



一篇文章带你搞懂DEX文件的结构

转载

秦修  于 2018-12-19 17:05:19 发布  3126  收藏 4

分类专栏: [安卓逆向](#) 文章标签: [Dex文件结构](#) [Dex文件解析](#)

原文链接: https://blog.csdn.net/sinat_18268881/article/details/55832757

版权



[安卓逆向 专栏收录该内容](#)

4 篇文章 1 订阅

订阅专栏

DEX文件就是Android Dalvik虚拟机运行的程序，关于DEX文件的结构的重要性我就不多说了。下面，开练！

建议：不要只看，跟着我做。看再多遍不如自己亲自实践一遍来的可靠，别问我为什么知道。泪崩ing....

首先，我们需要自己构造一个dex文件，因为自己构造的比较简单，分析起来比较容易。等你简单的会了，难的自然也就懂了。

0x00■ 构造DEX文件

首先，我们编写一个简单的Java程序，如下：

```
public class HelloWorld {
    int a = 0;
    static String b = "HelloDalvik";
    public int getNumber(int i, int j) {
        int e = 3;
        return e + i + j;
    }
    public static void main(String[] args) {
        int c = 1;
        int d = 2;
        HelloWorld helloWorld = new HelloWorld();
        String sayNumber = String.valueOf(helloWorld.getNumber(c, d));
        System.out.println("HelloDex!" + sayNumber);
    }
}
```

然后将其编译成dex文件：打开命令行，进入HelloWorld.class所在文件夹下，执行命令：

```
javac HelloWorld.java
```

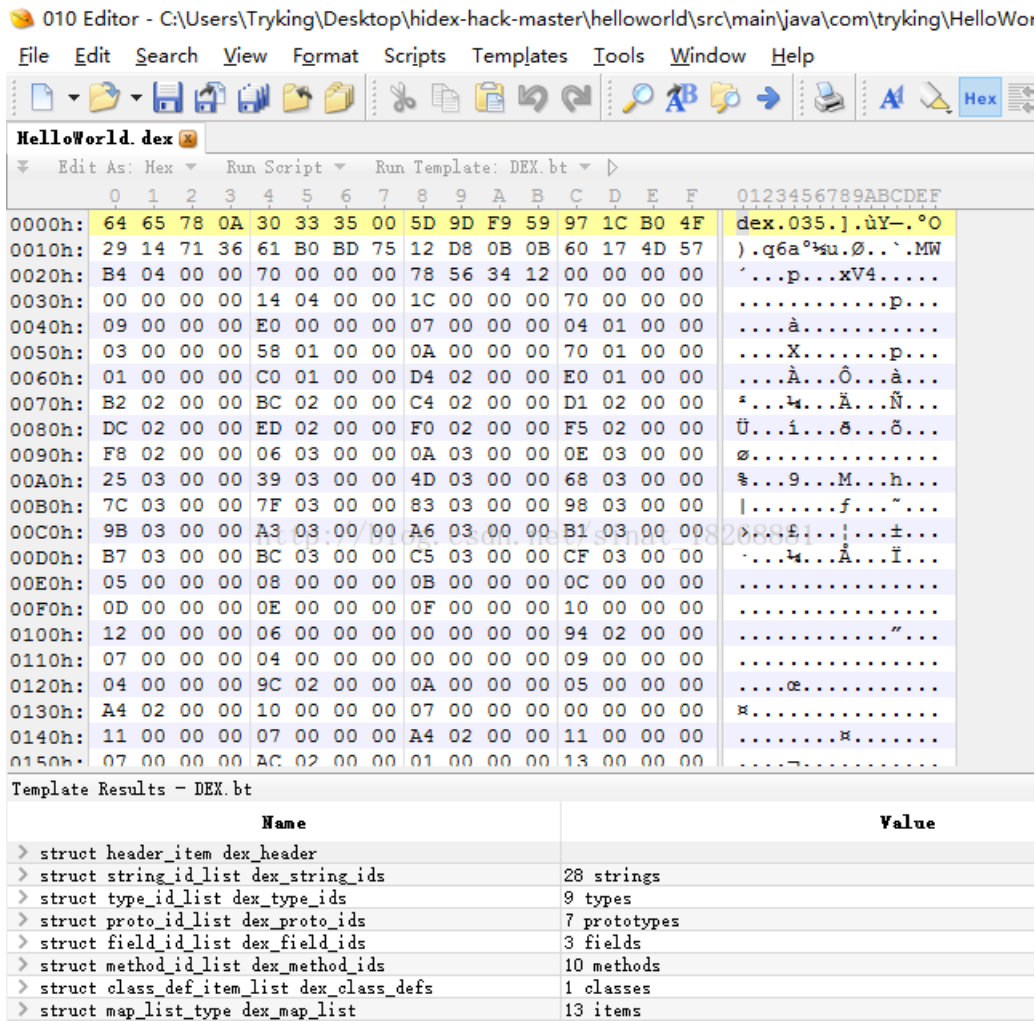
接下来会出现一个HelloWorld.class文件，然后继续执行命令：

```
dx --dex --output=HelloWorld.dex HelloWorld.class
```

我的dx工具所在的位置是：D:\Android\sdk\build-tools\23.0.3\dx --dex，因此该命令为：D:\Android\sdk\build-tools\23.0.3\dx --dex --output=HelloWorld.dex HelloWorld.class

就会出现HelloWorld.dex文件了。这时，我们需要下载一个十六进位文本编辑器，因为用它解析二进制文件，我们用它打开dex文件就会全部以十六进制的数进行展现了。这里推荐010Editor，下载地址：[010Editor](#)（收费软件，可以免费试用30天）。

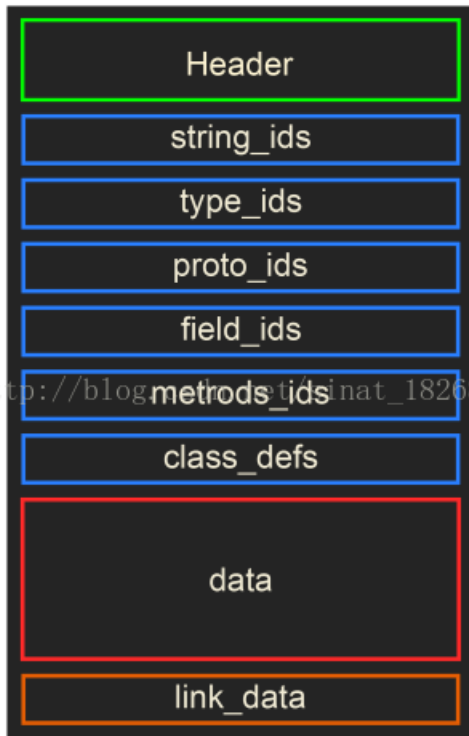
下载完成之后，我们可以用它打开dex文件了，打开之后，你的界面应该是这样的：



一下子看到这些东西，是不是立马懵逼了，正常，我刚开始看的时候也是，这什么玩意儿啊！其实，这就是二进制流文件中的内容，010Editor把它转化成了16进制的内容，以方便我们阅读的。

0x01 DEX文件结构总览

不要慌，下面我跟你解释，这些东西我们虽然看了懵逼，但是Dalvik虚拟机不会，因为它就是解析这些东西的，这些东西虽然看起来头大，但是它是有自己的格式标准的。dex文件的结构如下图所示：



http://blog.csdn.net/sinat_18268881

这就是dex的文件格式了，下面我们从最上面的Header说起，Header中存储了什么内容呢？下面我们还得来一张图：

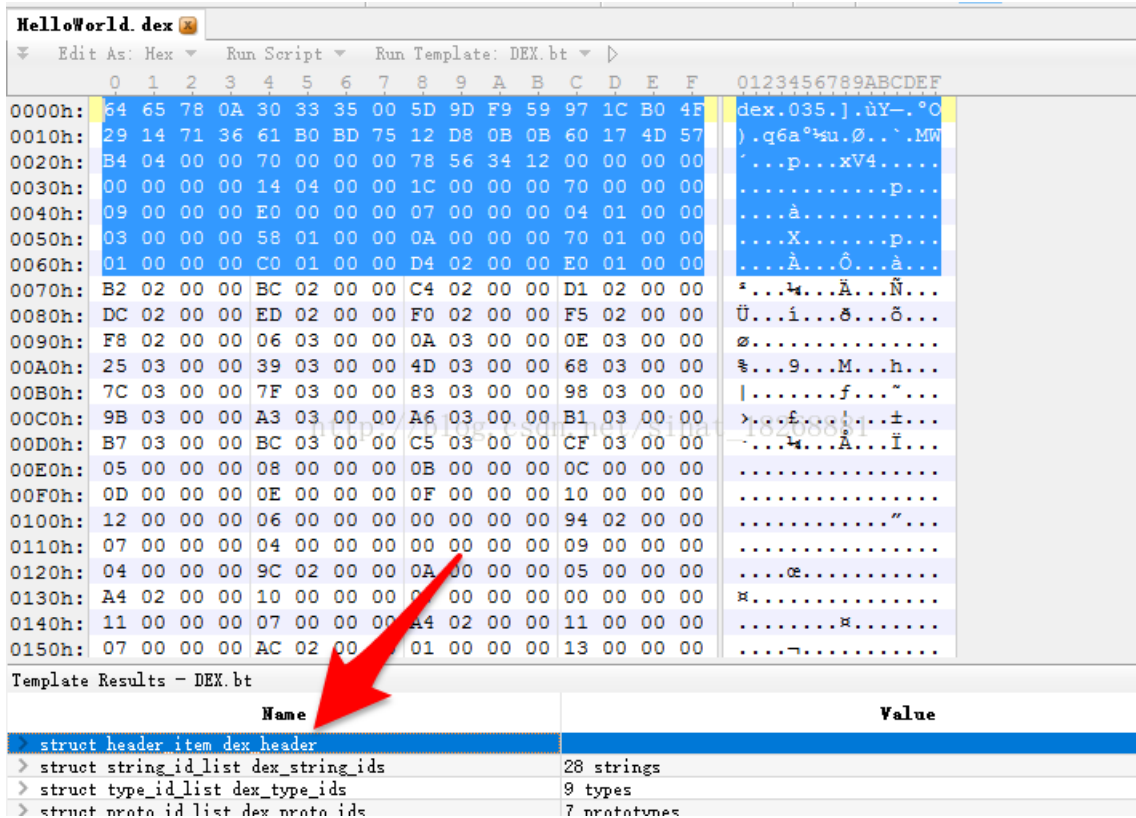
```

1 struct DexHeader{
2     u1 magic[8]; /* dex版本标识 */
3     u4 checksum; /* adler32检验 */
4     u1 signature[kSHA1DigestLen]; /* SHA-1 哈希值*/
5     u4 fileSize; /* 整个文件的大小*/
6     u4 headerSize; /* DexHeader结构大小*/
7     u4 endianTag; /* 字节序标记*/
8     u4 linkSize; /* 链接段大小*/
9     u4 linkOff; /* 链接段偏移*/
10    u4 mapOff; /* DexMapList的文件偏移*/
11    u4 stringIdsSize; /* DexStringId的个数*/
12    u4 stringIdsOff; /* DexStringId的文件偏移*/
13    u4 typeIdsSize; /* DexTypeId的个数*/
14    u4 typeIdsOff; /* DexTypeId的文件偏移*/
15    u4 protoIdsSize; /* DexProtoId的个数*/
16    u4 protoIdsOff; /* DexProtoId的文件偏移*/
17    u4 fieldIdsSize; /* DexFieldId的个数*/
18    u4 fieldIdsOff; /* DexFieldId的文件偏移*/
19    u4 methodIdsSize; /* DexMethodId的个数*/
20    u4 methodIdsOff; /* DexMethodId的文件偏移*/
21    u4 classDefsSize; /* DexClassDef的个数*/
22    u4 classDefsOff; /* DexClassDef的文件偏移*/
23    u4 dataSize; /* 数据段的大小 */
24    u4 dataOff; /* 数据段的文件偏移 */
25 }

```

0x02 DEX文件结构解析

先看下就行，不用着急，下面我们一步一步来，首先点击你的010Editor的这里：



010Editor interface showing a hex dump of a dex file. The hex dump displays memory addresses (0000h to 0150h) and their corresponding hexadecimal values. A red arrow points to the 'Template Results - DEX.bt' section at the bottom, which lists fields like 'struct header item dex header', 'struct string_id_list dex_string_ids', etc.

对，就是箭头指的那里，点击之后，你会发现上面的有一片区域成了选中的颜色，这部分里面存储的就是Header中的数据了，下面我们根据Header的数据图以此来进行分析。

首先，我们看到DexHeader中每个数据前面有个u1或者u4，这个是什么意思呢？它们其实就是代表1个或者4个字节的无符号数。下面我们依次根据Header中的数据段进行解释。

1. 从第一个看起，**magic[8]**；它代表dex中的文件标识，一般被称为魔数。是用来识别dex这种文件的，它可以判断当前的dex文件是否有效，可以看到它用了8个1字节的无符号数来表示，我们在010Editor中可以看到也就是“64 65 78 0A 30 33 35 00”这8个字节，这些字节都是用16进制表示的，用16进制表示的话，两个数代表一个字节（一个字节等于8位，一个16进制的数能表示4位）。这8个字节用ASCII码表转化一下可以转化为：dex.035（[点击这里可以进行十六进制转ASCII](#)，你可以试试：其中，'!'不是转化来的）。目前，dex的魔数固定为dex.035。

2. 第二个是，**checksum**；它是dex文件的校验和，通过它可以判断dex文件是否被损坏或者被篡改。它占用4个字节，也就是“5D 9D F9 59”。这里提醒一下，在010Editor中，其实可以分别识别我们在DexHeader中看到的这些字段的，你可以点一下这里：

对，就点我

| Name | Value |
|-------------------------------|--|
| struct header item dex_header | |
| > struct dex_magic magic | dex 035 |
| uint checksum | 59F99D5Dh |
| > SHA1 signature[20] | 971CB04F2914713661B0BD7512D80B0B60174D57 |
| uint file_size | 1204 |
| uint header_size | 112 |
| uint endian_tag | 12345678h |
| uint link_size | 0 |
| uint link_off | 0 |
| uint map_off | 1044 |
| uint string_ids_size | 28 |

你可以看到这个header列表展开了，其实我们分析下来就和它这个结构是一样的，你可以先看下，我们现在分析到了checksum中了，你可以看到后面对应的值是“59 F9 9D 5D”。咦？这好像和上面的字节不是一一对应的啊。对的，你可以发现它是反着写的。这是由于dex文件中采用的是小字节序的编码方式，也就是低位上存储的就是低字节内容，所以它们应该要反一下。

3. 第三个到了signature[kSHA1DigestLen]了，signature字段用于检验dex文件，其实就是把整个dex文件用SHA-1签名得到的一个值。这里占用20个字节，你可以自己点010Editor看一看。

4. 第四个fileSize;表示整个文件的大小，占用4个字节。

5. 第五个headerSize;表示DexHeader头结构的大小，占用4个字节。这里可以看到它一共占用了112个字节，112对应的16进制数为70h，你可以选中头文件看看010Editor是不是真的占用了这么多：

| Offset | Hex |
|--------|--|
| 0000h | 64 65 78 0A 30 33 35 00 5D 9D F9 59 97 1C B0 4F |
| 0010h | 29 14 71 36 61 B0 BD 75 12 D8 0B 0B 60 17 4D 57 |
| 0020h | B4 04 00 00 70 00 00 00 78 56 34 12 00 00 00 00 |
| 0030h | 00 00 00 00 00 1C 00 00 00 70 00 00 00 00 00 00 |
| 0040h | 09 00 00 00 00 07 00 00 00 04 01 00 00 00 00 00 |
| 0050h | 00 00 00 58 01 00 00 0A 00 00 00 70 01 00 00 00 |
| 0060h | 00 00 00 C0 01 00 00 D4 02 00 00 E0 01 00 00 00 |
| 0070h | B2 02 00 00 BC 02 00 00 C4 02 00 00 D1 02 00 00 00 |
| 0080h | DC 02 00 00 ED 02 00 00 F0 02 00 00 F5 02 00 00 00 |
| 0090h | F8 02 00 00 06 03 00 00 0A 03 00 00 0E 03 00 00 00 |
| 00A0h | 25 03 00 00 39 03 00 00 4D 03 00 00 68 03 00 00 00 |
| 00B0h | 7C 03 00 00 7F 03 00 00 83 03 00 00 98 03 00 00 00 |
| 00C0h | 9B 03 00 00 A3 03 00 00 A6 03 00 00 B1 03 00 00 00 |
| 00D0h | B7 03 00 00 BC 03 00 00 C5 03 00 00 CF 03 00 00 00 |
| 00E0h | 05 00 00 00 08 00 00 00 0B 00 00 00 0C 00 00 00 00 |
| 00F0h | 0D 00 00 00 0E 00 00 00 0F 00 00 00 10 00 00 00 00 |
| 0100h | 12 00 00 00 06 00 00 00 00 00 00 00 94 02 00 00 00 |
| 0110h | 07 00 00 00 04 00 00 00 00 00 00 00 09 00 00 00 00 |
| 0120h | 04 00 00 00 9C 02 00 00 0A 00 00 00 05 00 00 00 00 |
| 0130h | A4 02 00 00 10 00 00 00 07 00 00 00 00 00 00 00 00 |
| 0140h | 11 00 00 00 07 00 00 00 A4 02 00 00 11 00 00 00 00 |
| 0150h | 07 00 00 00 AC 02 00 00 01 00 00 00 13 00 00 00 00 |

6. 第6个是endianTag;代表字节序标记，用于指定dex运行环境的cpu，预设值为0x12345678，对应010Editor中为“78 56 34 12”（小字节序）。

7. 接下来两个分别是linkSize;和u4 linkOff;这两个字段，它们分别指定了链接段的大小和文件偏移，通常情况下它们都为0。linkSize为0的话表示静态链接。

8. 再下来就是mapOff字段了，它指定了DexMapList的文件偏移，这里我们先不过多介绍它，你可以看一下它的值为“14 04 00 00”，它其实对应的16进制数就是414h（别忘了小字节序），我们可以在414h的位置看一下它在哪里：



其实就是dex文件最后一部分内容。关于这部分内容里面是什么，我们先不说，继续往下看。

9. stringIdsSize 和 stringIdsOff字段：这两个字段指定了dex文件中所有用到的字符串的个数和位置偏移，我们先看stringIdsSize，它的值为：“1C 00 00 00”，16进制的1C也就是十进制的28，也就是说我们这个dex文件中一共有28个字符串，然后stringIdsOff为：“70 00 00 00”，代表字符串的偏移位置为70h，这下我们找到70h的地方：



这下我们就要先介绍一下DexStringId这个结构了，图中从70h开始，所有被选中的都是DexStringId这种数据结构的内容，DexStringId代表的是字符串的位置偏移，每个DexStringId占用4个字节，也就是说它里面存的还不是真正的字符串，它们只是存储了真正字符串的偏移位置。

下面我们先分析几个看看，

①取第一个“B2 02 00 00”，它代表的位置偏移是2B2h，我们先找到这个位置：

http://blog.csdn.net/sinat_18268881

我是2B2h
我在这儿

```
0290h: 00 00 00 00 02 00 00 00 00 00 00 00 01 00 00 00
02A0h: 00 00 00 00 01 00 00 00 04 00 00 00 01 00 00 00
02B0h: 08 00 08 3C 63 6C 69 6E 69 74 3E 00 06 3C 69 6E
02C0h: 69 74 3E 00 0B 48 65 6C 6C 6F 44 61 6C 76 69 6B
02D0h: 00 09 48 65 6C 6C 6F 44 65 78 21 00 0F 48 65 6C
```

可以发现我一共选中了10个字节，这10个字节就表示了一个字符串。下面我们看一下dex文件中的字符串是如何表示的。dex中的字符串采用了一种叫做MUTF-8这样的编码，它是经过传统的UTF-8编码修改的。在MUTF-8中，它的头部存放的是由uleb128编码的字符的个数。（至于uleb128编码是什么编码，这里我不详细展开说，有兴趣的可以搜索看看。）

也就是说在“08 3C 63 6C 69 6E 69 74 3E 00”这些字节中，第一个08指定的是后面需要用到的编码的个数，也就是8个，即“3C 63 6C 69 6E 69 74 3E”这8个，但是我们为什么一共选中了10个字节呢，因为最后一个空字符“0”表示的是字符串的结尾，字符个数没有把它算进去。下面我们来看看“3C 63 6C 69 6E 69 74 3E”这8个字符代表了什么字符串：

| 字符 | 3C | 63 | 6C | 69 | 6E | 69 | 74 | 3E |
|----------|----|----|----|----|----|----|----|----|
| 对应ASCII码 | < | c | l | i | n | i | t | > |

依旧可以点这里查询ASCII。（要说明的一点是，这里凑巧这几个uleb128编码的字符都用了1个字节，所以我们可以这样进行查询，uleb128编码标准用的是1~5个字节，这里只是恰好都是一个字节）。也就是说上面的70h开始的第一个DexStringId指向的其实是字符串“<clint>”（但是貌似我们的代码中没有用到这个字符串啊，先不用管，我们接着分析）。再看到这里：

这里就是70h了

```
0050h: 00 00 00 00 58 01 00 00 0A 00 00 00 70 01 00 00
0060h: 00 00 00 00 C0 01 00 00 D4 02 00 00 E0 01 00 00
0070h: B2 02 00 00 BC 02 00 00 C4 02 00 00 D1 02 00 00
0080h: DC 02 00 00 ED 02 00 00 F0 02 00 00 F5 02 00 00
0090h: F8 02 00 00 06 03 00 00 0A 03 00 00 0E 03 00 00
00A0h: 25 03 00 00 39 03 00 00 4D 03 00 00 68 03 00 00
00B0h: 7C 03 00 00 7F 03 00 00 83 03 00 00 98 03 00 00
00C0h: 9B 03 00 00 A3 03 00 00 A6 03 00 00 B1 03 00 00
00D0h: B7 03 00 00 BC 03 00 00 C5 03 00 00 CF 03 00 00
00E0h: 05 00 00 00 08 00 00 00 0B 00 00 00 0C 00 00 00
00F0h: 0D 00 00 00 0E 00 00 00 0F 00 00 00 10 00 00 00
```

②刚刚我们分析到“B2 02 00 00”所指向的真实字符串了，下面我们接着再分析一个，我们直接分析第三个，不分析第二个了。第三个为“C4 02 00 00”，对应的位置也就是2C4h，我们找到它：

我就是2C4h啦

| | |
|--------|---|
| 02B0h: | 08 00 08 3C 65 6C 69 6E 69 74 3E 00 06 3C 69 6E |
| 02C0h: | 69 74 3E 00 0B 48 65 6C 6C 6F 44 61 6C 76 69 6B |
| 02D0h: | 00 09 48 65 6C 6C 6F 44 65 78 21 00 0F 48 65 6C |
| 02E0h: | 6C 6F 57 6F 72 6C 64 2E 6A 61 76 61 00 01 49 00 |

看这里，这就是2C4h的位置了。我们首先看第一个字符，它的值为0Bh，也就是十进制的11，也就是说接下来的11个字符代表了它的字符串，我们依旧是查看接下来11个字符代表的是什么，经过查询整理：

| | | | | | | | | | | | |
|----------|----|----|----|----|----|----|----|----|----|----|----|
| 字符 | 48 | 65 | 6C | 6C | 6F | 44 | 61 | 6C | 76 | 69 | 6B |
| 对应ASCII码 | H | e | l | l | o | D | a | l | v | i | k |

依旧可以在[这里查询ASCII](#)。上面就是“HelloDalvik”这个字符串，可以看看我们的代码，我们确实用了一个这样的字符串，bingo。

下面剩下的字符串就不分析了。经过整理，可以整理出我们一共用到的28个字符串为：

| | | | | |
|-----|--------------------|--------------------|---------------------------|-----------------------|
| 序号 | 0 | 1 | 2 | 3 |
| 字符串 | <clinit> | <init> | HelloDalvik | HelloDex! |
| 序号 | 4 | 5 | 6 | 7 |
| 字符串 | HelloWorld.java | I | III | L |
| 序号 | 8 | 9 | 10 | 11 |
| 字符串 | LHelloWorld; | LI | LL | Ljava/io/PrintStream; |
| 序号 | 12 | 13 | 14 | 15 |
| 字符串 | Ljava/lang/Object; | Ljava/lang/String; | Ljava/lang/StringBuilder; | Ljava/lang/System; |
| 序号 | 16 | 17 | 18 | 19 |
| 字符串 | V | VL | [Ljava/lang/String; | a |
| 序号 | 20 | 21 | 22 | 23 |
| 字符串 | append | b | getNumber | main |
| 序号 | 24 | 25 | 26 | 27 |
| 字符串 | out | println | toString | valueOf |

ok，字符串这里告一段落，下面我们继续看DexHeader的下面的字段。头好晕~乎乎

噢，读了，还不能结束呢，你现在可以看一下最开始发的那张dex结构图了：



看到了吧，我们这半天分析的stringIdsSize 和 stringIdsOff字段指向的位置就是上面那个箭头指向的位置，它们里面存储的是真实字符串的位置偏移，它们都存储在data区域。（先透露一下，后面我们要分析的几个也和stringIdsSize 与stringIdsOff字段类似，它们里面存储的基本都是位置偏移，并不是真正的数据，真正的数据都在data区域）

好，我们继续。

10. 继续看DexHeader图，我们现在该typeldsSize和typeldsOff了。它们代表什么呢？它们代表的是类的类型的数量和位置偏

| | typeldsSize | typeldsOff |
|--------|-------------|-------------------------|
| 0000h: | 64 65 78 0A | 30 33 35 00 |
| 0010h: | 14 71 36 61 | B0 75 12 D8 0B 0B |
| 0020h: | 04 00 00 00 | 70 00 00 78 56 34 12 |
| 0030h: | 00 00 00 00 | 14 00 00 00 1C 00 00 00 |
| 0040h: | 09 00 00 00 | E0 00 00 00 07 00 00 00 |
| 0050h: | 03 00 00 00 | 58 01 00 00 0A 00 00 00 |
| 0060h: | 01 00 00 00 | C0 01 00 00 D4 02 00 00 |
| 0070h: | B2 02 00 00 | BC 02 00 00 C4 02 00 00 |

移，也是都占4个字节，下面我们看它们的值

可以看到，typeldsSize的值为9h，也就是我们dex文件中用到的类的类型一共有9个，位置偏移在E0h位置，下面我们找到这个位置

| | | | | | |
|--------|-------------|-------------|-------------|-------------|------|
| 00C0h: | 03 00 00 00 | A3 03 00 00 | A6 03 00 00 | B1 03 00 00 | >... |
| 00D0h: | 07 03 00 00 | BC 03 00 00 | C5 03 00 00 | CF 03 00 00 | >... |
| 00E0h: | 05 00 00 00 | 08 00 00 00 | 0B 00 00 00 | 0C 00 00 00 | >... |
| 00F0h: | 0D 00 00 00 | 0E 00 00 00 | 0F 00 00 00 | 10 00 00 00 | >... |
| 0100h: | 12 00 00 00 | 06 00 00 00 | 00 00 00 00 | 94 02 00 00 | >... |
| 0110h: | 07 00 00 00 | 04 00 00 00 | 00 00 00 00 | 09 00 00 00 | >... |
| 0120h: | 04 00 00 00 | 9C 02 00 00 | 0A 00 00 00 | 05 00 00 00 | >... |

看到了吧，我选中的位置就是了。这里我们又得介绍一种数据结构了，因为这里的数据也是一种数据结构的数据组成的。那就是DexTypeId，也就是说选中的内容都是DexTypeId这种数据，这种数据结构中只有一个变量，如下所示：

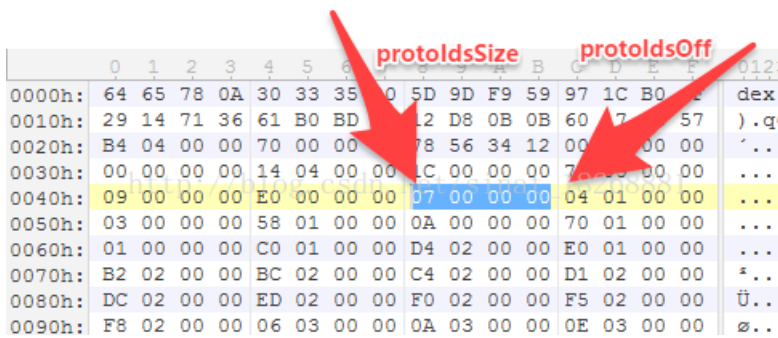
```
struct DexTypeId{
    u4 descriptorIdx; /*指向DexStringId列表的索引*/
}
```

看到了吧，这就是DexTypeId数据结构，它里面只有一个数据descriptorIdx，它的值的内容是DexStringId列表的索引。还记得DexStringId是什么吗？在上面我们分析字符串时，字符串的偏移位置就是由DexStringId这种数据结构描述的，也就是说descriptorIdx指向的是所有的DexStringId组成的列表的索引。上面我们整理出了所有的字符串，你可以翻上去看看图。然后我们看这里一共是9个类的类型代表的都是什么。先看第一个“05 00 00 00”，也就是05h，即十进位的5。然后我们在上面所有整理出的字符串看看5索引的是什么？翻上去可以看到是“i”。接下来我们依次整理这些类的类型，也可以得到类的类型的列表

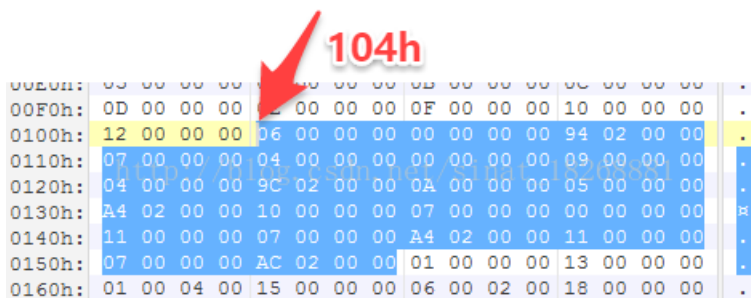
| | | | |
|------|--------------------|-------------------|---------------------------|
| 序号 | 0 | 1 | 2 |
| 类的类型 | I | LHelloWorld; | Ljava/io/PrintStream |
| 序号 | 3 | 4 | 5 |
| 类的类型 | Ljava/lang/Object; | Ljava/lang/String | Ljava/lang/StringBuilder; |
| 序号 | 6 | 7 | 8 |
| 类的类型 | Ljava/lang/System; | V | [Ljava/lang/String |

看到了吧，这就是我们dex文件中所有用到的类的类型。比如“i”代表的就是int，LHelloWorld代表的就是HelloWorld，Ljava/io/PrintStream代表的就是java.io.PrintStream。后面的几个先就不说了。我们接着往下分析。

11. 这下到了protoldsSize和protoldsOff了，它们代表的是dex文件中方法原型的个数和位置偏移。我们先看它们的值



如上图就是它们的值了，protoldsSize的值为十进制的7，说明有7个方法原型，然后位置偏移为104h，我们找到这个位置



看到了吧，这里就是了。对，下面又有新的数据结构了。这下一个数据结构不能满足这块的内容了，我们先看第一个数据结构，DexProtoId

```
struct DexProtoId{
    u4 shortyIdx; /*指向DexStringId列表的索引*/
    u4 returnTypeIdx; /*指向DexTypeId列表的索引*/
    u4 parametersOff; /*指向DexTypeList的位置偏移*/
}
```

可以看到，这个数据结构由三个变量组成。第一个shortyIdx它指向的是我们上面分析的DexStringId列表的索引，代表的是方法声明字符串。第二个returnTypeIdx它指向的是我们上边分析的DexTypeId列表的索引，代表的是方法返回类型字符串。第三个parametersOff指向的是DexTypeList的位置索引，这又是一个新的数据结构了，先说一下这里面存储的是方法的参数列表。可以看到这三个参数，有方法声明字符串，有返回类型，有方法的参数列表，这基本上就确定了我们一个方法的大体内容。

我们接着看看DexTypeList这个数据结构，看看参数列表是如何存储的。

```
struct DexTypeList{
    u4 size; /*DexTypeItem的个数*/
    DexTypeItem list[1]; /*DexTypeItem结构*/
}
```

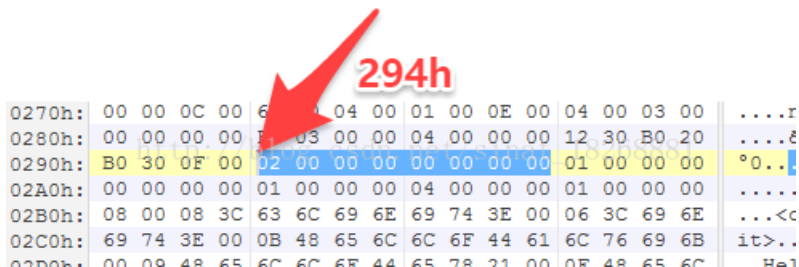
看到了嘛，它有两个参数，其中第一个size说的是DexTypeItem的个数，那DexTypeItem又是啥咧？它又是一种数据结构。我们继续看看

```
struct DexTypeItem{
    u2 typeIdx; /*指向DexTypeId列表的索引*/
}
```

恩，还好，里面就一个参数。也比较简单，就是一个指向DexTypeId列表的索引，也就是代表参数列表中某一个具体的参数的位置。

分析完这几个数据结构了，下面我们具体地分析一个类吧。别走神，我们该从上图的104h开始了。

在104h这里，由于都是DexProtoId这种数据结构的数据，一个DexProtoId一共占用12个字节。所以，我们取前12个字节进行分析。“06 00 00 00, 00 00 00 00, 94 02 00 00”，这就是那12个字节了。首先“06 00 00 00”代表的是shortyIdx，它的值是指向DexStringId列表的索引，我们找到DexStringId列表中第6个对应的值，也就是III，说明这个方法中声明字符串为三个int。接着，“00 00 00 00”代表的是returnTypeIdx，它的值指向的是DexTypeId列表的索引，我们找到对应的值，也就是I，说明这个方法的返回值是int类型的。最后，我们看“94 02 00 00”，它代表的是DexTypeList的位置偏移，它的值为294h，我们找到这个位置



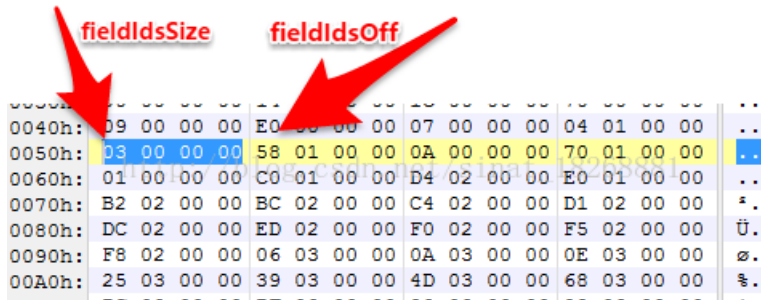
| | | | | | | | | | | | | | | | | |
|--------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|-----|
| 0270h: | 00 | 00 | 0C | 00 | 6 | 04 | 00 | 01 | 00 | 0E | 00 | 04 | 00 | 03 | 00 | ... |
| 0280h: | 00 | 00 | 00 | 00 | 03 | 00 | 00 | 04 | 00 | 00 | 00 | 12 | 30 | B0 | 20 | ... |
| 0290h: | B0 | 30 | 0F | 00 | 02 | 00 | 00 | 00 | 00 | 00 | 00 | 01 | 00 | 00 | 00 | ... |
| 02A0h: | 00 | 00 | 00 | 00 | 01 | 00 | 00 | 00 | 04 | 00 | 00 | 00 | 01 | 00 | 00 | ... |
| 02B0h: | 08 | 00 | 08 | 3C | 63 | 6C | 69 | 6E | 69 | 74 | 3E | 00 | 06 | 3C | 69 | ... |
| 02C0h: | 69 | 74 | 3E | 00 | 0B | 48 | 65 | 6C | 6C | 6F | 44 | 61 | 6C | 76 | 69 | ... |
| 02D0h: | 00 | 08 | 48 | 65 | 6C | 6C | 6F | 44 | 65 | 78 | 21 | 00 | 0F | 48 | 65 | ... |

这里是DexTypeList结构，首先看前4个字节，代表的是DexTypeItem的个数，“02 00 00 00”也就是2，说明接下来有2个DexTypeItem的数据，每个DexTypeItem占用2个字节，也就是两个都是“00 00”，它们的值是DexTypeId列表的索引，我们去找一下，发现0对应的是I，也就是说它的两个参数都是int型的。因此这个方法的声明我们也就确定了。也就是int(int,int)，可以看看我们的源代码，getNumber方法确实是这样的。好，第一个方法就这样分析完了，下面我们依旧是将这些方法的声明整理成列表，后面可能有数据会指向它们的索引。

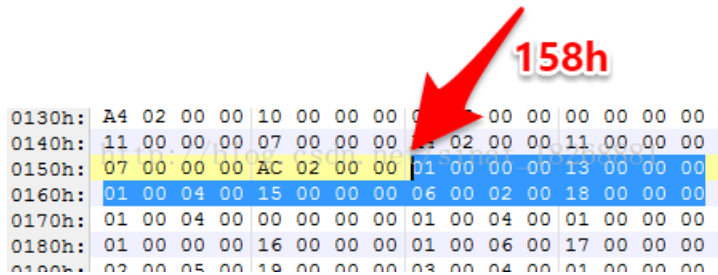
| | | | |
|------|--|---------------------|-------------------------|
| 序号 | 0 | 1 | 2 |
| 方法声明 | int (int, int) | java.lang.String () | java.lang.String (int) |
| 序号 | 3 | 4 | 5 |
| 方法声明 | java.lang.StringBuilder p (java.lang.String) | void () | void (java.lang.String) |
| 序号 | 6 | | |
| 方法声明 | void (java.lang.String[]) | | |

终于又完了一个。我们准备继续下面的。累了就先去听听歌吧，歇一歇再看 -_-

12. **fieldIdsSize**和**fieldIdsOff**字段。这两个字段指向的是dex文件中字段名的信息。我们看到这里



可以看到，fieldIdsSize为3h，说明共有3个字段。fieldIdsOff为158h，说明偏移为158h，我们继续看到158h这里



咳咳，又该新的数据结构了，再忍一忍，接下来的数据结构是DexFieldId，我们看下

```

struct DexFieldId{
    u2 classIdx; /*类的类型，指向DexTypeId列表的索引*/
    u2 typeIdIdx; /*字段类型，指向DexTypeId列表的索引*/
    u4 nameIdx; /*字段名，指向DexStringId列表的索引*/
}

```

可以看到，这三个数据都是指向的索引值，具体的就不说了，看后面的备注就是。我们依旧是分析一下第一个字段，“01 00，00 00，13 00 00 00”，类的类型为DexTypeId列表的索引1，也就是HelloWorld，字段的类型为DexTypeId列表中的索引0，也就是int，字段名为DexStringId列表中的索引13h，即十进制的19，找一下，是a，也就是说我们这个字段就确认了，即int HelloWorld.a。这不就是我们在HelloWorld.java文件里定义的变量a嘛。然后我们依次把我们所有的3个字段都列出来：

Oint HelloWorld.a , ①java.lang.String HelloWorld.b ,②java.io.PrintStream java.lang.System.out

ok，先告一段落。继续分析下一个

13. **methodIdsSize**和**methodIdsOff**字段。这俩字段指明了方法所在的类、方法的声明以及方法名。我们看看

| | | | | | |
|--------|-------------|-------------|-------------|-------------|----------|
| 0030h: | 00 00 00 00 | 14 04 00 00 | 1C 00 00 00 | 70 00 00 00 | |
| 0040h: | 09 00 00 00 | E0 00 00 00 | 07 00 00 00 | 04 00 00 00 |à... |
| 0050h: | 03 00 00 00 | 58 01 00 00 | 0A 00 00 00 | 70 01 00 00 | ...X... |
| 0060h: | 01 00 00 00 | C0 01 00 00 | D4 02 00 00 | E0 01 00 00 | ...À... |
| 0070h: | B2 02 00 00 | BC 02 00 00 | C4 02 00 00 | D1 02 00 00 | ...¼... |
| 0080h: | DC 02 00 00 | ED 02 00 00 | F0 02 00 00 | F5 02 00 00 | Û...í... |
| 0090h: | F8 02 00 00 | 06 03 00 00 | 0A 03 00 00 | 0E 03 00 00 | ø..... |
| 00A0h: | 25 03 00 00 | 39 03 00 00 | 4D 03 00 00 | 68 03 00 00 | §...9... |

先是，methodIdsSize，为Ah，即十进制的10，说明共有10个方法。methodIdsOff，为170h，说明它们的位置偏移在170h。我们看到这里

| | | | | | |
|--------|-------------|-------------|-------------|-------------|-----|
| 0150h: | 01 00 00 00 | AC 02 00 00 | 01 00 00 00 | 13 00 00 00 | ... |
| 0160h: | 01 00 04 00 | 15 00 00 00 | 06 00 02 00 | 18 00 00 00 | ... |
| 0170h: | 01 00 04 00 | 00 00 00 00 | 01 00 04 00 | 01 00 00 00 | ... |
| 0180h: | 01 00 00 00 | 16 00 00 00 | 01 00 06 00 | 17 00 00 00 | ... |
| 0190h: | 02 00 05 00 | 19 00 00 00 | 03 00 04 00 | 01 00 00 00 | ... |
| 01A0h: | 04 00 02 00 | 1B 00 00 00 | 05 00 04 00 | 01 00 00 00 | ... |
| 01B0h: | 05 00 03 00 | 14 00 00 00 | 05 00 01 00 | 1A 00 00 00 | ... |
| 01C0h: | 01 00 00 00 | 01 00 00 00 | 03 00 00 00 | 00 00 00 00 | ... |

对对对，又是新的数据结构，不过这个和上个一样简单，请看DexMethodId

```
struct DexMethodId{
    u2 classIdx; /*类的类型，指向DexTypeId列表的索引*/
    u2 protoIdx; /*声明类型，指向DexProtoId列表的索引*/
    u4 nameIdx; /*方法名，指向DexStringId列表的索引*/
}
```

对吧，这个也简单，三个数据也都是指向对应的结构的索引值。我们直接分析一下第一个数据，“01 00, 04 00, 00 00 00 00”，首先，classIdx，为1，对应DexTypeId列表的索引1，也就是HelloWorld；其次，protoIdx，为4，对应DexProtoId列表中的索引4，也就是void()；最后，nameIdx，为0，对应DexStringId列表中的索引0，也就是<clinit>。因此，第一个数据就出来了，即void HelloWorld.<clinit>()。后面的不进行了，我们依旧是把其余的9个方法列出来

| 索引 | 方法 |
|----|--|
| 0 | void HelloWorld.<clinit>() |
| 1 | void HelloWorld.<init>() |
| 2 | int HelloWorld.getNumber(int, int) |
| 3 | void HelloWorld.main(java.lang.String[]) |
| 4 | void java.io.PrintStream.println(java.lang.String) |
| 5 | void java.lang.Object.<init>() |
| 6 | java.lang.String java.lang.String.valueOf(int) |
| 7 | void java.lang.StringBuilder.<init>() |
| 8 | java.lang.StringBuilder java.lang.StringBuilder.append(java.lang.String) |
| 9 | java.lang.String java.lang.StringBuilder.toString() |

好了，这个就算分析完了。下面真正开始我们的重头戏了。先缓一缓再继续吧。

14. **classDefsSize**和**classDefsOff**字段。这两个字段指明的是dex文件中类的定义的相关信息。我们先找到它们的位置。

```

0020h: 01 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0030h: 00 00 00 00 14 00 00 00 1C 00 00 00 70 00 00 00
0040h: 09 00 00 00 E0 00 00 00 07 00 00 00 04 01 00 00
0050h: 03 00 00 00 58 00 00 00 0A 00 00 00 70 01 00 00
0060h: 01 00 00 00 C0 01 00 00 D4 02 00 00 E0 01 00 00
0070h: B2 02 00 00 BC 02 00 00 C4 02 00 00 D1 02 00 00
0080h: DC 02 00 00 ED 02 00 00 F0 02 00 00 F5 02 00 00
0090h: F8 02 00 00 06 03 00 00 0A 03 00 00 0E 03 00 00
00A0h: 25 03 00 00 39 03 00 00 4D 03 00 00 68 03 00 00
00B0h: 7C 03 00 00 7F 03 00 00 83 03 00 00 98 03 00 00
00C0h: 9B 03 00 00 A3 03 00 00 A6 03 00 00 B1 03 00 00
00D0h: B7 03 00 00 BC 03 00 00 C5 03 00 00 CF 03 00 00
  
```

classDefsSize字段，为1，也就是只有一个类定义，**classDefsOff**，为1C0h，我们找到它的偏移位置。

```

01A0h: 00 00 02 00 1B 00 00 00 05 00 04 00 01 00 00 00
01B0h: 00 00 03 00 14 00 00 00 05 00 01 00 1A 00 00 00
01C0h: 01 00 00 00 01 00 00 00 03 00 00 00 00 00 00 00
01D0h: 04 00 00 00 00 00 00 00 F8 03 00 00 00 00 00 00
01E0h: 01 00 00 00 00 00 00 00 D8 03 00 00 05 00 00 00
01F0h: 1A 00 02 00 69 00 01 00 0E 00 00 00 02 00 01 00
0200h: 01 00 00 00 DD 03 00 00 07 00 00 00 70 10 05 00
0210h: 01 00 12 00 59 10 00 00 0E 00 00 00 05 00 01 00
  
```

这里就是了，到了这里，你现在应该也知道又有新的数据结构了。对的，接下来的数据结构是DexClassDef，请看


```

struct DexClassDef{
    u4 classIdx; /*类的类型, 指向DexTypeId列表的索引*/
    u4 accessFlags; /*访问标志*/
    u4 superclassIdx; /*父类类型, 指向DexTypeId列表的索引*/
    u4 interfacesOff; /*接口, 指向DexTypeList的偏移*/
    u4 sourceFileIdx; /*源文件名, 指向DexStringId列表的索引*/
    u4 annotationsOff; /*注解, 指向DexAnnotationsDirectoryItem结构*/
    u4 classDataOff; /*指向DexClassData结构的偏移*/
    u4 staticValuesOff; /*指向DexEncodedArray结构的偏移*/
}

```

不多说了，我们直接根据结构开始分析吧，反正就只有一个类定义。classIdx为1，对应DexTypeId列表的索引1，找到是HelloWorld，确实是我们源程序中的类的类型。accessFlags为1，它是类的访问标志，对应的值是一个以ACC_开头的枚举值，1对应的是ACC_PUBLIC，你可以在010Editor中看一下，说明我们的类是public的。superclassIdx的值为3，找到DexTypeId列表中的索引3，对应的是java.lang.object，说明我们的类的父类类型是Object的。interfaceOff指向的是DexTypeList结构，我们这里是0说明没有接口。如果有接口的话直接对应到DexTypeList，就和之前我们分析的一样了，这里不多解释，有兴趣的可以写一个有接口的类验证下。再下来sourceFileIdx指向的是DexStringId列表的索引，代表源文件名，我们这里位4，找一下对应到了字符串"HelloWorld.java"，说明我们类程序的源文件名为HelloWorld.java。annotationsOff字段指向注解目录接口，根据类型不同会有注解类、注解方法、注解字段与注解参数，我们这里的值为0，说明没有注解，这里也不过多解释，有兴趣可以自己试试。

接下来是classDataOff了，它指向的是DexClassData结构的位置偏移，DexClassData中存储的是类的数据部分，我们开始详细分析一下它，首先，还是先找到偏移位置3F8h

| | | |
|--------|---|-----|
| 03D0h: | 76 61 6C 75 65 4F 66 00 00 00 07 0E 00 01 00 07 | val |
| 03E0h: | 0E 3C 00 0B 01 00 07 0E 01 1E 5A 87 01 18 0F 00 | < |
| 03F0h: | 06 02 00 00 07 0E 1E 00 01 01 03 01 01 08 00 00 | ... |
| 0400h: | 00 88 80 04 E0 03 01 81 80 04 FC 03 02 09 9C 04 | ... |
| 0410h: | 02 01 FC 04 0D 00 00 00 00 00 00 00 01 00 00 00 | ... |
| 0420h: | 00 00 00 00 01 00 00 00 1C 00 00 00 70 00 00 00 | ... |
| 0430h: | 02 00 00 00 09 00 00 00 E0 00 00 00 03 00 00 00 | ... |
| 0440h: | 07 00 00 00 04 01 00 00 04 00 00 00 03 00 00 00 | ... |
| 0450h: | 58 01 00 00 05 00 00 00 07 00 00 00 70 01 00 00 | v |

接着，我们看看DexClassData数据结构

```

struct DexClassData{
    DexClassDataHeader header; /*指定字段与方法的个数*/
    DexField* staticFields; /*静态字段, DexField结构*/
    DexField* instanceFields; /*实例字段, DexField结构*/
    DexMethod* directMethods; /*直接方法, DexMethod结构*/
    DexMethod* virtualMethods; /*虚方法, DexMethod结构*/
}

```

可以看到，在DexClassData结构中又引入了三种结构，我们一起写出来看一下吧

```

struct DexClassDataHeader{
    u4 staticFieldsSize; /*静态字段个数*/
    u4 instanceFieldsSize; /*实例字段个数*/
    u4 directMethodsSize; /*直接方法个数*/
    u4 virtualMethodsSize; /*虚方法个数*/
}

```

```

struct DexField{
u4 fieldIdx; /指向DexFieldId的索引/
u4 accessFlags; /访问标志/
}

```

```

struct DexMethod{
u4 methodIdx; /指向DexMethodId的索引/
u4 accessFlags; /访问标志/
u4 codeOff; /指向DexCode结构的偏移/
}

```

代码中的注释写的也都很清楚了，我们就不多说了。但是请注意，在这些结构中的u4不是指的占用4个字节，而是指它们是uleb128类型（占用1~5个字节）的数据。关于uleb128还是不多说，想了解的可以自己查查看。

好，接下来开始分析，对于DexClassData，第一个为DexClassDataHeader，我们找到相应的位置，第一个staticFieldsSize其实只占了一个字节，即01h就是它的值，也就是说共有1个静态字段，接下来instanceFieldsSize，directMethodsSize，virtualMethodsSize也都是只占了一个字节，即实例字段的个数为1，直接方法的个数为3，虚方法的个数为1。（这里只是凑巧它们几个都占用一个字节，并不一定是只占用一个字节，这关于到uleb128数据类型，具体可以自己了解下）。

然后接下来就是staticFields了，它对应的数据结构为DexField，可以看到，第一个fieldIdx，是指向DexFieldId的索引，值为1，找到对应的索引值为java.lang.String HelloWorld.b。第二个accessFlags，值为8，对应的ACC_开头的数据为ACC_STATIC（可以在010Editor中对应查看一下），说明我们这个静态字段为：static java.lang.String HelloWorld.b。可以对应我们的源代码看一下，我们确实定义了一个static的b变量。

接着看instanceFields，它和staticFields对应的数据结构是一样的，我们直接分析，第一个fieldIdx，值为0，对应的DexField的索引值为int HelloWorld.a。第二个accessFlags，值为0，对应的ACC_开头的数据为空，就是什么也没有。说明我们这个实例字段为：int HelloWorld.a。可以对应我们的源码看看，我们确实定义了一个a实例变量。

再接着，根据directMethodsSize，有3个直接方法，我们先看第一个，它对应的数据结构是DexMethod，首先methodIdx指向的是DexMethodId的索引，值为0，找到对应的索引值为void HelloWorld.<clinit>()。然后accessFlags为.....为.....为....我的个天！我以为就这样能蒙混过关了，没想到还真碰到一个uleb128数据不是占用一个字节的，这个accessFlags对应的值占用了三个字节，“88 80 04”，为什么？因为是按照uleb128格式的数据读出来的（还是自己去查查吧，这个坑先不填了，其实这种数据也不麻烦，就是前面字节上的最高位指定了是否需要下一个字节上的内容）。“88 80 04”对应的ACC_开头的数据为 ACC_STATIC ACC_CONSTRUCTOR，表明这个方法是静态的，并且是构造方法。最后，看看codeOff，它对应了DexCode结构的偏移，DexCode中存放了方法的指令集等信息，也就是真正的代码了。我们暂且不分析DexCode，就先看看它的偏移位置为“E0 03”，这个等于多少呢？uleb128转化为16进制数结果为：1E0h。也就是DexCode存放在偏移位置1E0h的位置上。

1E0h，这里存放的就是DexCode了

| | | | | | |
|--------|-------------|-------------|-------------|-------------|-------------------|
| 01B0h: | 05 10 00 00 | 14 00 00 00 | 05 00 01 00 | 1A 00 00 00 | |
| 01C0h: | 01 00 00 00 | 01 00 00 00 | 03 00 00 00 | 00 00 00 00 | |
| 01D0h: | 00 00 00 00 | 00 00 00 00 | FB 03 00 00 | 00 00 00 00 |ø..... |
| 01E0h: | 01 00 00 00 | 00 00 00 00 | D8 03 00 00 | 05 00 00 00 |Ø..... |
| 01F0h: | 1A 00 02 00 | 69 00 01 00 | 0E 00 00 00 | 02 00 01 00 |i..... |
| 0200h: | 01 00 00 00 | DD 03 00 00 | 07 00 00 00 | 70 10 05 00 |Ý.....p... |
| 0210h: | 01 00 12 00 | 59 10 00 00 | 0E 00 00 00 | 05 00 01 00 |Y..... |
| 0220h: | 03 00 00 00 | E3 03 00 00 | 28 00 00 00 | 12 10 12 21 |ã...(!.....! |
| 0230h: | 22 02 01 00 | 70 10 01 00 | 02 00 6E 30 | 02 00 02 01 | "...p...n0.... |
| 0240h: | 0A 00 71 10 | 06 00 00 00 | 0C 00 62 01 | 02 00 22 02 | ..q.....b...". |

具体的DexCode我们就先不分析了，因为它里面存放的一些指令局需要根据相关资料一一查找，有兴趣的自己可以找资料看看。剩下的两个直接方法我们也不分析了。

接下来，我们看根据virtualMethodsSize，有1个虚方法，我们直接看。首先methodIdx的值为2，对应的DexMethodId的索引值为int HelloWorld.getNumber(int, int)。然后accessFlags为1，对应的值为ACC_PUBLIC，表明这是一个public类。codeOff为“FC04”，对应的位置为27Ch，这里就不上图了，自己找找吧。

好了，我们整个DEX文件的结构就这样从DexHeader开始基本分析完了，好累啊，不过这样分析一遍，对DEX文件的格式会有更深刻的认识。总是看别人的真不如自己来一遍来的实在！

0x03 ■ 参考资料

参考资料：

《Android软件安全与逆向分析》.非虫

原文：https://blog.csdn.net/sinat_18268881/article/details/55832757