db  dex.035..^6.....
0000010: 7635 4d97 1289 c008 30b1 506a 7512 4cfb  v5M.....0.Pju.L.
0000020: e402 0000 7000 0000 7856 3412 0000 0000  ....p...xV4.....
0000030: 0000 0000 4402 0000 0e00 0000 7000 0000  ....D.......p...
0000040: 0700 0000 a800 0000 0300 0000 c400 0000  ................
0000050: 0100 0000 e800 0000 0400 0000 f000 0000  ................
0000060: 0100 0000 1001 0000 b401 0000 3001 0000  ............0...
```

根据结构体描述和 2 进制信息，整理出 header_item 的内容如下：

address	name	size / byte	value
0	Magic[8]	8	0x6465 780a 3033 3500
8	checksum	4	0xc136 5e17
C	Signature[20]	20	
20	file_size	4	0x02e4
24	header_size	4	0x70
28	endan_tag	4	0x12345678
2C	link_size	4	0x00
30	link_off	4	0x00
34	map_off	4	0x0244
38	string_ids_size	4	0x0e
3C	string_ids_off	4	0x70
40	type_ids_size	4	0x07
44	type_ids_off	4	0xa8
48	proto_ids_size	4	0x03
4C	proto_ids_off	4	0xc4
50	field_ids_size	4	0x01
54	field_ids_off	4	0xe8
58	method_ids_size	4	0x04
5C	method_ids_off	4	0xf0
60	class_defs_size	4	0x01
64	class_defs_off	4	0x0110
68	data_size	4	0x01b4
6C	data_off	4	0x0130

里面一对一对以_size 和_off 为后缀的描述：data_size 是以 Byte 为单位描述 data 区的大小，其余的_size 都是描述该区里元素的个数；_off 描述相对与文件起始位置的偏移量。其余的 6 个是描述.dex 文件信息的，各项说明如下：

(1) magic value

这 8 个 字节一般是常量 ，为了使 .dex 文件能够被识别出来 ，它必须出现在 .dex 文件的最开头的位置 。数组的值可以转换为一个字符串如下 ：

{ 0x64 0x65 0x78 0x0a 0x30 0x33 0x35 0x00 }

= "dex\n035\0"

中间是一个 '\n' 符号 ，后面 035 是 Dex 文件格式的版本 。

（2）checksum 和 signature

文件校验码 ，使用 alder32 算法校验文件除去 maigc ，checksum 外余下的所有文件区域 ，用于检查文件错误 。

signature , 使用 SHA-1 算法 hash 除去 magic ,checksum 和 signature 外余下的所有文件区域 ，用于唯一识别本文件 。

（3）file_size

Dex 文件的大小 。

（4）header_size

header 区域的大小 ，单位 Byte ，一般固定为 0x70 常量 。

（5）endian_tag

大小端标签 ，标准 .dex 文件格式为 小端 ，此项一般固定为 0x1234 5678 常量 。

（6）map_off

map item 的偏移地址 ，该 item 属于 data 区里的内容 ，值要大于等于 data_off 的大小 。结构如 map_list 描述 。

定义位置 ： data 区

引用位置 ：header 区 。

ushort 16-bit unsigned int, little-endian.
uint 32-bit unsigned int, little-endian.

```
struct maplist
{
        uint            size;
        map_item        list [size];
}

struct map_item
{
        ushort type;
        ushort unuse;
        uint    size;
        uint offset;
}
```

map_list 里先用一个 uint 描述后面有 size 个 map_item , 后续就是对应的 size 个 map_item 描述 。map_item 结构有 4 个元素 ： type 表示该 map_item 的类型 ，本节能用到的描述如下 ，详细 Dalvik Executable Format 里 Type Code 的定义 ；size 表示再细分此 item ， 该类型的个数 ；offset 是第一个元素的针对文件初始位置的偏移量 ； unuse 是用对齐字节的 ，无实际用处 。

Item Type	Constant	Value	Item Size In Bytes
header_item	TYPE_HEADER_ITEM	0x0000	0x70
string_id_item	TYPE_STRING_ID_ITEM	0x0001	0x04
type_id_item	TYPE_TYPE_ID_ITEM	0x0002	0x04
proto_id_item	TYPE_PROTO_ID_ITEM	0x0003	0x0c
field_id_item	TYPE_FIELD_ID_ITEM	0x0004	0x08
method_id_item	TYPE_METHOD_ID_ITEM	0x0005	0x08
class_def_item	TYPE_CLASS_DEF_ITEM	0x0006	0x20
map_list	TYPE_MAP_LIST	0x1000	4 + (item.size * 12)
type_list	TYPE_TYPE_LIST	0x1001	4 + (item.size * 2)
annotation_set_ref	TYPE_ANNOTATION_SET_REF	0x1002	4 + (item.size * 4)
annotation_set_ite	TYPE_ANNOTATION_SET_ITEM	0x1003	4 + (item.size * 4)
class_data_item	TYPE_CLASS_DATA_ITEM	0x2000	implicit; must parse
code_item	TYPE_CODE_ITEM	0x2001	implicit; must parse
string_data_item	TYPE_STRING_DATA_ITEM	0x2002	implicit; must parse
debug_info_item	TYPE_DEBUG_INFO_ITEM	0x2003	implicit; must parse
annotation_item	TYPE_ANNOTATION_ITEM	0x2004	implicit; must parse
encoded_array_ite	TYPE_ENCODED_ARRAY_ITEM	0x2005	implicit; must parse
annotations_direct	TYPE_ANNOTATIONS_DIRECT	0x2006	implicit; must parse

header->map_off = 0x0244 , 偏移为 0244 的位置值为 0x 000d 。

每个 map_item 描述占用 12 Byte ， 整个 map_list 占用 12 * size + 4 个字节 。所以整个 map_list 占用空间为 12 * 13 + 4 = 160 = 0x00a0 ， 占用空间为 0x 0244 ~ 0x 02E3 。从文件内容上看 ，也是从 0x 0244 到文件结束的位置 。

```
0000240: c802 0000 0d00 0000 0000 0000 0100 0000  ................
0000250: 0000 0000 0100 0000 0e00 0000 7000 0000  ............p...
0000260: 0200 0000 0700 0000 a800 0000 0300 0000  ................
0000270: 0300 0000 c400 0000 0400 0000 0100 0000  ................
0000280: e800 0000 0500 0000 0400 0000 f000 0000  ................
0000290: 0600 0000 0100 0000 1001 0000 0120 0000  ............. ..
00002a0: 0200 0000 3001 0000 0110 0000 0200 0000  ....0...........
00002b0: 6801 0000 0220 0000 0e00 0000 7601 0000  h.... ......v...
00002c0: 0320 0000 0200 0000 2802 0000 0020 0000  . ......(.... ..
00002d0: 0100 0000 3402 0000 0010 0000 0100 0000  ....4...........
00002e0: 4402 0000                                D...
```

地址 0x0244 的一个 uinit 的值为 0x0000 000d ，map_list - > size = 0x0d = 13 ，说明后续有 13 个 map_item 。根据 map_item 的结构描述在 0x0248 ~ 0x02e3 里的值 ，整理出这段二进制所表示的 13 个 map_item 内容 ，汇成表格如下 ：

index	address	type	size	offset	types' meaning	offset in header	name in header
1	0x0248	0x00	0x01	0x00	TYPE_HEADER_ITEM	0x00	
2	0x0254	0x0001	0x0e	0x70	TYPE_STRING_ID_ITEM	0x70	string_ids_off
3	0x0260	0x0002	0x07	0x00a8	TYPE_TYPE_ID_ITEM	0xa8	type_ids_off
4	0x026C	0x0003	0x03	0x00c4	TYPE_PROTO_ID_ITEM	0xc4	proto_ids_off
5	0x0278	0x0004	0x01	0x00e8	TYPE_FIELD_ID_ITEM	0xe8	field_ids_off
6	0x0284	0x0005	0x0004	0x00f0	TYPE_METHOD_ID_ITEM	0xf0	method_ids_off
7	0x0290	0x0006	0x0001	0x0110	TYPE_CLASS_DEF_ITEM	0x0110	class_defs_off
8	0x029C	0x2001	0x0002	0x0130	TYPE_CODE_ITEM	0x0130	data_off
9	0x02A8	0x1001	0x0002	0x0168	TYPE_TYPE_LIST		
10	0x02B4	0x2002	0x000e	0x0176	TYPE_STRING_DATA_ITEM		
11	0x02C0	0x2003	0x0002	0x0228	TYPE_DEBUG_INFO_ITEM		
12	0x02CC	0x2000	0x0001	0x0234	TYPE_CLASS_DATA_ITEM		
13	0x02d8	0x1000	0x0001	0x0244	TYPE_MAP_LIST		

map_list - > map_item 里的内容 ，有部分 item 跟 header 里面相应 item 的 offset 地址描述相同 。但 map_list 描述的更为全面些 ，又包括了 HEADER_ITEM , TYPE_LIST , STRING_DATA_ITEM 等 ，最后还有它自己 TYPE_MAP_LIST 。

至此 ， header 部分描述完毕 ，它包括描述 .dex 文件的信息 ，其余各索引区和 data 区的偏移信息 ， 一个 map_list 结构 。map_list 里除了对索引区和数据区的偏移地址又一次描述 ，也有其它诸如 HEAD_ITEM , DEBUG_INFO_ITEM 等信息 。

3. string_ids

string_ids 区索引了 .dex 文件所有的字符串 。 本区里的元素格式为 string_ids_item , 可以使用结构体如下描述 。

uint , 32-bit unsigned int , little-endian

```
struct string_ids_item
{
    uint     string_data_off;
}
```

以 _ids 结尾的各个 section 里放置的都是对应数据的偏移地址 ，只是一个索引 ，所以才会在 .dex 文件布局里把这些区归类为 “索引区” 。

string_data_off 只是一个偏移地址 ，它指向的数据结构为 string_data_item

uleb128 : unsigned LEB128, valriable length
ubyte: 8-bit unsinged int

```
struct string_data_item
{
    uleb128        utf16_size;
    ubyte          data;
}
```

上述描述里提到了 LEB128 （ little endian base 128 ）格式 ，是基于 1 个 Byte 的一种不定长度的编码方式 。若第一个 Byte 的最高位为 1 ，则表示还需要下一个 Byte 来描述 ，直至最后一个 Byte 的最高位为 0 。每个 Byte 的其余 Bit 用来表示数据 ，如下表所示 。

a two-byte LEB128 value															
First byte								Second byte							
1	bit6	bit5	bit4	bit3	bit2	bit1	bit0	0	bit13	bit12	bit11	bit10	bit9	bit8	bit7

根据 string_ids_item 和 string_data_item 的描述 ，加上 header 里提供的入口位置 string_ids_size = 0x0e , string_ids_off = 0x70 ，我们可以整理出 string_ids 及其对应的数据如下 ：

.dex 里 string_ids_item 的二进制

```
0000070: 7601 0000 7e01 0000 9001 0000 9c01 0000  v...~...........
0000080: a501 0000 bc01 0000 d001 0000 e401 0000  ................
0000090: f801 0000 fb01 0000 ff01 0000 1402 0000  ................
00000a0: 1a02 0000 1f02 0000 0300 0000 0400 0000  ................
```

.dex 里 string_data_item 的二进制

```
0000170: 0100 0000 0600 063c 696e 6974 3e00 1048        .......<init>..H
0000180: 656c 6c6f 2c20 416e 6472 6f69 6421 0a00        ello, Android!..
0000190: 0a48 656c 6c6f 2e6a 6176 6100 074c 4865        .Hello.java..LHe
```

```
00001a0: 6c6c 6f3b 0015 4c6a 6176 612f 696f 2f50    llo;..Ljava/io/P
00001b0: 7269 6e74 5374 7265 616d 3b00 124c 6a61    rintStream;..Lja
00001c0: 7661 2f6c 616e 672f 4f62 6a65 6374 3b00    va/lang/Object;.
00001d0: 124c 6a61 7661 2f6c 616e 672f 5374 7269    .Ljava/lang/Stri
00001e0: 6e67 3b00 124c 6a61 7661 2f6c 616e 672f    ng;..Ljava/lang/
00001f0: 5379 7374 656d 3b00 0156 0002 564c 0013    System;..V..VL..
0000200: 5b4c 6a61 7661 2f6c 616e 672f 5374 7269    [Ljava/lang/Stri
0000210: 6e67 3b00 046d 6169 6e00 036f 7574 0007    ng;..main..out..
0000220: 7072 696e 746c 6e00 0100 070e 0005 0100    println.........
```

string_ids_item 和 string_data_item 里提取出的对应数据表格 ：

index	string_data_off	utf16_size	ubyte	string
1	0x0176	0x06	0x3c 69 6e 69 74 3e 00	<init>
2	0x017e	0x10	0x48 65 6c 6c 6f 2c 20 41 6e 64 72 6f 69 64 21 0a 00	Hello, Android!
3	0x0190	0x0a	0x48 65 6c 6c 6f 2e 6a 61 76 61 00	Hello.java
4	0x019c	0x07	0x4c 48 65 6c 6c 6f 3b 00	LHello;
5	0x01a5	0x15	0x4c 6a 61 76 61 2f 69 6f 2f 50 72 69 6e 74 53 74 72 65 61 6d 3b 00	Ljava/io/PrintStream;
6	0x01bc	0x12	0x4c 6a 61 76 61 2f 6c 61 6e 67 2f 4f 62 6a 65 63 74 3b 00	Ljava/lang/Object;
7	0x01d0	0x12	0x4c 62 61 76 61 2f 6c 61 6e 67 2f 53 74 72 69 6e 67 3b 00	Ljava/lang/String;
8	0x01e4	0x12	0x4c 6a 61 76 61 2f 6c 61 6e 67 2f 53 79 73 74 65 6d 3b 00	Ljava/lang/System;
9	0x01f8	0x01	0x56 00	V
10	0x01fb	0x02	0x56 4c 00	VL
11	0x01ff	0x13	0x5b 4c 6a 61 76 61 2f 6c 61 6e 67 2f 53 74 72 69 6e 67 3b 00	[Ljava/lang/String;
12	0x0214	0x04	0x6d 61 69 6e 00	main
13	0x021a	0x03	0x6f 75 74 00	out
14	0x021f	0x07	0x70 72 69 6e 74 6c 6e 00	println

string 里的各种标志符号 ，诸如 L ， V ， VL , [等在 .dex 文件里有特殊的意思 ，参考 dex-format.html 里的 String Syntax 章节 。

string_ids 的终极奥义就是找到这些字符串 。其实使用二进制编辑器打开 .dex 文件时 ，一般工具默认翻译成 ASCII 码 ，总会一大片熟悉的字符白生生地很是亲切， 也很是晃眼 。刚才走过的一路子分析流程 ，就是顺藤摸瓜找到它们是怎么来的。以后的一些 type-ids , method_ids 也会引用到这一片熟悉的字符串 。

4. type_ids

type_ids 区索引了 .dex 文件里的所有数据类型 ，包括 class 类型 ，数组类型（array types）和基本类型 (primitive types) 。 本区域里的元素格式为 type_ids_item , 结构描述如下 ：

```
struct type_ids_item
{
        uint     descriptor_idx;
}
```

type_ids_item 里面 descriptor_idx 的值的意思 ，是 string_ids 里的 index 序号 ，是用来描述此 type 的字符串 。

根据 header 里 type_ids_size = 0x07 , type_ids_off = 0xa8 , 找到对应的二进制描述区 。

```
00000a0: 1a02 0000 1f02 0000 0300 0000 0400 0000  ................
00000b0: 0500 0000 0600 0000 0700 0000 0800 0000  ................
00000c0: 0a00 0000 0800 0000 0500 0000 0000 0000  ................
```

根据 type_id_item 的描述 ，整理出表格如下 。因为 type_id_item - > descriptor_idx 里存放的是指向 string_ids 的 index 号 ，所以我们也能得到该 type 的字符串描述 。这里出现了 3 个 type descriptor :
L 表示 class 的详细描述 ，一般以分号表示 class 描述结束 ；
V 表示 void 返回类型 ，只有在返回值的时候有效 ；
[表示数组 ，[Ljava/lang/String; 可以对应到 java 语言里的 java.lang.String[] 类型 。

index	value	the String
0	0x03	LHello;
1	0x04	Ljava/io/PrintStream;
2	0x05	Ljava/lang/Object;
3	0x06	Ljava/lang/String;
4	0x07	Ljava/lang/System;
5	0x08	V
6	0x0a	[Ljava/lang/String;

5. proto_ids

proto 的意思是 method prototype 代表 java 语言里的一个 method 的原型 。proto_ids 里的元素为 proto_id_item , 结构如下 。

```
uint    32-bit unsigned int, little-endian
struct proto_id_item
{
        uint            shorty_idx;
        uint            return_type_idx;
        uint            parameters_off;
}
```

shorty_idx , 跟 type_ids 一样 ，它的值是一个 string_ids 的 index 号 ，最终是一个简短的字符串描述 ，用来说明该 method 原型 。
return_type_idx , 它的值是一个 type_ids 的 index 号 ，表示该 method 原型的返回值类型 。
parameters_off , 后缀 off 是 offset , 指向 method 原型的参数列表 type_list ; 若 method 没有参数 ，值为 0 。参数列表的格式是 type_list ，结构从逻辑上如下描述 。size 表示参数的个数 ；type_idx 是对应参数的类型 ，它的值是一个 type_ids 的 index 号 ，跟 return_type_idx 是同一个品种的东西 。

```
uint     32-bit unsigned int, little-endian
ushort  16-bit unsigned int, little-endian

struct type_list
{
        uint     size;
        ushort  type_idx[size];
}
```

header 里 proto_ids_size = 0x03 , proto_ids_off = 0xc4 , 它的二进制描述区如下 ：

```
00000c0: 0a00 0000 0800 0000 0500 0000 0000 0000  ................
00000d0: 0900 0000 0500 0000 6801 0000 0900 0000  ........h.......
00000e0: 0500 0000 7001 0000 0400 0100 0c00 0000  ....p...........
```

type_list 的二进制描述区如下 ：

```
0000160: 6e20 0200 1000 0e00 0100 0000 0300 0000  n ...............
0000170: 0100 0000 0600 063c 696e 6974 3e00 1048  .......<init>..H
```

根据 proto_id_item 和 type_list 的格式 ，对照这它们的二进制部分 ，整理出表格如下 ：

index	0	1	2
shorty_idx	0x08	0x09	0x09
return_type_idx	0x05	0x05	0x05
param_idx	0x00	0x0168	0x0170
shorty string	V	V	VL
return string	V	V	V

type_list->size	0x00	0x01	0x01
type_list->type_idx[0]		0x03	0x06
type_idx[0] string		Ljava/lang/String;	[Ljava/lang/String;

可以看出 ，有 3 个 method 原型 ，返回值都为 void ，index = 0 的没有参数传入 ，index = 1 的传入一个 String 参数 ，index=2 的传入一个 String[] 类型的参数 。

6. field_ids

filed_ids 区里面有被本 .dex 文件引用的所有的 field 。本区的元素格式是 field_id_item ，逻辑结构描述如

下 ：

```
ushort  16-bit unsigned int, little-endian
uint    32-bit unsigned int, little-endian
struct filed_id_item
{
        ushort          class_idx;
        ushort          type_idx;
        uint            name_idx;
}
```

class_idx , 表示本 field 所属的 class 类型 , class_idx 的值是 type_ids 的一个 index ， 并且必须指向一个 class 类型 。
type_idx , 表示本 field 的类型 ，它的值也是 type_ids 的一个 index 。
name_idx , 表示本 field 的名称 ，它的值是 string_ids 的一个 index 。

header 里 field_ids_size = 1 , field_ids_off = 0xe8 。说明本 .dex 只有一个 field ，这部分的二进制描述如下 ：
00000e0: 0500 0000 7001 0000 0400 0100 0c00 0000 ....p.....

filed_ids 只有一个一个元素 ，比较简单 。根据 fields_ids 的格式 ，整理出表格如下 。它是 Java 最常用的 System.out 标准输出部分 。

index	0
class_idx	0x04
type_idx	0x01
name_idx	0x0c
class string	Ljava/lang/System;
type string	Ljava/io/PrintStream;
name string	out

7. method_ids

method_ids 是索引区的最后一个条目 ，欢呼吧 ！！它索引了 .dex 文件里的所有的 method. method_ids 的元素格式是 method_id_item ， 结构跟 fields_ids 很相似 :

```
ushort  16-bit unsigned int, little-endian
uint    32-bit unsigned int, little-endian
struct filed_id_item
{
        ushort          class_idx;
        ushort          proto_idx;
        uint            name_idx;
}
```

class_idx , 和 name_idx 跟 fields_ids 是一样的 。
class_idx , 表示本 method 所属的 class 类型 , class_idx 的值是 type_ids 的一个 index ， 并且必须指向一个 class 类型 。
name_idx , 表示本 method 的名称 ，它的值是 string_ids 的一个 index 。
proto_idx 描述该 method 的原型 ，指向 proto_ids 的一个 index 。

header 里 method_ids_size = 0x04 , method_ids_off = 0xf0 。本部分的二进制描述如下 ：
00000f0: 0000 0000 0000 0000 0000 0200 0b00 0000 ....
0000100: 0100 0100 0d00 0000 0200 0000 0000 0000 ....

整理出表格如下 ：

index	0	1	2	3
class_idx	0x00	0x00	0x01	0x02
proto_idx	0x00	0x02	0x01	0x00
name_idx	0x00	0x0b	0x0d	0x00
class string	LHello;	LHello;	Ljava/io/PrintStream;	Ljava/lang/Object;
protostring	V()V	VL([Ljava/lang/String;)V	V(Ljava/lang/String;)V	V()V
name string	<init>	main	println	<init>

对 .dex 反汇编的时候 ，常用的 method 表示方法是这种形式 :
Lpackage/name/ObjectName;->MethodName(III)Z

将上述表格里的字符串再次整理下 ，method 的描述分别为 ：
0 : Lhello; -> <init>()V
1: LHello; -> main([Ljava/lang/String;)V
2: Ljava/io/PrintStream; -> println(Ljava/lang/String;)V
3 : Ljava/lang/Object; -> <init>()V

至此 ，索引区的内容描述完毕 ，包括 string_ids , type_ids , proto_ids , field_ids , method_ids 。每个索引区域里存放着指向具体数据的偏移地址 （如 string_ids ） ， 或者存放的数据是其它索引区域里面的 index 号。

8. class_defs

8.1 class_def_item

从字面意思解释 ，class_defs 区域里存放着 class definitions , class 的定义 。它的结构较 .dex 区都要复杂些 , 因为有些数据都直接指向了 data 区里面 。

class_defs 的数据格式为 class_def_item ， 结构描述如下 :

```
uint        32-bit unsigned int, little-endian
struct class_def_item
{
        uint        class_idx;
        uint        access_flags;
        uint        superclass_idx;
        uint        interfaces_off;
```

```
    uint       source_file_idx;
    uint       annotations_off;
    uint       class_data_off;
    uint       static_value_off;
}
```

(1) class_idx 描述具体的 class 类型 ，值是 type_ids 的一个 index 。值必须是一个 class 类型 ，不能是数组类型或者基本类型 。

(2) access_flags 描述 class 的访问类型 ，诸如 public , final , static 等 。在 dex-format.html 里 “access_flags Definitions” 有具体的描述 。

(3) superclass_idx , 描述 supperclass 的类型 ，值的形式跟 class_idx 一样 。

(4) interfaces_off , 值为偏移地址 ，指向 class 的 interfaces , 被指向的数据结构为 type_list 。class 若没有 interfaces ,值为 0。

(5) source_file_idx , 表示源代码文件的信息 ，值是 string_ids 的一个 index 。若此项信息缺失 ，此项值赋值为 NO_INDEX=0xffff ffff 。

(6) annotions_off , 值是一个偏移地址 ，指向的内容是该 class 的注释 ，位置在 data 区，格式为 annotations_direcotry_item 。若没有此项内容 ，值为 0 。

(7) class_data_off , 值是一个偏移地址 ，指向的内容是该 class 的使用到的数据 ，位置在 data 区，格式为 class_data_item 。若没有此项内容 ，值为 0 。该结构里有很多内容 ，详细描述该 class 的 field ， method , method 里的执行代码等信息 ，后面有一个比较大的篇幅来讲述 class_data_item 。

(8) static_value_off ， 值是一个偏移地址 ，指向 data 区里的一个列表 (list) ，格式为 encoded_array_item 。若没有此项内容 ，值为 0 。

header 里 class_defs_size = 0x01 , class_defs_off = 0x 0110 。则此段二进制描述为 ：

```
0000110: 0000 0000 0100 0000 0200 0000 0000 0000  ................
0000120: 0200 0000 0000 0000 3402 0000 0000 0000  ........4.......
```

根据对数据结构 class_def_item 的描述 ，整理出表格如下 ：

element	value	associated strinigs
class_idx	0x00	LHello;
access_flags	0x01	ACC_PUBLIC
superclass_idx	0x02	Ljava/lang/Object;
interface_off	0x00	
source_file_idx	0x02	Hello.java
annotations_off	0x00	
class_data_off	0x0234	
static_value_off	0x00	

其实最初被编译的源码只有几行 ，和 class_def_item 的表格对照下 ，一目了然 。

```
source file : Hello.java
public class Hello
{
```

```
    public static void main(String[] argc)
    {
        System.out.println("Hello, Android!\n");
    }
}
```

8.2 class_def_item --> class_data_item

class_data_off 指向 data 区里的 class_data_item 结构 ,class_data_item 里存放着本 class 使用到的各种数据 ，下面是 class_data_item 的逻辑结构 ：

```
uleb128  unsigned little-endian base 128
struct class_data_item
{
    uleb128                 static_fields_size;
    uleb128                 instance_fields_size;
    uleb128                 direct_methods_size;
    uleb128                 virtual_methods_size;

    encoded_field           static_fields [ static_fields_size ];
    encoded_field           instance_fields [ instance_fields_size ];
    encoded_method          direct_methods [ direct_method_size ];
    encoded_method          virtual_methods [ virtual_methods_size ];
}
```

关于元素的格式 uleb128 在 3. string_ids 里有讲述过 ，不赘述 。

encoded_field 的结构如下 ：

```
struct encoded_field
{
    uleb128         filed_idx_diff; // index into filed_ids for ID of this filed
    uleb128         access_flags; // access flags like public, static etc.
}
```

encoded_method 的结构如下 ：

```
struct encoded_method
{
    uleb128         method_idx_diff;
    uleb128         access_flags;
    uleb128         code_off;
```

}

（1）method_idx_diff，前缀 methd_idx 表示它的值是 method_ids 的一个 index ，后缀 _diff 表示它是于另外一个 method_idx 的一个差值 ，就是相对于 encodeed_method [] 数组里上一个元素的 method_idx 的差值 。其实 encoded_filed - > field_idx_diff 表示的也是相同的意思 ，只是编译出来的 Hello.dex 文件里没有使用到 class filed 所以没有仔细讲 ，详细的参考 dex_format.html 的官网文档 。

（2）access_flags ， 访问权限 ， 比如 public、private、static、final 等 。

（3） code_off， 一个指向 data 区的偏移地址 ，目标是本 method 的代码实现 。被指向的结构是 code_item ，有近 10 项元素 ，后面再详细解释 。

class_def_item -- > class_data_off = 0x 0234 。

0000230: 070e 7800 0000 0200 0081 8004 b002 0109 ..x.....
0000240: c802 0000 0d00 0000 0000 0000 0100 0000 ....

element	value
static_fields_size	0x00
instance_fields_size	0x00
directive_methods_size	0x02
vitual_methods_size	0x00
static_fields[]	
instance_fields[]	
directive_methods[]	{0x00, 0x81 80 04, 0xb0 02}, {0x01, 0x09, 0xc8 02}
vitual_methods[]	

名称为 LHello; 的 class 里只有 2 个 directive methods 。 directive_methods 里的值都是 uleb128 的原始二进制值 。按照 directive_methods 的格式 encoded_method 再整理一次这 2 个 method 描述 ，得到结果如下表格所描述 。method 一个是 <init>，一个是 main ， 两个都比较底层 ，详细的原理自行搜索 Java 语言的解释 ，此不赘述 。

directive_method	son-element	value	meaning
directive_method[0]	method_idx_diff	0x00	Lhello;-><init>()V
	access_flags	0x10001	ACC_PUBLIC | ACC_CONSTRUCTOR
	code_off	0x0130	
directive_method[1]	method_idx_diff	0x01	LHello;->main([Ljava/lang/String;)V
	access_flags	0x09	ACC_PUBLIC | ACC_STATIC
	code_off	0x0148	

8.3 class_def_item --> class_data_item --> code_item

到这里 ，逻辑的描述有点深入了 。我自己都有点分析不过来 ，先理一下是怎么走到这一步的 ，code_item 在 .dex 里处于一个什么位置 。

（1）一个 .dex 文件被分成了 9 个区 ，详细见 “1. dex 整个文件的布局 ” 。其中有一个索引区叫做 class_defs ， 索引了 .dex 里面用到的 class ，以及对这个 class 的描述 。Hello.dex 里只有一个 class – “

LHello; ”。

（2） class_defs 区 ， 这里面其实是 class_def_item 结构 。这个结构里描述了 LHello; 的各种信息 ，诸如名称 ，superclass , access flag， interface 等 。class_def_item 里有一个元素 class_data_off , 指向 data 区里的一个 class_data_item 结构 ，用来描述 class 使用到的各种数据 。自此以后的结构都归于 data 区了 。

（3）class_data_item 结构 ，里描述值着 class 里使用到的 static field , instance field , direct_method ，和 virtual_method 的数目和描述 。例子 Hello.dex 里 ，只有 2 个 direct_method , 其余的 field 和 method 的数目都为 0 。描述 direct_method 的结构叫做 encoded_method ，是用来详细描述某个 method 的 。

（4） encoded_method 结构 ，描述某个 method 的 method 类型 ， access flags 和一个指向 code_item 的偏移地址 ，里面存放的是该 method 的具体实现 。

（5） code_item ， 一层又一层 ，盗梦空间啊！简要的说 ，code_item 结构里描述着某个 method 的具体实现 。它的结构如下描述 ：

```
struct code_item
{
    ushort          registers_size;
    ushort          ins_size;
    ushort          outs_size;
    ushort          tries_size;
    uint            debug_info_off;
    uint            insns_size;
    ushort          insns [ insns_size ];
    ushort          paddding; // optional
    try_item        tries [ tyies_size ]; // optional
    encoded_catch_handler_list handlers; // optional
}
```

末尾的 3 项标志为 optional , 表示可能有 ，也可能没有 ，根据具体的代码来 。

(1) registers_size, 本段代码使用到的寄存器数目。

(2) ins_size, method 传入参数的数目 。

(3) outs_size, 本段代码调用其它 method 时需要的参数个数 。

(4) tries_size, try_item 结构的个数 。

(5) debug_off, 偏移地址 ，指向本段代码的 debug 信息存放位置 ，是一个 debug_info_item 结构。

(6) insns_size, 指令列表的大小 ，以 16-bit 为单位 。 insns 是 instructions 的缩写 。

(7) padding , 值为 0 ，用于对齐字节 。

(8) tries 和 handlers ， 用于处理 java 中的 exception , 常见的语法有 try catch 。

8.4 分析 main method 的执行代码并与 smali 反编译的结果比较

在 8.2 节里有 2 个 method , 因为 main 里的执行代码是自己写的 ，分析它会熟悉很多 。偏移地址是 directive_method [1] -> code_off = 0x0148 ，二进制描述如下 ：

0000140: 7010 0300 0000 0e00 0300 0100 0200 0000 p.....

0000150: 2d02 0000 0800 0000 6200 0000 1a01 0100 -.....b.....

0000160: 6e20 0200 1000 0e00 0100 0000 0300 0000 n .....

根据 code_item 的结构整理表格如下 ：

name	value
registers_size	0x03
ins_size	0x01
outs_size	0x02
tries_size	0x00
debug_info_off	0x022d
insns_size	0x08
insns	0x0062 0x0000 0x011a 0x0001 0x206e 0x0002 0x0010 0x000e

insns 数组里的 8 个二进制原始数据 ， 对这些数据的解析 ，需要对照官网的文档 《Dalvik VM Instruction Format》和《Bytecode for Dalvik VM》。

分析思路整理如下

（1）《Dalvik VM Instruction Format》 里操作符 op 都是位于首个 16bit 数据的低 8 bit ，起始的是 op = 0x62。

(2) 在 《Bytecode for Dalvik VM》 里找到对应的 Syntax 和 format 。

syntax = sget_object

format = 0x21c 。

（3） 在《Dalvik VM Instruction Format》里查找 21c ， 得知 op = 0x62 的指令占据 2 个 16 bit 数据 ，

格式是 AA|*op* BBBB ，解释为 op vAA, type@BBBB 。因此这 8 组 16 bit 数据里 ，前 2 个是一组 。对比数据得 AA=0x00, BBBB = 0x0000。

(4)返回《Bytecode for Dalvik VM》里查阅对 sget_object 的解释， AA 的值表示 Value Register ，即 0 号寄存器； BBBB 表示 static field 的 index ，就是之前分析的 field_ids 区里 Index = 0 指向的那个东西 ，当时的 fields_ids 的分析结果如下 ：

index	0
class_idx	0x04
type_idx	0x01
name_idx	0x0c
class string	Ljava/lang/System;
type string	Ljava/io/PrintStream;
name string	out

对 field 常用的表述是

包含 field 的类型 -> field 名称 ：field 类型 。

此次指向的就是 Ljava/lang/System; -> out:Ljava/io/printStream;

（5）综上 ，前 2 个 16 bit 数据 0x 0062 0000，解释为

sget_object v0, Ljava/lang/System; -> out:Ljava/io/printStream;

其余的 6 个 16 bit 数据分析思路跟这个一样 ，依次整理如下 ：

0x011a 0x0001:　　const-string v1, “Hello, Android!”

0x206e 0x0002 0x0010:

invoke-virtual {v0, v1}, Ljava/io/PrintStream; -> println(Ljava/lang/String;)V

0x000e: return-void

（6） 最后再整理下 main method , 用容易理解的方式表示出来就是 。

ACC_PUBLIC ACC_STATIC LHello;->main([Ljava/lang/String;)V

{

sget_object v0, Ljava/lang/System; -> out:Ljava/io/printStream;

const-string v1,Hello, Android!

invoke-virtual {v0, v1}, Ljava/io/PrintStream; -> println(Ljava/lang/String;)V

return-void

}

看起来很像 smali 格式语言 ，不妨使用 smali 反编译下 Hello.dex，看看 smali 生成的代码跟方才推导出来的有什么差异 。

```
.method public static main([Ljava/lang/String;)V
    .registers 3

    .prologue
    .line 5
    sget-object v0, Ljava/lang/System;->out:Ljava/io/PrintStream;

    const-string v1, "Hello, Android!\n"
```

```
    invoke-virtual {v0, v1}, Ljava/io/PrintStream;->println(Ljava/lang/String;)V

    .line 6
    return-void
```

从内容上看 ，二者形式上有些差异 ，但表述的是同一个 method 。这说明刚才的分析走的路子是没有跑偏的 。另外一个 method 是 <init>， 若是分析的话 ，思路和流程跟 main 一样 。走到这里，心里很踏实了。

这两天的分析过程 ，脑袋充满了各种 “ 因为... 所以... ” 和指向箭头 。就像小时候看动画片《名侦探柯南》，各种推理，心情随着故事情节起起伏伏。当最终对比自己的分析代码和生成的反汇编 smali 文件，并且对比结果是大家在表述同一个执行时，心里面很开心。对逆向分析的学习，才刚起了一个头，加油！