

Android Oat 文件格式分析笔记

ManyFace
2015.12.2

目录

1 Elf Header	4
1.1 e_type 取值	4
1.2 e_machine 取值	4
1.3 文件中 Elf header 分析	5
2 program header table	6
2.1 p_type 取值	6
2.2 p_flags 取值	7
2.3 文件中 program header table 分析	8
3 section header table	9
3.1 sh_type 取值	10
3.2 sh_flag 取值	13
3.3 文件中 section header table 分析	15
3.3.1 第零个 section header	15
3.3.2 第一个 section header	16
3.3.3 第二个 section header	19
3.3.4 第三个 section header	20
3.3.5 第四个 section header	20
3.3.6 第五个 section header	21
3.3.7 第六个 section header	21
3.3.8 第七个 section header	22
4 Oatdata 段	23
4.1 OatHeader	24
4.2 文件中 oatHeader 分析	27
4.3 Dex Meta Data	28
4.4 文件中 Dex Meta Data 分析	29
4.5 OatClass	30
4.5.1 文件中分析 OatClass	33
5 总结	35

参考：

<http://blog.csdn.net/luoshengyang/article/details/39307813>

<http://blog.csdn.net/tenfyguo/article/details/5631561>

https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

http://androidxref.com/5.1.1_r6/

最近分析了 Oat 文件格式，基于 Android5.1.1、32 位，现把分析笔记与大家共享。由于能力有限，笔记中有什么不对的地方，还请各位大神指正。

本次分析基于一个从 Nexus6 中拷贝出来的真实 oat 文件。以该文件为样本，逐步分析，达到深入理解的效果。大家在看本文时，可以结合附件中的图和 oat 文件(demo.oat)。下面进入正题。

Oat 文件其实就是类型为 shared object 的 Elf 文件，因而具有 elf 文件的外壳，图 1 是来自维基的 Elf 文件结构图。在 Oat 文件中，有两个特殊的段：oatdata 段和 oatexec 段，其实就是图 1 中的.rodata 和.text。oatdata 段包含与原始 dex 文件相关的信息，oatexec 段存的是方法的 native code。Program header table 中的信息用于告诉操作系统如何加载该文件，section header table 中的信息主要用于链接阶段。因而，本次分析从 Elf 文件格式开始分析，再深入分析 Oatdata 段。

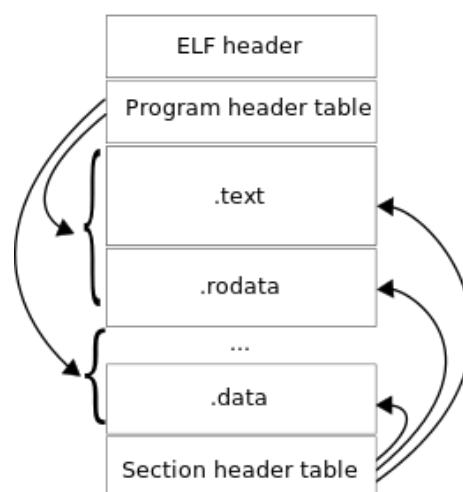


图 1 Elf 文件格式

1 Elf Header

/art/runtime/elf.h Elf32_Ehdr 34h bytes

```
69 struct Elf32_Ehdr {
70     unsigned char e_ident[EI_NIDENT]; // ELF Identification bytes
71     Elf32_Half    e_type;             // Type of file (see ET_* below)
72     Elf32_Half    e_machine;          // Required architecture for this file (see EM_*)
73     Elf32_Word    e_version;          // Must be equal to 1
74     Elf32_Addr    e_entry;            // Address to jump to in order to start program
75     Elf32_Off     e_phoff;            // Program header table's file offset, in bytes
76     Elf32_Off     e_shoff;            // Section header table's file offset, in bytes
77     Elf32_Word    e_flags;            // Processor-specific flags
78     Elf32_Half    e_ehsize;           // Size of ELF header, in bytes
79     Elf32_Half    e_phentsize;        // Size of an entry in the program header table
80     Elf32_Half    e_phnum;            // Number of entries in the program header table
81     Elf32_Half    e_shentsize;        // Size of an entry in the section header table
82     Elf32_Half    e_shnum;            // Number of entries in the section header table
83     Elf32_Half    e_shstrndx;         // Sect hdr table index of sect name string table
84     bool checkMagic() const {
85         return (memcmp(e_ident, ElfMagic, strlen(ElfMagic))) == 0;
86     }
87     unsigned char getFileClass() const { return e_ident[EI_CLASS]; }
88     unsigned char getDataEncoding() const { return e_ident[EI_DATA]; }
89 }
```

1.1 e_type 取值

```
115 // File types
116 enum {
117     ET_NONE = 0,          // No file type
118     ET_REL  = 1,          // Relocatable file
119     ET_EXEC  = 2,          // Executable file
120     ET_DYN   = 3,          // Shared object file
121     ET_CORE  = 4,          // Core file
122     ET_LOPROC = 0xff00,    // Beginning of processor-specific codes
123     ET_HIPROC = 0xffff     // Processor-specific
124 };
```

1.2 e_machine 取值

(0028h 表示是 arm)

```
132 // Machine architectures
133 enum {
134     ... ..
```

```
157  EM_ARM          = 40, // ARM
158  ... ...
297};
```

1.3 文件中 Elf header 分析

字段	值	地址
unsigned char e_ident[16] 该字段不用转字节序	7F454C46 (magicNumber) 01 (32 or 64) 01 (little or big endianness) 01 (Version) 03 (OSabi) 00 (abiVersion) 0000000000000000 (pad)	0
e_type (2b)	0003 (共享目标文件)	10
e_machine (2b)	0028	12
e_version (4b)	00000001	14
e_entry (4b)	00000000	18
e_phoff (4b)	00000034	1C
e_shoff (4b)	0012B070	20
e_flags (4b)	05000000	24
e_ehsize(2b)	0034	28
e_phentsize(2b)	0020	2A
e_phnum(2b)	0005	2C
e_shentsize(2b)	0028	2E
e_shnum(2b)	0008	30
e_shstrndx(2b)	0007	32

由上可知，该 elf 类型是共享目标文件。

- Elf header 的字节数为 e_ehsize 34h bytes。
- Program header table 开始地址是 e_phoff 34h，table 中每一条记录的字节数是 e_phentsize 20h bytes，记录条数为 e_phnum 5h。因而 Program header table 的字节数是 20h*5h=A0h bytes，在文件中的范围[34 h, D3 h]。
- Section header table 开始地址是 e_shoff 12B070h，table 中每一条记录的字节数是 e_shentsize 28h bytes，记录条数为 e_shnum 8h。因而 Section header table 的字节数是 28h*8h=140h bytes，在文件中的范围[12B070 h, 12B1AF h]。
- Section names string table 在 section header table 中的索引是 e_shstrndx 7，也就是说 section header table 中的第 8 条记录是关于 Section names string table 的相关信息。Section names string table 中存着每个 section 的名字。

2 program header table

table 中每条记录的数据结构:

/art/runtime/elf.h Elf32_Phdr 20h bytes

```
1551// Program header for ELF32.
1552struct Elf32_Phdr {
1553    Elf32_Word p_type;    // Type of segment
1554    Elf32_Off  p_offset;  // File offset where segment is located, in bytes
1555    Elf32_Addr p_vaddr;   // Virtual address of beginning of segment
1556    Elf32_Addr p_paddr;   // Physical address of beginning of segment (OS-
                           // specific)
1557    Elf32_Word p_filesz;  // Num. of bytes in file image of segment (may be
                           // zero)
1558    Elf32_Word p_memsz;   // Num. of bytes in mem image of segment (may be
                           // zero)
1559    Elf32_Word p_flags;   // Segment flags
1560    Elf32_Word p_align;   // Segment alignment constraint
1561};
```

- **p_type** 表示段的类型
- **p_offset** 表示该段相对于文件开始的偏移
- **p_vaddr** 表示该段在内存中的首字节地址（即虚拟地址）
- **p_paddr**: 在物理地址定位有关的系统中, 该字段是为该段的物理地址而保留的, 对于可执行文件和共享的 **object** 而言是未指定内容的。
- **p_filesz** 表示在文件映像中该段的字节数(可能是 0)
- **p_memsz** 表示在内存映像中该段的字节数（可能是 0）
- **p_flags**: 该段的相关标志
- **p_align**: 可载入的进程段必须有合适的 **p_vaddr** 、 **p_offset** 值, 取页面大小的模。该字段给出了该段在内存和文件中排列值。0 和 1 表示不需要排列。否则, **p_align** 必须为正的 2 的幂。

2.1 p_type 取值

```
1575// Segment types.
1576enum {
1577    PT_NULL    = 0, // Unused segment.
1578    PT_LOAD    = 1, // Loadable segment.
1579    PT_DYNAMIC = 2, // Dynamic linking information.
1580    PT_INTERP  = 3, // Interpreter pathname.
1581    PT_NOTE    = 4, // Auxiliary information.
1582    PT_SHLIB   = 5, // Reserved.
```

```

1583 PT_PHDR    = 6, // The program header table itself.
1584 PT_TLS     = 7, // The thread-local storage template.
1585 PT_LOOS    = 0x60000000, // Lowest operating system-specific pt entry
type.
1586 PT_HIOS    = 0x6fffffff, // Highest operating system-specific pt entry
type.
1587 PT_LOPROC = 0x70000000, // Lowest processor-specific program hdr entry
type.
1588 PT_HIPROC = 0x7fffffff, // Highest processor-specific program hdr entry
type.
1589
1590 // x86-64 program header types.
1591 // These all contain stack unwind tables.
1592 PT_GNU_EH_FRAME = 0x6474e550,
1593 PT_SUNW_EH_FRAME = 0x6474e550,
1594 PT_SUNW_UNWIND  = 0x6464e550,
1595
1596 PT_GNU_STACK   = 0x6474e551, // Indicates stack executability.
1597 PT_GNU_RELRO   = 0x6474e552, // Read-only after relocation.
1598
1599 // ARM program header types.
1600 PT_ARM_ARCHEXT = 0x70000000, // Platform architecture compatibility info
1601 // These all contain stack unwind tables.
1602 PT_ARM_EXIDX   = 0x70000001,
1603 PT_ARM_UNWIND  = 0x70000001,
1604
1605 // MIPS program header types.
1606 PT_MIPS_REGINFO = 0x70000000, // Register usage information.
1607 PT_MIPS_RTPROC  = 0x70000001, // Runtime procedure table.
1608 PT_MIPS_OPTIONS = 0x70000002 // Options segment.
1609};

```

2.2 p_flags 取值

```

1611// Segment flag bits.
1612enum : unsigned {
1613    PF_X      = 1,          // Execute
1614    PF_W      = 2,          // Write
1615    PF_R      = 4,          // Read
1616    PF_MASKOS = 0x0ff00000, // Bits for operating system-specific semantics.
1617    PF_MASKPROC = 0xf0000000 // Bits for processor-specific semantics.
1618};

```

2.3 文件中 program header table 分析

从上面的分析可以知道，该 table 开始地址是 34h，字节数为 A0h bytes，共有 5 个元素，分别如下：(段的地址范围[p_offset, p_offset+p_filesz-1])

	字段	值	地址	范围
0	p_type (4b)	00000006	34	[34,53]
	p_offset (4b)	00000034	38	
	p_vaddr (4b)	00000034	3C	
	p_paddr (4b)	00000034	40	
	p_filesz (4b)	000000A0	44	
	p_memsz (4b)	000000A0	48	
	p_flags (4b)	00000004 (只读)	4C	
	p_align (4b)	00000004	50	
该元素所指向的段的地址范围			[34, D3]	

从 p_type 可知，第一个元素描述的段是 program header table。

	字段	值	地址	范围
1	p_type (4b)	00000001(Loadable segment)	54	[54,73]
	p_offset (4b)	00000000	58	
	p_vaddr (4b)	00000000	5C	
	p_paddr (4b)	00000000	50	
	p_filesz (4b)	000B5000	64	
	p_memsz (4b)	000B5000	68	
	p_flags (4b)	00000004 (只读)	6C	
	p_align (4b)	00001000	70	
该元素所指向的段的地址范围			[0, B4FFF]	

	字段	值	地址	范围
2	p_type (4b)	00000001(Loadable segment)	74	[74,93]
	p_offset (4b)	000B5000	78	
	p_vaddr (4b)	000B5000	7C	
	p_paddr (4b)	000B5000	80	
	p_filesz (4b)	00075FE8	84	

	p_memsz (4b)	00075FE8	88	
	p_flags (4b)	00000005(可读可执行)	8C	
	p_align (4b)	00001000	90	
该元素所指向的段的地址范围			[B5000, 12AFE7]	

	字段	值	地址	范围
3	p_type (4b)	00000001(Loadable segment)	94	[94,B3]
	p_offset (4b)	0012B000	98	
	p_vaddr (4b)	0012B000	9C	
	p_paddr (4b)	0012B000	A0	
	p_filesz (4b)	00000038	A4	
	p_memsz (4b)	00000038	A8	
	p_flags (4b)	00000006(可读可写)	AC	
	p_align (4b)	00001000	B0	
该元素所指向的段的地址范围			[12B000, 12B037]	

	字段	值	地址	范围
4	p_type (4b)	00000002(Dynamic segment)	B4	[B4,D3]
	p_offset (4b)	0012B000	B8	
	p_vaddr (4b)	0012B000	BC	
	p_paddr (4b)	0012B000	C0	
	p_filesz (4b)	00000038	C4	
	p_memsz (4b)	00000038	C8	
	p_flags (4b)	00000006(可读可写)	CC	
	p_align (4b)	00001000	D0	
该元素所指向的段的地址范围			[12B000, 12B037]	

3 section header table

table 中每条记录的数据结构:

/art/runtime/elf.h Elf32_Shdr 28h bytes

1203	// Section header.		
1204	struct <i>Elf32_Shdr</i> {		
1205	Elf32_Word	sh_name;	// Section name (index into string table)

```

1206 Elf32_Word sh_type;      // Section type (SHT_*)
1207 Elf32_Word sh_flags;     // Section flags (SHF_*)
1208 Elf32_Addr sh_addr;      // Address where section is to be loaded
1209 Elf32_Off  sh_offset;     // File offset of section data, in bytes
1210 Elf32_Word sh_size;      // Size of section, in bytes
1211 Elf32_Word sh_link;      // Section type-specific header table index link
1212 Elf32_Word sh_info;      // Section type-specific extra information
1213 Elf32_Word sh_addralign;  // Section address alignment
1214 Elf32_Word sh_entsize;   // Size of records contained within the section
1215};

```

- **sh_name**: 该字段表示 Section 的名字，是 string table 的索引，通过 string table 找到具体的字符串。
- **sh_type**: Section 的类型。
- **sh_flags**: section 支持位的标记，用来描述多个属性。
- **sh_addr**: 假如该 section 将出现在进程的内存映像空间里，该字段表示该 section 在内存中的位置。否则，该字段为 0。
- **sh_offset**: 该 section 在文件中的偏移。SHT_NOBITS 类型的 section 在文件中不占空间，它的 sh_offset 定位在文件中的概念上的位置。
- **sh_size**: 表示 section 字节大小。类型为 SHT_NOBITS 的 section 的 sh_size 可能为非 0 大小，但是不占文件空间。
- **sh_link**: 该成员保存了一个 section 报头表的索引连接，它的解释依靠该 section 的类型。
- **sh_info**: 额外信息，它的解释依靠该 section 的类型。
- **sh_addralign**: 一些 sections 有地址对齐的约束。例如，假如一个 section 保存着双字，系统就必须确定整个 section 是否双字对齐。所以 sh_addr 的值以 sh_addralign 的值作模，那么一定为 0。当然的，仅仅 0 和正的 2 的次方是允许的。值 0 和 1 意味着该 section 没有对齐要求。
- **sh_entsize**: 一些 sections 保存着一张固定大小入口的表，就像符号表。对于这样一个 section 来说，该字段给出了每个入口的字节大小。如果该 section 没有保存着一张固定大小入口的表，该成员就为 0。

3.1 sh_type 取值

```

1245// Section types.
1246enum : unsigned {

```

```

1247 SHT_NULL          = 0, // No associated section (inactive entry).
1248 SHT_PROGBITS       = 1, // Program-defined contents.
1249 SHT_SYMTAB         = 2, // Symbol table.
1250 SHT_STRTAB         = 3, // String table.
1251 SHT_RELA           = 4, // Relocation entries; explicit addends.
1252 SHT_HASH           = 5, // Symbol hash table.
1253 SHT_DYNAMIC        = 6, // Information for dynamic linking.
1254 SHT_NOTE           = 7, // Information about the file.
1255 SHT_NOBITS         = 8, // Data occupies no space in the file.
1256 SHT_REL           = 9, // Relocation entries; no explicit addends.
1257 SHT_SHLIB          = 10, // Reserved.
1258 SHT_DYNSYM         = 11, // Symbol table.
1259 SHT_INIT_ARRAY     = 14, // Pointers to initialization functions.
1260 SHT_FINI_ARRAY     = 15, // Pointers to termination functions.
1261 SHT_PREINIT_ARRAY  = 16, // Pointers to pre-init functions.
1262 SHT_GROUP          = 17, // Section group.
1263 SHT_SYMTAB_SHNDX   = 18, // Indices for SHN_XINDEX entries.
1264 SHT_LOOS          = 0x60000000, // Lowest operating system-specific type.
1265 SHT_GNU_ATTRIBUTES = 0x6ffffff5, // Object attributes.
1266 SHT_GNU_HASH       = 0x6ffffff6, // GNU-style hash table.
1267 SHT_GNU_verdef     = 0x6ffffffd, // GNU version definitions.
1268 SHT_GNU_verneed    = 0x6ffffffe, // GNU version references.
1269 SHT_GNU_versym     = 0x6fffffff, // GNU symbol versions table.
1270 SHT_HIOS          = 0x7fffffff, // Highest operating system-specific type.
1271 SHT_LOPROC        = 0x70000000, // Lowest processor arch-specific type.
1272 // Fixme: All this is duplicated in MCSectionELF. Why??
1273 // Exception Index table
1274 SHT_ARM_EXIDX      = 0x70000001U,
1275 // BPABI DLL dynamic linking pre-emption map
1276 SHT_ARM_PREEMPTMAP = 0x70000002U,
1277 // Object file compatibility attributes
1278 SHT_ARM_ATTRIBUTES = 0x70000003U,
1279 SHT_ARM_DEBUGOVERLAY = 0x70000004U,
1280 SHT_ARM_OVERLAYSECTION = 0x70000005U,
1281 SHT_HEX_ORDERED    = 0x70000000, // Link editor is to sort the
entries in
1282                                     // this section based on their sizes
1283 SHT_X86_64_UNWIND   = 0x70000001, // Unwind information
1284
1285 SHT_MIPS_REGINFO    = 0x70000006, // Register usage information
1286 SHT_MIPS_OPTIONS    = 0x7000000d, // General options
1287
1288 SHT_HIPROC         = 0x7fffffff, // Highest processor arch-specific type.
1289 SHT_LOUSER         = 0x80000000, // Lowest type reserved for applications.

```

```
1290 SHT_HIUSER          = 0xffffffff // Highest type reserved for applications.  
1291};
```

- **SHT_NULL**: 该值表示该 section 头是无效的, 它没有相关的 section, 该 section 的其他成员的值都是未定义的。
- **SHT_PROGBITS**: 该 section 保存被程序定义了的一些信息, 它的格式和意义取决于程序本身。
- **SHT_SYMTAB** and **SHT_DYNSYM**: 该 section 保存着一个符号表(symbol table)。
- **SHT_STRTAB**: 该 section 保存着一个字符串表。一个 object 文件可以包含多个字符串表的 section。
- **SHT_RELA**: 该 section 保存着具有明确加数的重定位入口。就象 object 文件 32 位的 Elf32_Rela 类型。一个 object 文件可能有多个重定位的 sections。
- **SHT_HASH**: 该标号保存着一个标号的哈希(hash)表。所有的参与动态连接的 object 一定包含了一个标号哈希表 (hash table)。当前的, 一个 object 文件可能只有一个哈希表。
- **SHT_DYNAMIC**: 该 section 保存着动态连接的信息。当前的, 一个 object 文件可能只有一个动态的 section。
- **SHT_NOTE**: 该 section 保存着其他的一些标志文件的信息。
- **SHT_NOBITS**: 该类型的 section 在文件中不占空间, 但是类似 SHT_PROGBITS。尽管该 section 不包含字节, sh_offset 成员包含了概念上的文件偏移量。
- **SHT_REL** 该 section 保存着具有明确加数的重定位的入口。就象 object 文件 32 位类型 Elf32_Rel 类型。一个 object 文件可能有多个重定位的 sections。
- **SHT_SHLIB**: 该 section 类型保留但语意没有指明。包含这个类型的 section 的程序是不符合 ABI 的。
- **SHT_LOPROC** through **SHT_HIPROC**: 在这范围之间的值为特定处理器语意保留的。
- **SHT_LOUSER**: 该变量为应用程序保留的索引范围的最小边界。
- **SHT_HIUSER**: 该变量为应用程序保留的索引范围的最大边界。在 SHT_LOUSER 和 HIUSER 的 section 类型可能被应用程序使用, 这 and 当前或者将来系统定义的 section 类型是不矛盾的。

3.2 sh_flag 取值

```
1293 // Section flags.
1294 enum : unsigned {
1295     // Section data should be writable during execution.
1296     SHF_WRITE = 0x1,
1297
1298     // Section occupies memory during program execution.
1299     SHF_ALLOC = 0x2,
1300
1301     // Section contains executable machine instructions.
1302     SHF_EXECINSTR = 0x4,
1303
1304     // The data in this section may be merged.
1305     SHF_MERGE = 0x10,
1306
1307     // The data in this section is null-terminated strings.
1308     SHF_STRINGS = 0x20,
1309
1310     // A field in this section holds a section header table index.
1311     SHF_INFO_LINK = 0x40U,
1312
1313     // Adds special ordering requirements for link editors.
1314     SHF_LINK_ORDER = 0x80U,
1315
1316     // This section requires special OS-specific processing to avoid incorrect
1317     // behavior.
1318     SHF_OS_NONCONFORMING = 0x100U,
1319
1320     // This section is a member of a section group.
1321     SHF_GROUP = 0x200U,
1322
1323     // This section holds Thread-Local Storage.
1324     SHF_TLS = 0x400U,
1325
1326     // This section is excluded from the final executable or shared library.
1327     SHF_EXCLUDE = 0x80000000U,
1328
1329     // Start of target-specific flags.
1330
1331     /// XCORE_SHF_CP_SECTION - All sections with the "c" flag are grouped
1332     /// together by the linker to form the constant pool and the cp register
1333     is
1334     /// set to the start of the constant pool by the boot code.
```

```
1334 XCORE_SHF_CP_SECTION = 0x800U,
1335
1336 /// XCORE_SHF_DP_SECTION - All sections with the "d" flag are grouped
1337 /// together by the linker to form the data section and the dp register is
1338 /// set to the start of the section by the boot code.
1339 XCORE_SHF_DP_SECTION = 0x1000U,
1340
1341 SHF_MASKOS    = 0x0ff00000,
1342
1343 // Bits indicating processor-specific flags.
1344 SHF_MASKPROC = 0xf0000000,
1345
1346 // If an object file section does not have this flag set, then it may not
hold
1347 // more than 2GB and can be freely referred to in objects using smaller
code
1348 // models. Otherwise, only objects using larger code models can refer to
them.
1349 // For example, a medium code model object can refer to data in a section
that
1350 // sets this flag besides being able to refer to data in a section that
does
1351 // not set it; likewise, a small code model object can refer only to code
in a
1352 // section that does not set this flag.
1353 SHF_X86_64_LARGE = 0x10000000,
1354
1355 // All sections with the GPREL flag are grouped into a global data area
1356 // for faster accesses
1357 SHF_HEX_GPREL = 0x10000000,
1358
1359 // Section contains text/data which may be replicated in other sections.
1360 // Linker must retain only one copy.
1361 SHF_MIPS_NODUPES = 0x01000000,
1362
1363 // Linker must generate implicit hidden weak names.
1364 SHF_MIPS_NAMES    = 0x02000000,
1365
1366 // Section data local to process.
1367 SHF_MIPS_LOCAL    = 0x04000000,
1368
1369 // Do not strip this section.
1370 SHF_MIPS_NOSTRIP = 0x08000000,
1371
```

```

1372 // Section must be part of global data area.
1373 SHF_MIPS_GPREL    = 0x10000000,
1374
1375 // This section should be merged.
1376 SHF_MIPS_MERGE    = 0x20000000,
1377
1378 // Address size to be inferred from section entry size.
1379 SHF_MIPS_ADDR     = 0x40000000,
1380
1381 // Section data is string data by default.
1382 SHF_MIPS_STRING   = 0x80000000
1383};

```

3.3 文件中 section header table 分析

从 Elf Header 的分析可以知道，该 table 的开始地址是 12B070h，table 中每一条记录的字节数是 28h bytes，section header 数目为 8h，Section header table 的字节数是 28h*8h=140h bytes，在文件中的范围[12B070 h, 12B1AF h]。Section names string table 在 section header table 中的索引是 7。8 个 section header 分别如下。(section 的地址范围[sh_offset, sh_offset+sh_size-1])

3.3.1 第零个 section header

	字段	值	地址	范围
0	sh_name (4b)	00000000 (NULL)	12B070	[12B070, 12B097]
	sh_type(4b)	00000000->SHT_NULL->无效	12B074	
	sh_flags (4b)	00000000	12B078	
	sh_addr (4b)	00000000	12B07C	
	sh_offset(4b)	00000000	12B080	
	sh_size (4b)	00000000	12B084	
	sh_link (4b)	00000000	12B088	
	sh_info (4b)	00000000	12B08C	
	sh_addralign(4b)	00000000	12B090	
	sh_entsize(4b)	00000000	12B094	
该元素所指向的 section 的地址范围				

由于该 section header 的 sh_type 是 0(SHT_NULL)，因而无效，没有定义。

3.3.2 第一个 section header

		字段	值	地址	范围
1		sh_name (4b)	00000001 (.dynsym)	12B098	[12B098, 12B0BF]
		sh_type(4b)	0000000B (SHT_DYNSYM : symbol table)	12B09C	
		sh_flags (4b)	00000002 (SHF_ALLOC)	12B0A0	
		sh_addr (4b)	000000D4	12B0A4	
		sh_offset(4b)	000000D4	12B0A8	
		sh_size (4b)	00000040	12B0AC	
		sh_link (4b)	00000002	12B0B0	
		sh_info (4b)	00000000	12B0B4	
		sh_addralign(4b)	00000004	12B0B8	
		sh_entsize(4b)	00000010	12B0BC	
		该元素所指向的 section 的地址范围		[D4, 113]	

3.3.2.1 symbol table

symbol table 符号表：一个 object 文件的符号表保存了一个程序在定位和重定位时需要的定义和引用的信息。一个符号表索引是相应的下标。0 表项特指了该表的第一个入口。

symbol table 的 entry 结构如下：

```
/art/runtime/elf.h Elf32_Sym 10h bytes
1392// Symbol table entries for ELF32.
1393struct Elf32_Sym {
1394    Elf32_Word    st_name; // Symbol name (index into string table)
1395    Elf32_Addr    st_value; // Value or address associated with the symbol
1396    Elf32_Word    st_size; // Size of the symbol
1397    unsigned char st_info; // Symbol's type and binding attributes
1398    unsigned char st_other; // Must be zero; reserved
1399    Elf32_Half    st_shndx; // Which section (header table index) it's defined
in
1401 // These accessors and mutators correspond to the ELF32_ST_BIND,
1402 // ELF32_ST_TYPE, and ELF32_ST_INFO macros defined in the ELF
specification:
1403 unsigned char getBinding() const { return st_info >> 4; }
1404 unsigned char getType() const { return st_info & 0x0f; }
1405 void setBinding(unsigned char b) { setBindingAndType(b, getType()); }
1406 void setType(unsigned char t) { setBindingAndType(getBinding(), t); }
```



```

1407 void setBindingAndType(unsigned char b, unsigned char t) {
1408     st_info = (b << 4) + (t & 0xf);
1409 }
1410};

```

- **st_name:** 该字段保存了该 object 文件的符号在 string table 中的索引。如果该值不为 0，则它代表了给出符号名的 string table 索引。否则，该符号无名。
- **st_value:** 该成员给出了相应的符号值。它可能是绝对值或地址等等（依赖于上下文）；
- **st_size:** 符号的大小。比如，一个数据对象的大小是该对象所包含的字节数目。如果该符号的大小未知或没有大小则这个成员为 0。
- **st_info:** 成员指出了符号的类型和相应的属性。
- **st_other:** 该成员目前为 0，没有含义。
- **st_shndx:** 每一个符号表的入口都定义为和某些 section 相关；该字段保存了相关 section header 的索引。由此可得到该 symbol 所指向的段的偏移 = section_header_table[Elf32_Sym.st_shndx].sh_offset + Elf32_Sym.st_value - section_header_table[Elf32_Sym.st_shndx].sh_addr

3.3.2.2 文件中 symbol table 分析

该 section header 所指向的 section 为 symbol table，section 的开始地址为 **sh_offset** D4h，字节数为 **sh_size** 40h bytes，地址范围为[D4 h, 113 h]，由于 symbol table 的一条 entry 的字节数为 10h bytes(可以从 section header 的 sh_entsize 得出，也可以直接从 Elf32_Sym 结构体的字节数得出)，所以该 table 有 4 条 entry，分别如下(string table 是第二个 section header 所描述的 section，以 st_name 为索引在 string table 中查找对应的 symbol 名字)：

索引	字段	值	地址	范围
0	st_name (4b)	00 00 00 00(0->无效的符号)	D4	[D4, E3]
	st_value(4b)	00 00 00 00	D8	
	st_size(4b)	00 00 00 00	DC	
	st_info (1b)	00	E0	
	st_other(1b)	00	E1	

	st_shndx(2b)	00 00	E2	
--	---------------------	--------------	----	--

索引	字段	值	地址	范围
1	st_name (4b)	00 00 00 01(oatdata)	E4	[E4, F3]
	st_value(4b)	00 00 10 00	E8	
	st_size(4b)	00 0B 40 00	EC	
	st_info (1b)	11	F0	
	st_other(1b)	00	F1	
	st_shndx(2b)	00 04	F2	

oatdata 段的地址范围：[1000h, B4FFFh]，该符号与第 4 个 section header 相关，即名为.rodata 的 section（且看后续分析）

索引	字段	值	地址	范围
2	st_name (4b)	00 00 00 09(oatexec)	F4	[F4, 103]
	st_value(4b)	00 0B 50 00	F8	
	st_size(4b)	00 07 5F E8	FC	
	st_info (1b)	11	100	
	st_other(1b)	00	101	
	st_shndx(2b)	00 05	102	

oatexec 段的地址范围：[B5000h, 12AFE7h]，该符号与第 5 个 section header 相关，即名为.text 的 section（且看后续分析）

索引	字段	值	地址	范围
3	st_name (4b)	00 00 00 11(oatlastword)	104	[104, 113]
	st_value(4b)	00 12 AF E4	108	
	st_size(4b)	00 00 00 04	10C	
	st_info (1b)	11	110	
	st_other(1b)	00	111	
	st_shndx(2b)	00 05	112	

oatlastword 符号指向的是 oatexec 段的最后一个字，因而 oatexec 段最后一

个字节的地址=12AFE4+4-1=12AFE7，和上面得到的是一个结果。

3.3.3 第二个 section header

	字段	值	地址	范围
2	sh_name (4b)	00000009(.dynstr)	12B0C0	[12B0C0, 12B0E7]
	sh_type(4b)	00000003 (SHT_STRTAB: string table)	12B0C4	
	sh_flags (4b)	00000002(SHF_ALLOC)	12B0C8	
	sh_addr (4b)	00000114	12B0CC	
	sh_offset(4b)	00000114	12B0D0	
	sh_size (4b)	0000004E	12B0D4	
	sh_link (4b)	00000000	12B0D8	
	sh_info (4b)	00000000	12B0DC	
	sh_addralign(4b)	00000001	12B0E0	
	sh_entsize(4b)	00000001	12B0E4	
该元素所指向的 section 的地址范围			[114, 161]	

3.3.3.1 string table

string table section 保存着以 NULL 为终止符的字符串，object 文件使用这些字符串来描绘符号和 section 名。其第一个字节，即索引 0，被定义保存着一个 NULL 字符。索引 0 的字符串是没有名字或者说是 NULL，它的解释依靠上下文。一个空的 string table section 是允许的，它的 section header 的字段 sh_size 将为 0，对空的 string table 来说，非 0 的索引是没有用的。注意索引的值是字符串的首字节在 string table 中的偏移。

该 section header 指向的 string table section 为：

0100h:	11 00 05 00	11 00 00 00	E4 AF 12 00	04 00 00 00a	
0110h:	11 00 05 00	00 6F 61 74	64 61 74 61	00 6F 61 74oatdata.oat	
0120h:	65 78 65 63	00 6F 61 74	6C 61 73 74	77 6F 72 64	exec.oatlastword	
0130h:	00 64 61 74	61 40 61 70	70 40 63 6F	6D 2E 63 70	.data@app@com.cp	
0140h:	66 2E 78 64	65 66 64 65	6D 6F 2D 31	40 62 61 73	f.xdefdemo-1@bas	
0150h:	65 2E 61 70	6B 40 63 6C	61 73 73 65	73 2E 64 65	e.apk@classes.de	
0160h:	78 00	00 00	02 00 00 00	04 00 00 00	03 00 00 00	x.....
0170h:	01 00	00 00	00 00 00 00	02 00 00 00	00 00 00 00

整理成表格：

索引	值
----	---

0	\00 即 NULL
1	"oatdata\00"
9	"oatexec\00"
11	"oatlastword\00"
1D	"data@app@com.cpf.xdefdemo-1@base.apk@classes.dex\00"

3.3.4 第三个 section header

	字段	值	地址	范围
3	sh_name (4b)	00000011(.hash)	12B0E8	[12B0E8, 12B10F]
	sh_type(4b)	00000005 (SHT_HASH)	12B0EC	
	sh_flags (4b)	00000002(SHF_ALLOC)	12B0F0	
	sh_addr (4b)	00000164	12B0F4	
	sh_offset(4b)	00000164	12B0F8	
	sh_size (4b)	00000020	12B0FC	
	sh_link (4b)	00000001	12B100	
	sh_info (4b)	00000000	12B104	
	sh_addralign(4b)	00000004	12B108	
	sh_entsize(4b)	00000004	12B10C	
该元素所指向的 section 的地址范围			[164, 183]	

sh_name=.hash 表示该 section header 描述的 section 是一个散列表，允许在不对全表元素进行先行搜索的情况下，快速访问所有符号表项。在 elf.h 中没有找到对 hash 表的定义。。。。。

3.3.5 第四个 section header

	字段	值	地址	范围
4	sh_name (4b)	00000017(.rodata)	12B110	[12B110, 12B137]
	sh_type(4b)	00000001 (SHT_PROGBITS)	12B114	
	sh_flags (4b)	00000002(SHF_ALLOC)	12B118	
	sh_addr (4b)	00001000	12B11C	
	sh_offset(4b)	00001000	12B120	
	sh_size (4b)	000B4000	12B124	

	sh_link (4b)	00000000	12B128	
	sh_info (4b)	00000000	12B12C	
	sh_addralign(4b)	00001000	12B130	
	sh_entsize(4b)	00000000	12B134	
该元素所指向的 section 的地址范围			[1000, B4FFF]	

sh_name=.rodata 表示该 section header 描述的 section 保存着只读数据，可以读取，但不能修改。比如，初始化了的全局静态变量和局部静态变量，printf 语句中所有的静态字符串也保存在该 section。在 Android oat 文件中，该 section 保存着 oat 的相关信息、dex 文件、dex 文件类中与 native code 的映射关系。

3.3.6 第五个 section header

	字段	值	地址	范围
5	sh_name (4b)	0000001F(.text)	12B138	[12B138, 12B15F]
	sh_type(4b)	00000001 (SHT_PROGBITS)	12B13C	
	sh_flags (4b)	00000006 (SHF_ALLOC+ SHF_EXECINSTR)	12B140	
	sh_addr (4b)	000B5000	12B144	
	sh_offset(4b)	000B5000	12B148	
	sh_size (4b)	00075FE8	12B14C	
	sh_link (4b)	00000000	12B150	
	sh_info (4b)	00000000	12B154	
	sh_addralign(4b)	00001000	12B158	
	sh_entsize(4b)	00000000	12B15C	
该元素所指向的 section 的地址范围			[B5000,12AFE7]	

sh_name=.text 表示该 section header 描述的 section 保存着已编译程序的机器码，即程序的指令序列（dex 文件中类方法对应的 native code）

3.3.7 第六个 section header

	字段	值	地址	范围
6	sh_name (4b)	00000025(.dynamic)	12B160	[12B160, 12B187]
	sh_type(4b)	00000006(SHT_DYNAMIC)	12B164	
	sh_flags (4b)	00000002(SHF_ALLOC)	12B168	
	sh_addr (4b)	0012B000	12B16C	

	sh_offset(4b)	0012B000	12B170	
	sh_size (4b)	00000038	12B174	
	sh_link (4b)	00000001	12B178	
	sh_info (4b)	00000000	12B17C	
	sh_addralign(4b)	00001000	12B180	
	sh_entsize(4b)	00000008	12B184	
该元素所指向的 section 的地址范围			[12B000,12B037]	

sh_name=.dynamic 表示该 section header 描述的 section 保存着动态链接信息。

3.3.7.1 dynamic table

dynamic table 中是关于动态链接信息，每一条 entry 的大小是 sh_entsize 8 bytes，其结构如下：

/art/runtime/elf.h Elf32_Dyn 8h bytes

```
1620// Dynamic table entry for ELF32.
1621struct Elf32_Dyn
1622{
1623    Elf32_Sword d_tag;                // Type of dynamic table entry.
1624    union
1625    {
1626        Elf32_Word d_val;            // Integer value of entry.
1627        Elf32_Addr d_ptr;            // Pointer value of entry.
1628    } d_un;
1629};
```

具体在文件中就不分析了，我们的关注焦点是 oatdata 段(.rodata)和 oatexec 段(.text)。

3.3.8 第七个 section header

	字段	值	地址	范围
7	sh_name (4b)	0000002E(.shstrtab)	12B188	[12B188, 12B1AF]
	sh_type(4b)	00000003 (SHT_STRTAB:string table)	12B18C	
	sh_flags (4b)	00000000	12B190	
	sh_addr (4b)	00000000	12B194	

	sh_offset(4b)	0012B038	12B198	
	sh_size (4b)	00000038	12B19C	
	sh_link (4b)	00000000	12B1A0	
	sh_info (4b)	00000000	12B1A4	
	sh_addralign(4b)	00000001	12B1A8	
	sh_entsize(4b)	00000001	12B1AC	
该元素所指向的 section 的地址范围			[12B038, 12B06F]	

sh_name=.shstrtab 表示该 section header 指向的 section 包含了 section 名字的字符串表。也可通过前面分析 Elf header，可知，第 8 个记录是关于 Section names string table 的相关信息，sh_type 表明该 section header 指向的 section 是一个 string table，该 section header 指向的 section 中的字符串是 section header 的名字，各个 section header 通过索引获取相应的字符串，注意索引的值是字符串的首字节在 string table 中的偏移。

该 section header 指向的 string table section 为：

12:B020h:	0A 00 00 00	4E 00 00 00	0E 00 00 00	1D 00 00 00N.....
12:B030h:	00 00 00 00	00 00 00 00	00 2E 64 79	6E 73 79 6Ddynsym
12:B040h:	00 2E 64 79	6E 73 74 72	00 2E 68 61	73 68 00 2E	..dynstr..hash..
12:B050h:	72 6F 64 61	74 61 00 2E	74 65 78 74	00 2E 64 79	rodata..text..dy
12:B060h:	6E 61 6D 69	63 00 2E 73	68 73 74 72	74 61 62 00	namic..shstrtab.
12:B070h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00

整理成表格：

索引	值
0	\00 即 NULL
1	".dynsym\00" (8 个字节)
9	".dynstr\00" (8 个字节)
11	".hash\00"
17	".rodata\00"
1F	".text\00"
25	".dynamic\00"
2E	".shstrtab\00"

4 Oatdata 段

oatdata 段包含了一个 OatHeader、dex 文件的相关信息、dex 原始文件、类

中方法与翻译为 native code 的对应关系。因而，oatdata 段的解析是 oat 文件加载过程中很重要的环节，通过 oatdata 段(.rodata section)中的信息，可以找到方法对应的 native code。native code 位于 oatexec 段(.text section)。

由 [symbol table](#) 可以知道，Oatdata 段地址范围是 [1000h, B4FFFh]。

4.1 OatHeader

art/runtime/oat.h

```
30class PACKED(4) OatHeader {
109 private:
110 ... ..
119 uint8_t magic_[4];
120 uint8_t version_[4];
121 uint32_t adler32_checksum_;
122
123 InstructionSet instruction_set_;
124 InstructionSetFeatures instruction_set_features_;
125 uint32_t dex_file_count_;
126 uint32_t executable_offset_;
127 uint32_t interpreter_to_interpreter_bridge_offset_;
128 uint32_t interpreter_to_compiled_code_bridge_offset_;
129 uint32_t jni_dlsym_lookup_offset_;
130 uint32_t portable_imt_conflict_trampoline_offset_;
131 uint32_t portable_resolution_trampoline_offset_;
132 uint32_t portable_to_interpreter_bridge_offset_;
133 uint32_t quick_generic_jni_trampoline_offset_;
134 uint32_t quick_imt_conflict_trampoline_offset_;
135 uint32_t quick_resolution_trampoline_offset_;
136 uint32_t quick_to_interpreter_bridge_offset_;
137
138 // The amount that the image this oat is associated with has been patched.
139 int32_t image_patch_delta_;
140
141 uint32_t image_file_location_oat_checksum_;
142 uint32_t image_file_location_oat_data_begin_;
143
144 uint32_t key_value_store_size_;
145 uint8_t key_value_store_[0]; // note variable width data at end
146
147 DISALLOW_COPY_AND_ASSIGN(OatHeader);
148};
```

OatHeader 各个成员的含义如下：

uint8_t **magic**[4]: 魔数，为“oat\n”；

uint8_t **version**[4]: OAT 文件版本号；

uint32_t **adler32_checksum**: OAT 头部校验和；

InstructionSet **instruction_set**: 本地机指令集，表示指令的类型；

InstructionSet 在/art/runtime/instruction_set.h 中定义，是一个枚举类型，有 8 种取值，如下：

```
29 enum InstructionSet {
30     kNone,
31     kArm,
32     kArm64,
33     kThumb2,
34     kX86,
35     kX86_64,
36     kMips,
37     kMips64
38 };
```

InstructionSetFeatures **instruction_set_features**: This is a bitmask of supported features per architecture. 位掩码，用来表示每个构架的特性。

InstructionSetFeatures 在/art/runtime/instruction_set.h 中定义，占 32b，如下所示：

```
185 // This is a bitmask of supported features per architecture.
186 class PACKED(4) InstructionSetFeatures {
187     ... ..
225 private:
226     uint32_t mask_; //掩码
227 };
```

uint32_t **dex_file_count**: OAT 文件包含的 dex 文件个数

uint32_t **executable_offset**: oatexec 段开始位置与 oatdata 段开始位置的偏移值。(即：oatexec 段开始位置+executable_offset=oatexec 段开始位置)

uint32_t **interpreter_to_interpreter_bridge_offset** 和 uint32_t **interpreter_to_compiled_code_bridge_offset** : ART 运行时在启动的时候，可以通过-Xint 选项指定所有类的方法都是解释执行的，这与传统的虚拟机使用解释器来执行类方法差不多。同时，有些类方法可能没有被翻成本地机器指令，这时候也要求对它们进行解释执行。这意味着解释执行的类方法在执行的过程中，可能会调用到另外一个也是解释执行的类方法，也可能调用到另外一个按本地机器指令执行的类方法中。OAT 文件在内部提供有两段 trampoline 代码，分别用

来从解释器调用另外一个也是通过解释器来执行的类方法和从解释器调用另外一个按照本地机器执行的类方法。这两段 trampoline 代码的偏移位置就保存在成员变量 `interpreter_to_interpreter_bridge_offset_` 和 `interpreter_to_compiled_code_bridge_offset_`。

`uint32_t jni_dlsym_lookup_offset_`: 类方法在执行的过程中, 如果要调用另外一个方法是一个 JNI 函数, 那么就要通过存在放置 `jni_dlsym_lookup_offset_` 的一段 trampoline 代码来调用。

`uint32_t portable_imt_conflict_trampoline_offset_`: 不知道, 待分析

`uint32_t portable_resolution_trampoline_offset_`:

用来在运行时解析还未链接的类方法的两段 trampoline 代码。其中, `portable_resolution_trampoline_offset_` 指向的 trampoline 代码用于 Portable 类型的 Backend 生成的本地机器指令, 而 `quick_resolution_trampoline_offset_` 用于 Quick 类型的 Backend 生成的本地机器指令。

`uint32_t portable_to_interpreter_bridge_offset_`:

与 `interpreter_to_interpreter_bridge_offset_` 和 `interpreter_to_compiled_code_bridge_offset_` 的作用刚好相反, 用来在按照本地机器指令执行的类方法中调用解释执行的类方法的两段 trampoline 代码。其中, `portable_to_interpreter_bridge_offset_` 用于 Portable 类型的 Backend 生成的本地机器指令, 而 `quick_to_interpreter_bridge_offset_` 用于 Quick 类型的 Backend 生成的本地机器指令。

`uint32_t quick_generic_jni_trampoline_offset_`: 不知道, 待分析

`uint32_t quick_imt_conflict_trampoline_offset_`: 不知道, 待分析

uint32_t quick_resolution_trampoline_offset_

uint32_t quick_to_interpreter_bridge_offset_;

uint32_t image_patch_delta_: 不知道，待分析（该 OAT 文件关联的 Image 被 patch 的数量）

uint32_t image_file_location_oat_checksum_: 用来创建 Image 空间的 OAT 文件的检验和

uint32_t image_file_location_oat_data_begin_: 用来创建 Image 空间的 OAT 文件的 oatdata 段在内存的位置

uint32_t key_value_store_size_: 不知道，待分析(用来创建 Image 空间的文件的路径的大小)

4.2 文件中 oatHeader 分析

字段	值	地址
uint8_t <i>magic</i> _[4]	6F61740A(即"oat\n") 不用转字节序，即大端	1000
uint8_t <i>version</i> _[4]	30343500(即"045") 不用转字节序，即大端	1004
uint32_t <i>adler32_checksum</i> _	93AECF45	1008
InstructionSet <i>instruction_set</i> _	00000003	100C
InstructionSetFeatures <i>instruction_set_features</i> _	00000001	1010
uint32_t <i>dex_file_count</i> _	00000001	1014
uint32_t <i>executable_offset</i> _	000B4000	1018
uint32_t <i>interpreter_to_interpreter_bridge_offset</i> _	00000000	101C
uint32_t <i>interpreter_to_compiled_code_bridge_offset</i> _	00000000	1020
uint32_t <i>jni_dlsym_lookup_offset</i> _	00000000	1024
uint32_t <i>portable_int_conflict_trampoline_offset</i> _	00000000	1028
uint32_t <i>portable_resolution_trampoline_offset</i> _	00000000	102C
uint32_t <i>portable_to_interpreter_bridge_offset</i> _	00000000	1030
uint32_t <i>quick_generic_jni_trampoline_offset</i> _	00000000	1034
uint32_t <i>quick_int_conflict_trampoline_offset</i> _	00000000	1038
uint32_t <i>quick_resolution_trampoline_offset</i> _	00000000	103C
uint32_t <i>quick_to_interpreter_bridge_offset</i> _	00000000	1040

int32_t <i>image_patch_delta_</i>	00000000	1044
uint32_t <i>image_file_location_oat_checksum_</i>	81101B01	1048
uint32_t <i>image_file_location_oat_data_begin_</i>	70FE0000	104C
uint32_t <i>key_value_store_size_</i>	00000190	1050

oatHeader 后面跟了个 key_value_store_，字节数为 *key_value_store_size_*，看具体的值像是 dex 转为 oat 时的参数信息。

字段	值	地址
<i>key_value_store_</i>	dex2oat-cmdline\x00 --zip-fd=6 --zip-location=/data/app/com.cpf.xdefdemo-1/base.apk --oat-fd=7 --oat-location=/data/dalvik-cache/arm/data@app@com.cpf.xdefdemo-1@base.apk@classes.dex --instruction-set=arm --instruction-set-features=div --runtime-arg -Xms64m --runtime-arg -Xmx512m --swap-fd=8 dex2oat-host\x00 Arm\x00 image-location\x00 /data/dalvik-cache/arm/system@framework@boot.art\x00 pic\x00 false\x00 xposed-oat-version\x00 2\x00	开始: 1054 结束: 11E3 长度: 190 即 key_value_store_size_ 的值

4.3 Dex Meta Data

在 oatHeader+key_value_store_后面存的是与 dex 相关的信息，这里我用 DexMetaData 表示，oatHeader 中 *dex_file_count_* 字段是 dex 文件的数目，因而共有 *dex_file_count_* 个 DexMetaData，DexMetaData 的字段如下：

- uint32_t *dex_file_location_size*: dex 文件路径的字节数
- char* *dex_file_location_data*: dex 文件的路径
- uint32_t *dex_file_checksum*: dex 文件的校验和
- uint32_t *dex_file_offset*: dex 文件相对于 oatdata 段开始地址的偏移
- const uint32_t* *methods_offsets_pointer* : 是一个数组，元素共有 class_defs_size_(dex 中类的数目)个，该数组的索引与 dex 中类的索引是一致的，就是说第 0 个类对应 methods_offsets_pointer[0]。元素的值是相对于 oatdata 段开始地址的偏移，比如，dex 中第 0 个类对应的 OatClass 在文件中

的开始地址=methods_offsets_pointer[0] + oatdata 段开始地址。

由上可以知道，整个 **Dex Meta Data** 的字节数= oatHeader->dex_file_count_
*(4+dex_file_location_size+4+4*dex->class_defs_size_)
=oatHeader->dex_file_count_*(12+dex_file_location_size+4*dex->class_defs_size_)

4.4 文件中 Dex Meta Data 分析

从 OatHeader 可知，该文件只包含一个 dex 文件。

字段	值	地址
uint32_t dex_file_location_size	00000025	11E4
char* dex_file_location_data	/data/app/com.cpf.xdefdemo-1/base.apk	11E8
uint32_t dex_file_checksum	6566A1EF	120D
uint32_t dex_file_offset	00000938	1211
const uint32_t* methods_offsets_pointer	00098A60	1215
	00098A64	1219
	00098A98	121D
	00098AB4	1221
	
	0009C344	1931
共有 class_defs_size_(1C8)个 methods_offsets_pointer， 结束地址=1215+1C8*4-1=1934		

由上可知 dex_file 的地址 =oatdata 段开始位置 +dex_file_offset=1000+938=1938。1935,1936,1937 处的字节没有用，用于对齐，接下来 **1938 开始就是 dex**。

Dex 文件(/art/runtime/dex_file.h):

字段	值	地址
file_size_(4b)	00098128	1958
class_defs_size_(4b)	000001C8	1998
dex 地址范围	[1938h, 99A5F]	

由 dex 文件格式知，dex 文件中的 class_defs_size_(4bytes)相对于 dex 开始地址的偏移是 0x60，因而在该文件中，class_defs_size_的偏移=dex_file 地址 +0x60=1998，class_defs_size_为 1C8，从而可以计算出该 DexMetaData 的大小，如上所示；file_size_相对于 dex 开始地址的偏移是 0x20，因而 file_size_的偏移

=dex_file 地址 +0x20=1938h+20h=1958h，得出 dex 文件的地址范围为 [1938h,1938h+file_size_-1]=[1938h, 99A5Fh]。

methods_offsets_pointer[0]+oatdata 段的开始地址 就是 dex 中第 0 个类对应的 OatClass 的开始地址，下面开始分析 OatClass。

4.5 OatClass

/art/runtime/oat_file.h **OatClass**

```
173 class OatClass {
174     ...
201 private:
    ...
209     const OatFile* oat_file_;
210
211     mirror::Class::Status status_; //2bytes
212
213     OatClassType type_; //2bytes
214
215     const uint32_t* bitmap_;
216 //方法的偏移数组,指向相应的 native code
217     const OatMethodOffsets* methods_pointer_;
218
219     friend class OatDexFile;
220 };
```

与 oatClass 相关的类:

/art/runtime/mirror/class.h **mirror::Class::Status** 2 bytes

```
138 enum Status {
139     kStatusRetired = -2,
140     kStatusError = -1,
141     kStatusNotReady = 0,
142     kStatusIdx = 1, // Loaded, DEX idx in super_class_type_idx_ and
    interfaces_type_idx_.
143     kStatusLoaded = 2, // DEX idx values resolved.
144     kStatusResolving = 3, // Just cloned from temporary class object.
145     kStatusResolved = 4, // Part of linking.
146     kStatusVerifying = 5, // In the process of being verified.
147     kStatusRetryVerificationAtRuntime = 6, // Compile time verification
    failed, retry at runtime.
148     kStatusVerifyingAtRuntime = 7, // Retrying verification at runtime.
```

```

149  kStatusVerified = 8, // Logically part of linking; done pre-init.
150  kStatusInitializing = 9, // Class init in progress.
151  kStatusInitialized = 10, // Ready to go.
152  kStatusMax = 11,
153  };

```

/art/runtime/oat.h **OatClassType** 2bytes

```

150// OatMethodOffsets are currently 5x32-bits=160-bits long, so if we can
151// save even one OatMethodOffsets struct, the more complicated encoding
152// using a bitmap pays for itself since few classes will have 160
153// methods.
154enum OatClassType {
155  kOatClassAllCompiled = 0, // OatClass is followed by an OatMethodOffsets
for each method.
156  kOatClassSomeCompiled = 1, // A bitmap of which OatMethodOffsets are
present follows the OatClass.
157  kOatClassNoneCompiled = 2, // All methods are interpreted so no
OatMethodOffsets are necessary.
158  kOatClassMax = 3,
159};

```

- *kOatClassAllCompiled*: 该类中的方法都被编译成了 native code，oatClass 后紧接着是每个方法的 OatMethodOffsets。
- *kOatClassSomeCompiled*: 该类中的部分方法被编译成了 native code，oatClass 后面会有一个 bitmap，接着才是 OatMethodOffsets。
- *kOatClassNoneCompiled*: 该类没有被编译，不需要 OatMethodOffsets

因而，不同的 OatClassType，方法在 const OatMethodOffsets* *methods_pointer_* 数组中的偏移计算方式还不一样。下面看一下获取 OatmethodOffset 的源码实现：

(/art/runtime/oat_file.cc)

```

550const OatMethodOffsets* OatFile::OatClass::GetOatMethodOffsets(uint32_t
method_index) const {
551  // NOTE: We don't keep the number of methods and cannot do a bounds check
for method_index.
552  if (methods_pointer_ == nullptr) {
553    CHECK_EQ(kOatClassNoneCompiled, type_); //没有编译成 native code
554    return nullptr;
555  }
556  size_t methods_pointer_index;
557  if (bitmap_ == nullptr) {
558    CHECK_EQ(kOatClassAllCompiled, type_);
559    methods_pointer_index = method_index; //该类所有方法编译成了 native code
560  } else {

```

```

561 CHECK_EQ(kOatClassSomeCompiled, type_);
562 if (!BitVector::IsBitSet(bitmap_, method_index)) {
563     return nullptr; //该方法没有对应的 native code
564 }
565 size_t num_set_bits = BitVector::NumSetBits(bitmap_, method_index);
566 methods_pointer_index = num_set_bits; //找到相应的索引
567 }
568 const OatMethodOffsets& oat_method_offsets =
methods_pointer_[methods_pointer_index];
569 return &oat_method_offsets;
570}

```

函数参数 `method_index` 是方法在 dex 表示的类中的偏移，比如类的第零个方法 `method_index` 为 0，第一个方法 `method_index` 为 1，依次类推。由于在 `OatClass` 中没有保存类中的方法数目，因而没有检查 `method_index` 的边界。

552~555 行表示，如果 `OatClass` 的类型是 `kOatClassNoneCompiled`，那么该类中的方法没有编译成与之对应的 native code，所以返回空指针。

557~559 行表示，如果 `OatClass` 的类型是 `kOatClassAllCompiled`，那么该类中所有方法都编译成了 native code，索引为 `method_index` 的方法与之对应的 native code 的偏移是 `methods_pointer_[method_index]`。即 `method_index` 与 `methods_pointer_` 数组的下标是对应的。

561~566 行表示，如果 `OatClass` 的类型是 `kOatClassSomeCompiled`，那么该类中部分方法编译成了 native code。因而，dex 类中方法的数目，就是不是 `OatMethodOffsets* methods_pointer_` 数组的大小。为了节约空间，这里 Android 采用了 bitmap 来记录哪些方法被编译成了 native code。bitmap 的具体原理大家可以自行查找，我这里举一个例子希望能帮助大家理解。假设 dex 中某一个类 A 有 34 个方法 (dexMethods)，其中 index 为 1, 8, 33 共 3 个方法有与之对应的 native code，很明显，`methods_pointer_` 数组的大小应该为 3。那么 bitmap 中应该有 64bit（这里 bitmap 中数组的元素类型是 `uint32_t`），如果第 i ($0 \leq i < 64$) 位是 1，那么表示 `dexMethods[i]` 有与之对应的 native code，反之就没有。由前面的假设可知，bitmap 中为 1 的位有 1,8,33。很容易得出，`dexMethods[i]` ($i=1,8,33$) 在 `methods_pointer_` 中的索引 = $\sum_{k=0}^{i-1} \text{bitmap}[k]$ 。有了这个例子，下面来看 565 行函数 `BitVector::NumSetBits` 的实现：(`/art/runtime/base/bit_vector.cc`)

```

383 uint32_t BitVector::NumSetBits(const uint32_t* storage, uint32_t end) {

```



```

384 uint32_t word_end = end >> 5; //就是 end/32
385 uint32_t partial_word_bits = end & 0x1f; //就是 end%32
386
387 uint32_t count = 0u; //为 1 的位的数目
388 for (uint32_t word = 0u; word < word_end; word++) {
389     count += POPCOUNT(storage[word]); //先算前 word_end 个字中为 1 的位的数目
390 } //再算不满一个字的前 partial_word_bits 个位中为 1 的位的数目
391 if (partial_word_bits != 0u) {
392     count += POPCOUNT(storage[word_end] & ~(0xffffffffu <<
partial_word_bits));
393 }
394 return count;
395}

```

/art/runtime/oat.h **OatMethodOffsets** 4bytes

```

163class PACKED(4) OatMethodOffsets {
164 public:
165     OatMethodOffsets(uint32_t code_offset = 0);
166
167     ~OatMethodOffsets();
168
169     uint32_t code_offset_; //native code 相对于 oatdata 段的偏移
170};

```

4.5.1 文件中分析 OatClass

methods_offsets_pointer[0]+oatdata 段的开始地址 就是 dex 中第 0 个类对应的 OatClass 的开始地址。即 98A60+1000=99A60。下面解析第 0 个 class:

字段	值	地址
status	000A(kStatusInitialized)	99A60
OatClassType	0002(kOatClassNoneCompiled)	99A62
bitmap_size(4b)	当 OatClassType 为 kOatClassSomeCompiled 时才有	
bitmap		
OatMethodOffset(4b)		
... ..		
OatMethodOffset 的个数是该类方法被编译为 native code 的个数		

由于 OatClassType 为 kOatClassNoneCompiled，因而没有 bitmap 和 OatMethodOffset。

再看第二个 class:

字段	值	地址
----	---	----

status	0008	99A64
OatClassType	0001(kOatClassSomeCompiled)	99A66
bitmap_size(4b)	00000004	99A68
bitmap	000007FE	99A6C
OatMethodOffset(4b)	000B401D	99A70
OatMethodOffset(4b)	000B4075	99A74
... ..		
OatMethodOffset(4b)	000B47E5	99A94
OatMethodOffset 的个数是该类方法被编译为 native code 的个数 该 class 的结束地址=99A70+A*4-1=99A97		

在解析时，bitmap 是按字节读取的，因而读取的时候 byte *bitmap 是 FE070000；但是在创建 OatClass 对象时，会将 bitmap 强制转为 uint32_t * bitmap。所以，在实际算方法偏移时，uint32_t * bitmap 为 000007FE，二进制为：0111 1111 1110，共有 10 个位为 1，因而 methods_pointer_数组元素(OatMethodOffset)的个数为 Ah。

从 OatClass 中获取指定方法的 native code 时，返回的是 OatMethod 对象，具体源码如下：

```

572 const OatFile::OatMethod OatFile::OatClass::GetOatMethod(uint32_t
method_index) const {
573     const OatMethodOffsets* oat_method_offsets =
GetOatMethodOffsets(method_index); //获取方法偏移
574     if (oat_method_offsets == nullptr) {
575         return OatMethod(nullptr, 0);
576     }
577     if (oat_file_>IsExecutable() ||
578         Runtime::Current() == nullptr || // This case applies for
oatdump.
579         Runtime::Current()>IsCompiler()) {
580         return OatMethod(oat_file_>Begin(), oat_method_offsets->code_offset_);
581     } else {
582         // We aren't allowed to use the compiled code. We just force it down the
interpreted version.
583         return OatMethod(oat_file_>Begin(), 0);
584     }
585 }

```

```

92 class OatMethod {
93     ... ..
166     const byte* begin_; //oatdata 段的开始地址:1000h
167

```

```
168     uint32_t code_offset_; //就是 OatMethodOffsets->code_offset_  
171 };
```

这样就可以通过 `OatMethod` 来定位到方法的 native code 了 (`begin_+code_offset_`)。

5 总结

分析到这里，相信大家对 Oat 的文件结构已经有了一定的认识。最后总结一下：

- Oat 文件其实就是类型为 shared object 的 Elf 文件，首先有一个 Elf Header，Elf Header 中 `e_phoff` 字段指向了 program header table，`e_shoff` 字段指向了 section header table；
- section header table 中有一个名字为 .dynsym 的 section head，该 section head 描述的 section 是符号表；
- Oat 文件导出了 3 个符号：oatdata，oatexec，oatlastword；
- oatdata 指向 oatdata 段，该区域存的是只读数据，包括 OatHeader，dex 相关信息，dex 类中方法与 native code 的映射关系（由 OatClass 表示），通过 OatClass 可以找到对应方法的 native code；
- oatexec 指向 oatexec 段，该区域存的是方法的 native code，是可执行的；
- oatlastword 指向 oatexec 段的最后一个字的开始地址。即 oatexec 段的结束地址 = `oatlastword->st_value+3`。

（完）