DEX基本结构

Dalvik 可执行文件格式 | Android Open Source Project

XRefAndroid - Support Android 16.0 & OpenHarmony 5.0 (AndroidXRef/AospXRef)

Dex文件结构-ReadDex解析器实现 - 吾爱破解 - 52pojie.cn

Android Dex文件详解-CSDN博客

DEX文件介绍

什么是DEX文件

Dex（**Dalvik Executable**）文件是 Android 系统的可执行文件格式，用于存放被 Dalvik 虚拟机或 ART（Android Runtime）解释执行的字节码。它是由 `.class` Java 编译文件通过 `dx` 或 `d8` 工具转换合并后生成的。

Dex 文件的优点

Java 虚拟机（JVM）基于栈，而 Android 虚拟机（Dalvik/ART）基于寄存器。基于寄存器的实现需要不同的指令集，因此需要一个不同于 JVM Class 文件的新格式。此外，将多个 `.class` 文件合并成一个 `.dex` 文件有助于**共享常量、减少冗余**，从而显著减小文件体积并提高运行时效率。

▼ 三种虚拟机

JVM

JVM是Java Virtual Machine的简称，即Java虚拟机，它本质是一层软件抽象，在这之上才可以运行Java程序。

执行文件：`.class` 文件。每个Java源文件编译后都会生成一个对应的 `.class` 文件。

Java文件经过编译后会生成JVM字节码，和C语言编译后生成的汇编语言不同，C编译成的汇编语言可以直接在硬件上跑，但是Java编译生成的字节码是在JVM上跑，需要由JVM把字节码翻译成机器指令。

执行模式：传统的JVM使用**解释器**来逐条执行字节码。为了提升性能，现代JVM会使用**JIT编译器**，它会监控代码的执行频率，将**热点代码**（频繁执行的方法）在运行时动态编译成本地机器码，后续执行直接运行机器码，极大提升效率。

架构：**基于栈**。指令的操作数存放在操作数栈中，指令从栈顶获取参数并将结果压回栈顶。例如，加法指令 `iadd` 会从栈顶弹出两个整数，相加后再将结果压回栈顶。这种设计非常通用，易于实现跨平台。

也是由于这个JVM在操作系统上屏蔽了底层实现的差异，从而有了Java的跨平台特性。

DVM

DVM是Dalvik Virtual Machine的简称，是Android4.4及以前使用的虚拟机，所有Android程序都运行在Android系统进程中，每个进程对应着一个Dalvik虚拟机实例。

JVM和DVM都提供了对对象生命周期管理，堆栈管理，安全和异常管理及垃圾回收等重要功能。

但是DVM却不能和JVM一样能直接运行Java字节码，它只能运行.dex文件，而这个.dex文件则是由Java字节码通过Android的dx工具生成的文件。

执行文件：`.dex` **(Dalvik Executable) 文件**。这是DVM与JVM最显著的区别。Android构建工具 (`dx` / `d8`) 会将所有Java编译后的 `.class` 文件进行转换、合并、优化，最终生成一个或几个 `.dex` 文件。

架构：**基于寄存器**。虚拟CPU有多个虚拟寄存器，指令直接操作这些寄存器。基于寄存器的指令集通常比基于栈的指令集更密集、更快速，完成相同操作所需的指令数更少。

例如：计算 `a = b + c`。

- JVM (栈): `iload b`, `iload c`, `iadd`, `istore a` (4条指令)。
- DVM (寄存器): `add-int a, b, c` (1条指令)。

执行模式：DVM最初是一个**纯解释器**，它直接解释执行 `.dex` 字节码。从Android 2.2 (Froyo) 开始，DVM引入了**JIT编译**，在应用运行时分析性能，将热点代码编译成本地机器码，以提高运行速度。

进程模型：每个Android应用运行在独立的Linux进程中，且每个进程都运行着一个独立的DVM实例。这提供了应用间的隔离和安全性。

ART

ART是在Android 4.4 (KitKat) 中作为实验性功能引入，并从 Android 5.0 (Lollipop) 开始**完全取代DVM**的新运行时环境。ART**延续了DVM的字节码格式**（仍使用 `.dex` 文件），但彻底改变了执行方式。

前面说了Dalvik虚拟机会在APP打开时去运行.dex文件，而这个是实时的，也就是JIT特性(Just In Time)，这也就会导致在启动APP时会先将.dex文件转换成机器码，这就导致了APP启动慢的问题。

而ART虚拟机有个很好的特性叫做AOT(ahead of time)，这个特性可以在安装APK的时候将dex直接处理成可直接供ART虚拟机使用的机器码，ART虚拟机将.dex文件转换成可直接运行的.oat文件，而且ART虚拟机天生支持多dex，所以ART虚拟机可以很大提升APP的冷启动速度。

- 在DVM中，编译发生在**运行时**（JIT）。

- 在ART中，编译发生在**应用安装时**。当你在设备上安装APK时，ART的工具（`dex2oat`）会将APK中的 `.dex` 字节码**预先编译成本地机器码**，生成 `.oat` 文件。
- **优点**：
  - **性能大幅提升**：由于直接执行本地机器码，应用启动速度和运行速度都远高于解释执行的DVM。
  - **功耗降低**：避免了运行时解释和JIT编译的CPU开销。

除了这个优点外，ART还提升了GC速度，提供功能更全面的Debug特性。同样因为本地机器码比 `.dex` 字节码大得多。所以需要更长的安装时间和更大的存储控件占用。

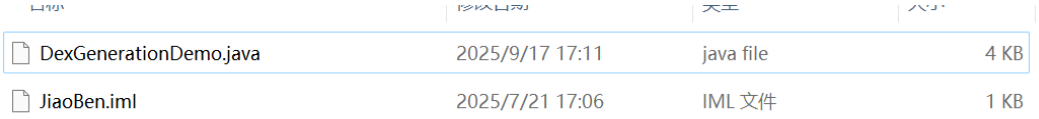
- **演进：混合运行时 (JIT + AOT)**

  由于纯AOT的缺点，从Android 7.0 (Nougat) 开始，ART引入了一种**混合模式**，结合了JIT和AOT的优点：

  1. **初次安装**：应用快速安装，**不进行全面的AOT编译**。ART首先使用一个**解释器**来执行应用。
  2. **运行分析**：同时，一个**JIT编译器**会开始工作，分析运行时的热点代码，并将分析结果（哪些方法是热点）存储起来。
  3. **后台编译**：当设备空闲且充电时，ART会根据JIT收集的分析数据，**只对热点代码进行AOT编译**，将其转换为高效的本地机器码。
  4. **后续运行**：下次运行应用时，编译好的热点代码直接以机器码运行，非热点代码继续解释执行。

Dex文件如何生成

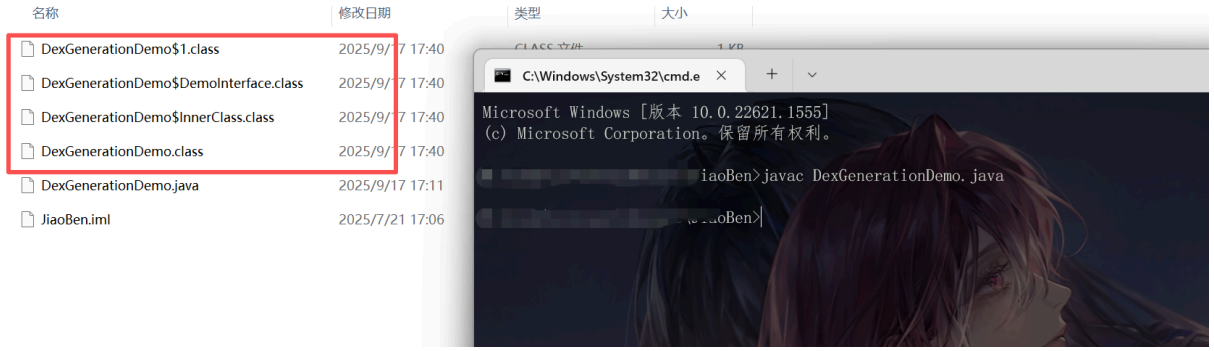
Java/Kotlin 源代码 —编译→ .class 文件 —合并、优化→ .dex 文件



1. 编译 Java 代码为 .class 文件

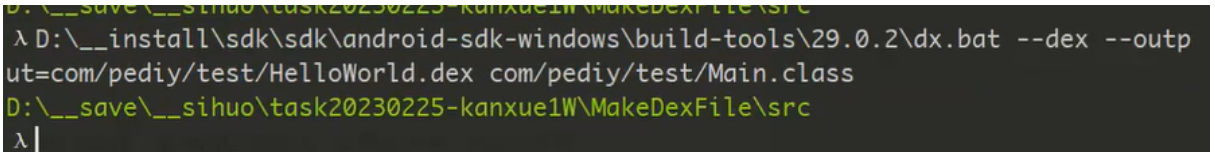
```
javac DexGenerationDemo.java
```

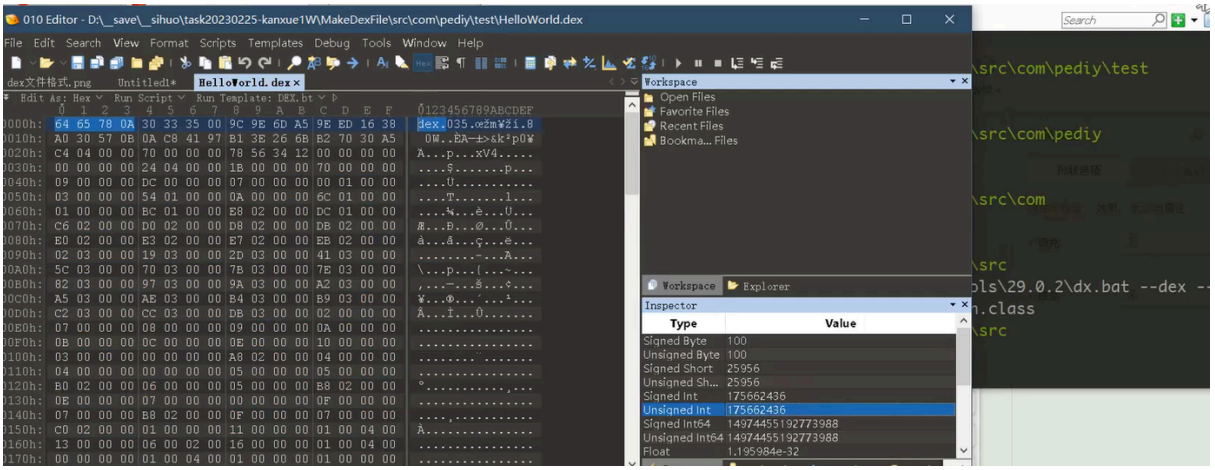
这将生成多个 .class 文件（主类、内部类和匿名类）。



2. 使用 dx 工具生成 DEX 文件（传统方式）

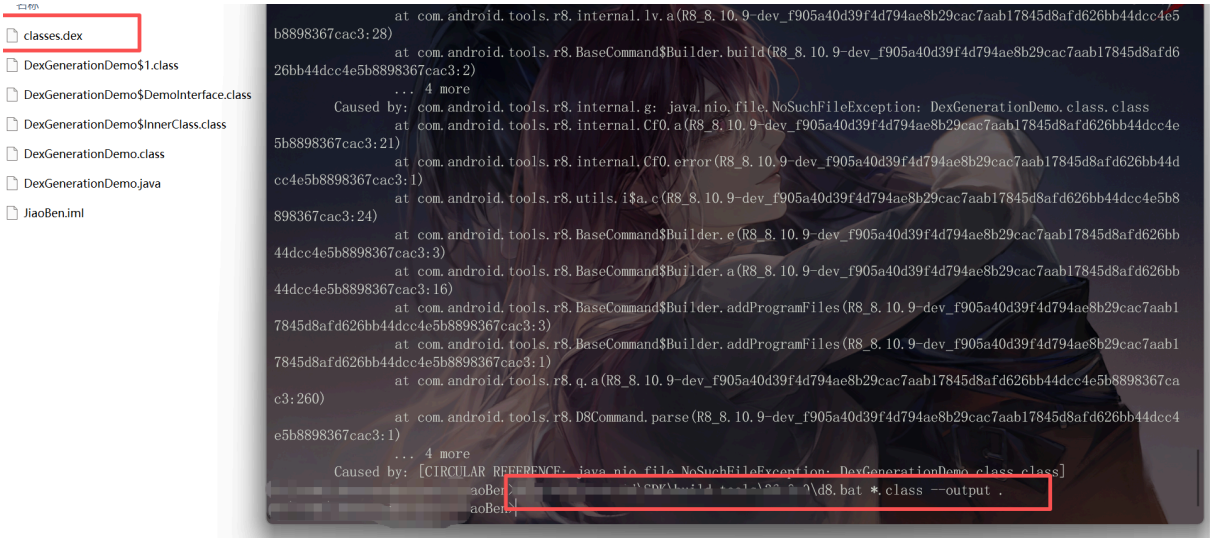
```
# 找到Android SDK中的dx工具路径# 通常位于: $ANDROID_HOME/build-tools/<version>/dx
dx --dex --output=classes.dx.dex *.class
```

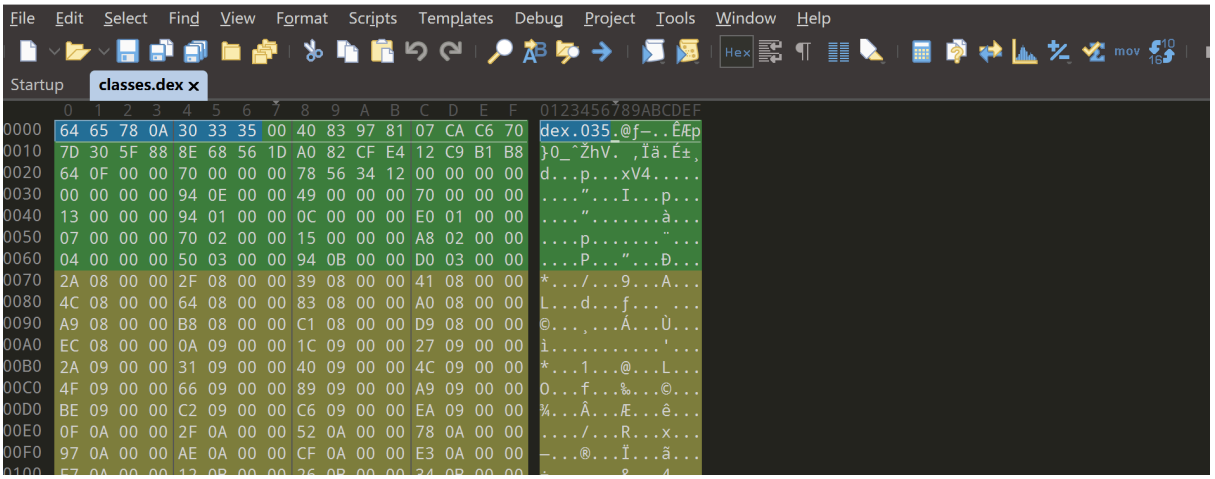




3. 使用 d8 工具生成 DEX 文件（现代方式）

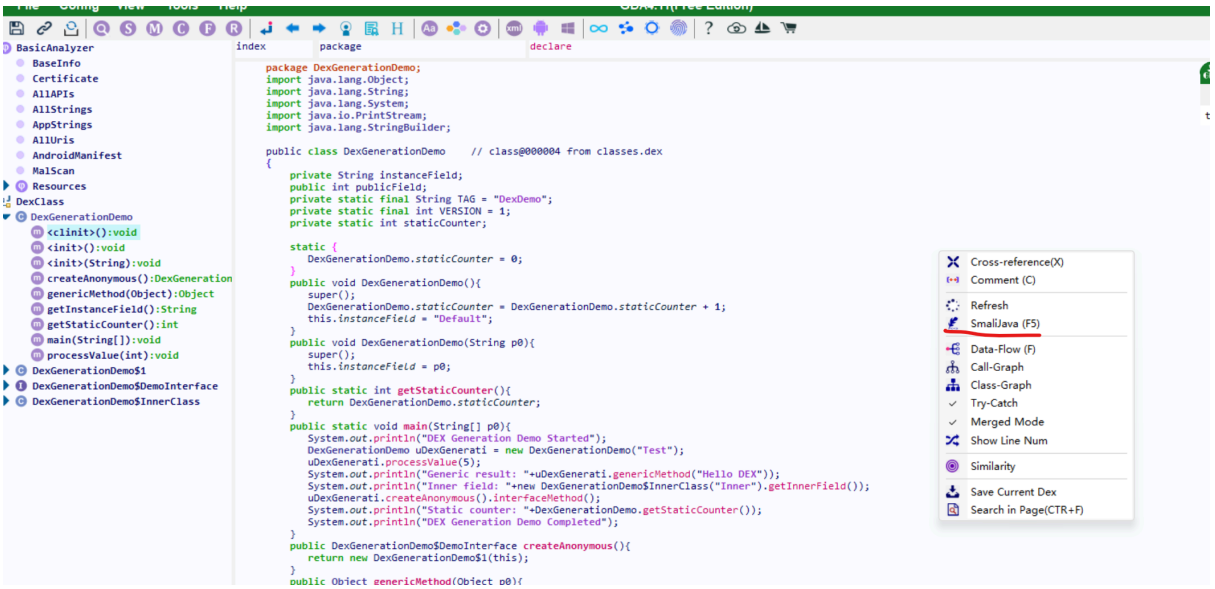
```
# 找到Android SDK中的d8工具路径# 通常位于: $ANDROID_HOME/build-tools/<version>/d8
d8 *.class --output .
```



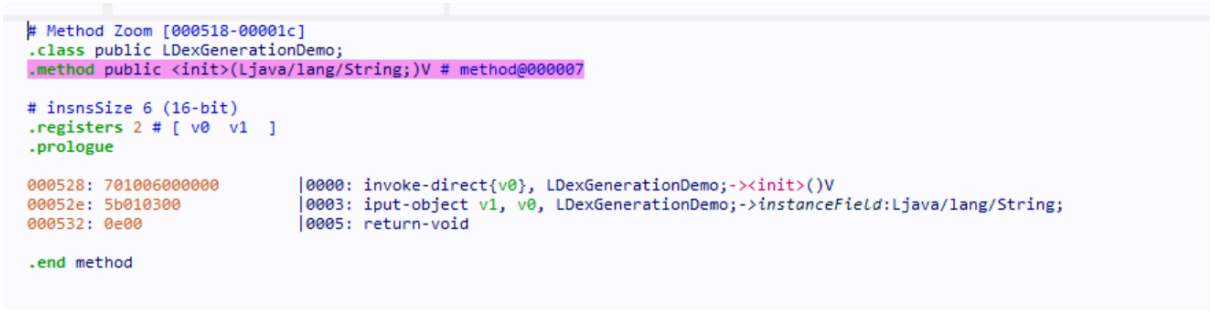


反编译查看smail代码

- 反编译看看smali汇编
  https://source.android.com/docs/core/runtime/instruction-formats?hl=zh-cn
  https://source.android.com/docs/core/runtime/dex-format?hl=zh-cn
- 其他名词解释：
  odex(5.0引入)、oat(elf，包含odex内容)、vdex(8.0引入)



显示字节码



Dex 文件的基本结构

DEX文件中的数据类型

1. 文件格式底层数据类型 (Primitive Data Types)

这些是构成 DEX 文件二进制结构的基石，用于存储长度、偏移量、索引等元信息。

数据类型	大小	描述	例子/用途
`ubyte`	1 字节	无符号字节	文件魔数的一部分（如 `dex`）
`ushort`	2 字节	无符号短整型	字符串长度、小尺寸的数量字段
`uint`	4 字节	无符号整型	**最常用的类型**。用于存储偏移量（offsets）、大小（sizes）、索引（indexes）、CRC 校验和等。例如 `string_ids_size`, `string_ids_off`
`ulong`	8 字节	无符号长整型	签名（SHA-1）、长整型常量

数据类型	大小	描述	例子/用途
`sleb128`	1-5 字节	**有符号** Little Endian Base 128	用于紧凑地存储有符号整数。主要用于 **`.dex` 文件内部**的 `class_data_item` 结构，存储字段、方法的索引差值等。
`uleb128`	1-5 字节	**无符号** Little Endian Base 128	用于紧凑地存储无符号整数。用途非常广泛：1. `string_data_item` 中的 **字符串长度** 2. `class_data_item` 中的静态/实例字段数、方法数等 3. `encoded_field` 和 `encoded_method` 中的字段/方法索引4. 调试信息
`uleb128p1`	1-5 字节	无符号 LEB128 值加 1	用于存储有符号数值的非负形式，常用于 `debug_info_item`

```
1.dex 文件中的数据类型
    ul/uint8_t=>表示 1 字节的无符号数
    u2/uint16_t=>表示 2 字节的无符号数
    u4/uint32_t=>表示 4 字节的无符号数
    u8/uint64_t=>表示 8 字节的无符号数
    sleb128=>有符号 leb128，可变长度为 1-5 字节
    uleb128=>无符号符号 leb128，可变长度为 1-5 字节
    uleb128p1=>无符号符号 leb128 值加 1，可变长度为 1-5 字节
```

uleb128、sleb128 和 uleb128p1

在程序中，一般使用 32 位比特位来表示一个整型的数值。不过，一般能够使用到的整数值都不会太大，使用 32 比特位来表示就有点太浪费了。对于普通计算机来说，这没什么问题，毕竟存储空间那么大。但是，对于移动设备来说，存储空间和内存空间都非常宝贵，不能浪费，能省就省。

Android 的 Dalvik 虚拟机中，就使用了 uleb128（Unsigned Little Endian Base 128）、uleb128p1（Unsigned Little Endian Base 128 Plus 1）和 sleb128（Signed Little Endian Base 128）编码来解决整形数值占用空间大小浪费的问题（在 Dalvik 虚拟机中只使用这三种编码来表示 32 位整形数值）。

首先，我们来看看 uleb128 编码是怎么回事。要了解原理，最简单的方法，还是阅读代码。Dalvik 使用 readUnsignedLeb128 函数来尝试读取一个 uleb128 编码的数值（代码位于 dalvik\libdex\Leb128.h 中）

LEB128

LEB128（“**L**ittle-**E**ndian **B**ase **128**”）表示任意有符号或无符号整数的可变长度编码。该格式借鉴了 DWARF3 规范。在 `.dex` 文件中，LEB128 仅用于对 32 位数字进行编码。

每个 LEB128 编码值均由 1-5 个字节组成，共同表示一个 32 位的值。每个字节均已设置其最高有效位（序列中的最后一个字节除外，其最高有效位已清除）。每个字节的剩余 7 位均为载荷，即第一个字节中有 7 个最低有效位，第二个字节中也是 7 个，依此类推。对于有符号 LEB128（`sleb128`），序列中最后一个字节的最高有效载荷位会进行符号扩展，以生成最终值。在无符号情况（`uleb128`）下，任何未明确表示的位都会被解译为 `0`。

双字节 **LEB128** 值的按位图

第一个字节								第二个字节							
`1`	bit_6	bit_5	bit_4	bit_3	bit_2	bit_1	bit_0	`0`	bit_{13}	bit_{12}	bit_{11}	bit_{10}	bit_9	bit_8	b

变体 `uleb128p1` 用于表示一个有符号值，其表示法是编码为 `uleb128` 的值加 1。这使得 `-1` 的编码（或被视为无符号值 `0xffffffff`）成为一个单字节（但没有任何其他负数），并且该编码在下面这些情况下非常实用：所表示的数值必须为非负数或 `-1`（或 `0xffffffff`）；不允许任何其他负值（或不太可能需要使用较大的无符号值）。

以下是这类格式的一些示例：

编码序列	As `sleb128`	As `uleb128`	As `uleb128p1`
00	0	0	-1
01	1	1	0

```
DEX_INLINE int readUnsignedLeb128(const u1** pStream) {
    const u1* ptr = *pStream;
    int result = *(ptr++);

    if (result > 0x7f) {
        int cur = *(ptr++);
        result = (result & 0x7f) | ((cur & 0x7f) << 7);
        if (cur > 0x7f) {
            cur = *(ptr++);
            result |= (cur & 0x7f) << 14;
            if (cur > 0x7f) {
                cur = *(ptr++);
                result |= (cur & 0x7f) << 21;
                if (cur > 0x7f) {
                    cur = *(ptr++);
                    result |= cur << 28;
                }
            }
        }
    }

    *pStream = ptr;
    return result;
}
```

▼ 代码解析

`INLINE int readUnsignedLeb128(const u1** pStream)`

- `INLINE`：宏，通常是 `static inline`，用于建议编译器将函数内联展开，提高性能。
- `const u1** pStream`：指向字节流指针的指针。
- 返回值是解码后的 `int` 类型整数

`const u1* ptr = *pStream;`

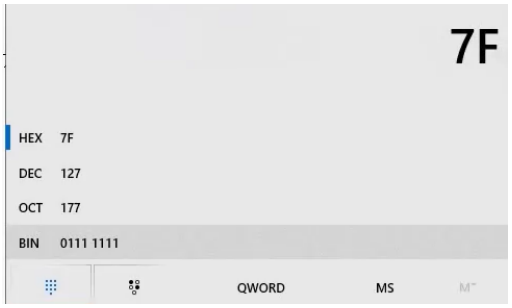
- 从传入的指针中取出当前读取位置。

`int result = *(ptr++);`

- 读取第一个字节，并将其值作为初始结果。
- `ptr++` 表示读取后指针向后移动一个字节。

`if (result > 0x7F) {`

LEB128 是**压缩整数编码方式**，每个字节只用低 7 位存储数据，最高位（bit 7）作为**继续标志位**：



- 判断第一个字节的最高位是否为 1（即是否大于 0x7F），如果是，说明还有后续字节。

```
int cur = *(ptr++);
result = (result & 0x7F) | ((cur & 0x7F) << 7);
```

- 读取第二个字节 `cur`，并将其低 7 位左移 7 位，与之前的结果合并。
- 这一步是将第二个字节的低 7 位作为高 7 位拼接到结果中。
- 过程中的与运算常用来清0，在这里取出后面的7个bit。位或用来将左移后的数值与上面的7bit相加。

```
if (cur > 0x7F) {
  cur = *(ptr++);
  result |= (cur & 0x7F) << 14;
```

- 如果第二个字节的最高位也是 1，继续读取第三个字节，并将其低 7 位左移 14 位拼接到结果中。

```
    if (cur > 0x7F) {
      cur = *(ptr++);
      result |= (cur & 0x7F) << 21;
```

- 同理，继续读取第四个字节，左移 21 位拼接。

```
      if (cur > 0x7F) {
        cur = *(ptr++);
        result |= cur << 28;
```

- 读取第五个字节，并将其**整个字节**左移 28 位拼接。
  - ⚠️ **注意**：这里 `cur` 没有取低 7 位，而是整个字节都参与运算。
  - 这可能导致**符号扩展问题**，因为 `int` 是带符号类型。
- ```
  pStream = ptr;
  return result;
  ```
- 更新传入的指针位置，指向下一个未读取的字节。
- 返回解码后的整数。

---

2. Java 世界的数据描述类型 (Descriptor Types)

a) 基本类型描述符 (Primitive Descriptors)
  ```

DEX 文件使用单个字符来表示 Java 基本类型。

字符	Java 类型
V	void
Z	boolean
B	byte
S	short
C	char
I	int
J	long
F	float
D	double

b) 引用类型描述符 (Reference Descriptors)

格式	例子	含义
Lpackage/name/ClassName;	Ljava/lang/String;	类类型
[	[I , [Ljava/lang/Object;	数组类型

c) 方法描述符 (Method Prototypes)

方法描述符定义了方法的**返回类型**和**参数列表**。它由括号内的参数类型列表和后面的返回类型组成。

格式： (参数类型1参数类型2...)返回值类型

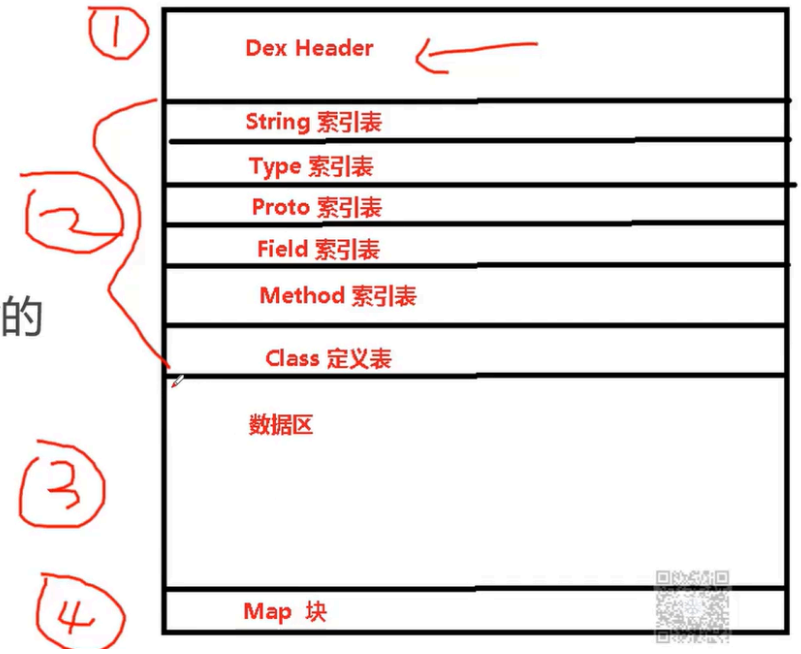
方法声明	描述符
void main(String[] args)	([Ljava/lang/String;)V
String toString()	()Ljava/lang/String;
int add(int a, int b)	(II)I

DEX文件基本结构概括

Dex 文件的设计非常精巧，它由一个文件头（Header）和多个查找表（Lookup Tables/Offsets）以及实际的数据区组成。其核心思想是**通过索引来访问数据**，最大限度地共享和复用数据，减少文件体积。

◆ 总体结构

◆ 每个区块干啥的



◆ 目录：

字符串表索引: header->偏移->字符串表(字节偏移数组->ULEB128 转String)

类型表索引：header->偏移->通过下标索引字符串表

方法原型表索引:header->偏移->结构体数组->解析[返回类型、参数]等

域表索引：header->偏移->结构体数组->解析[成员变量所属类、类型、名字]等

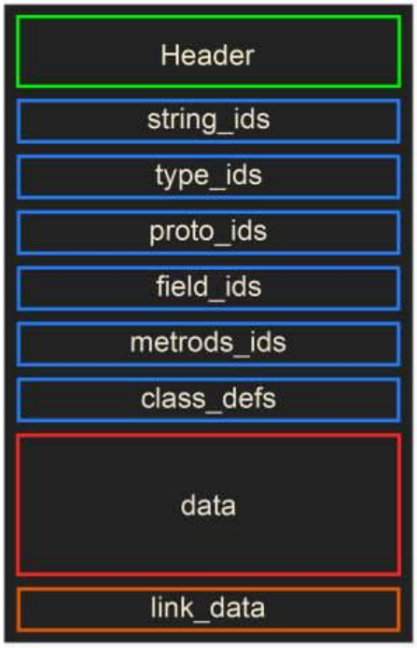
方法表索引：header->偏移->结构体数组->解析[方法的所属类、原型、名字]等

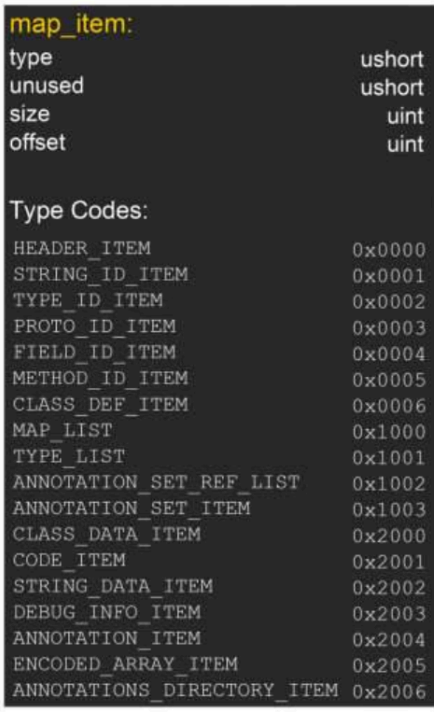
类定义：header->偏移->结构体数组->解析[方法的所属类、原型、名字]等

数据区：不用管

Map区块：header->偏移->结构体数组->解析[各个映射块的地址偏移、大小]等

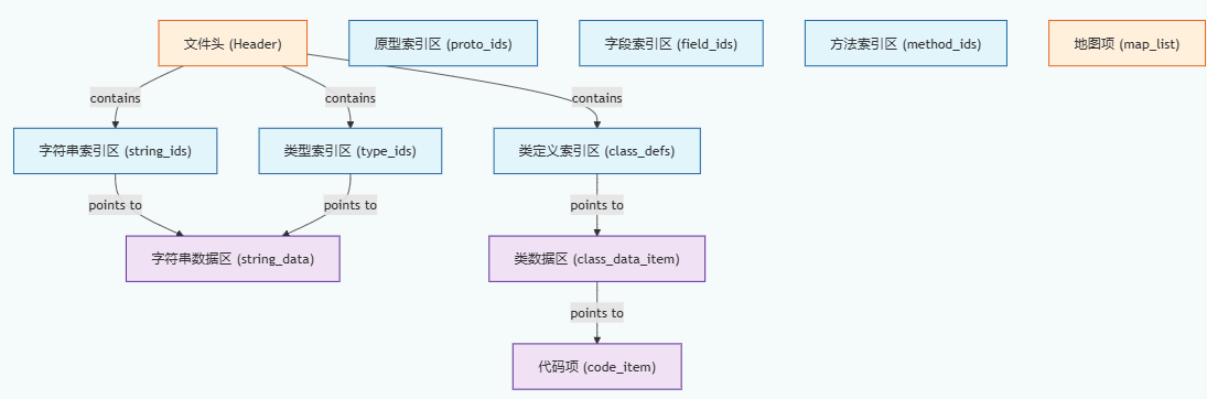
一个标准的 Dex 文件结构可以大致如下：





结构名称	描述
Dex Header	**文件头**，包含整个文件的元信息、魔数、签名以及其它数据段的偏移量和大小。
String Table	**字符串表**，存储代码中所有用到的字符串的偏移量指针。
String Data	**字符串数据区**，实际存储字符串内容（以 UTF-8 或 MUTF-8 格式存储，并以 `\0` 结尾）。
Type Table	**类型表**，存储所有类的描述符（如 `Ljava/lang/String;`）的字符串索引。
Proto Table	**方法原型表**，存储方法的原型信息（返回类型和参数列表）。
Field Table	**字段表**，存储所有字段的定义（所属类、类型、名称）。
Method Table	**方法表**，存储所有方法的定义（所属类、原型、名称）。
Class Def Table	**类定义表**，存储类的详细信息（类名、访问标志、超类、接口、注解、类数据偏移等）。
Class Data	**类数据区**，存储类的具体实现，包括静态/实例字段、虚方法/直接方法及其代码。**这部分数据不是以固定条目存储，而是为了节省空间采用 LEB128 编码的变长结构**。
Code Area	**代码区**，存储方法的具体字节码指令（Dalvik 指令集）以及指令所需的辅助数据（如寄存器数量、try-catch 信息、调试信息等）。
Link Data	**链接数据区**，用于静态链接的数据，通常不对内存中的 Dex 文件使用。

深入理解DEX结构




```
/*
495  * Structure representing a DEX file.
496  *
497  * Code should regard DexFile as opaque, using the API calls provided he
re
498  * to access specific structures.
499  */
500 struct DexFile {
501     /* directly-mapped "opt" header */
502     const DexOptHeader* pOptHeader;
503
504     /* pointers to directly-mapped structs and arrays in base DEX */
505     const DexHeader*    pHeader;
506     const DexStringId*  pStringIds;
507     const DexTypeId*    pTypeIds;
508     const DexFieldId*   pFieldIds;
509     const DexMethodId*  pMethodIds;
510     const DexProtoId*   pProtoIds;
511     const DexClassDef*  pClassDefs;
512     const DexLink*      pLinkData;
513
514     /*
515      * These are mapped out of the "auxillary" section, and may not be
516      * included in the file.
517      */
518     const DexClassLookup* pClassLookup;
519     const void*         pRegisterMapPool;       // RegisterMapClassPool
520
521     /* points to start of DEX file data */
522     const u1*           baseAddr;
```

```
523
524     /* track memory overhead for auxillary structures */
525     int             overhead;
526
527     /* additional app-specific data structures associated with the DEX */
528     //void*           auxData;
529 };
```

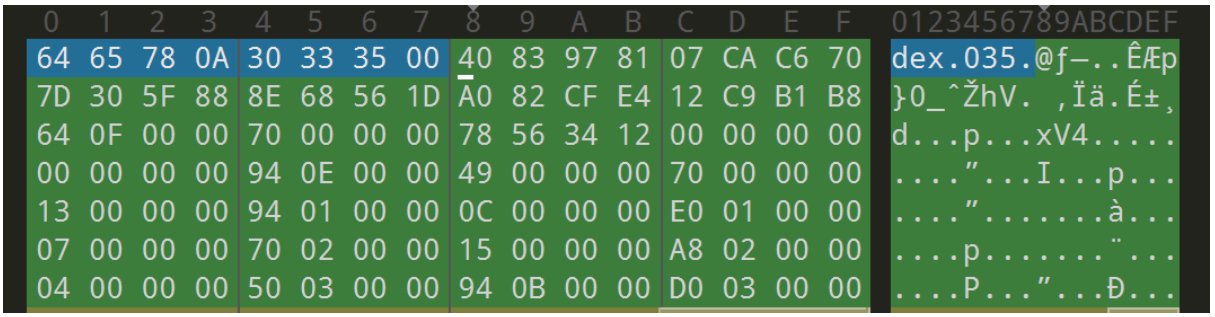
1. DexHeader（文件头）

```
 /*
214  * Direct-mapped "header_item" struct.
215  */
216 struct DexHeader {
217     u1  magic[8];           /* includes version number */
218     u4  checksum;           /* adler32 checksum */
219     u1  signature[kSHA1DigestLen]; /* SHA-1 hash */
220     u4  fileSize;           /* length of entire file */
221     u4  headerSize;         /* offset to start of next section */
222     u4  endianTag;
223     u4  linkSize;
224     u4  linkOff;
225     u4  mapOff;
226     u4  stringIdsSize;
227     u4  stringIdsOff;
228     u4  typeIdsSize;
229     u4  typeIdsOff;
230     u4  protoIdsSize;
231     u4  protoIdsOff;
232     u4  fieldIdsSize;
233     u4  fieldIdsOff;
234     u4  methodIdsSize;
235     u4  methodIdsOff;
236     u4  classDefsSize;
237     u4  classDefsOff;
238     u4  dataSize;
239     u4  dataOff;
240 };
```

文件头固定为 `0x70` 字节，包含了描述整个文件的关键信息。可以用 `010 Editor` 等二进制工具查看，其结构如下（部分重要字段）：

偏移	大小	含义	说明
0x0	8 bytes	**magic**	魔数，标识这是一个 Dex 文件。格式：`dex\n035\0` 或 `dex\n038\0` 等。
0x8	4 bytes	**checksum**	文件校验和，用于检查文件完整性。
0xC	20 bytes	**signature**	SHA-1 签名，用于唯一标识该文件。
0x20	4 bytes	**file_size**	整个 Dex 文件的大小。
0x24	4 bytes	**header_size**	文件头本身的大小（通常为 0x70）。
0x28	4 bytes	**endian_tag**	字节序标记，默认 `0x12345678` 表示小端序。
…	…	…	…
0x3C	4 bytes	**string_ids_size**	字符串索引表的条目数。
0x40	4 bytes	**string_ids_off**	**字符串索引表**在文件中的起始偏移量。
0x44	4 bytes	**type_ids_size**	类型索引表的条目数。
0x48	4 bytes	**type_ids_off**	**类型索引表**在文件中的起始偏移量。
…	…	…	…
0x60	4 bytes	**class_defs_size**	类定义表的条目数。
0x64	4 bytes	**class_defs_off**	**类定义表**在文件中的起始偏移量。

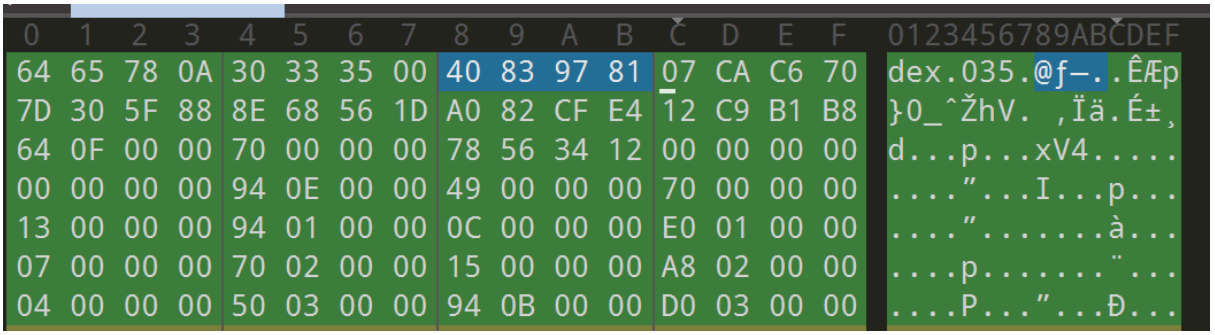
`u1 magic[8];`



魔数，标识这是一个 Dex 文件。

`u4 checksum;`

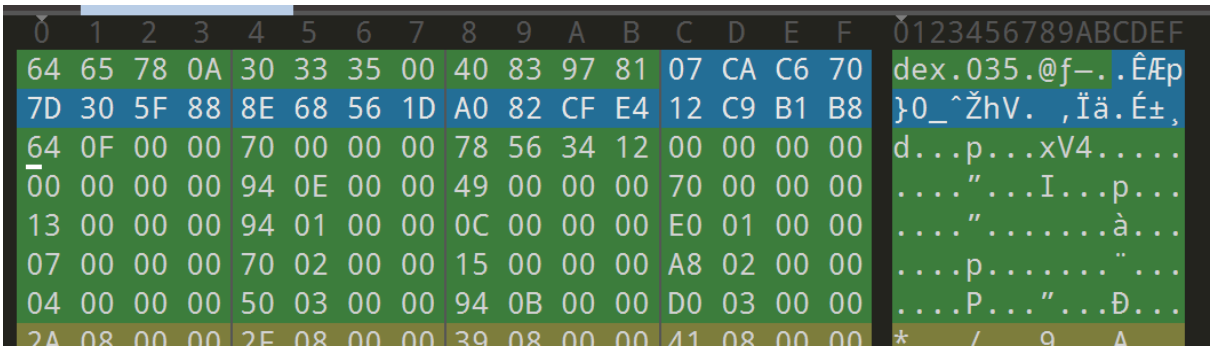
adler32校验，用来快速校验一个dex文件



所以这四个字节代表adler32的值。

`u1 signature[kSHA1DigestLen];`

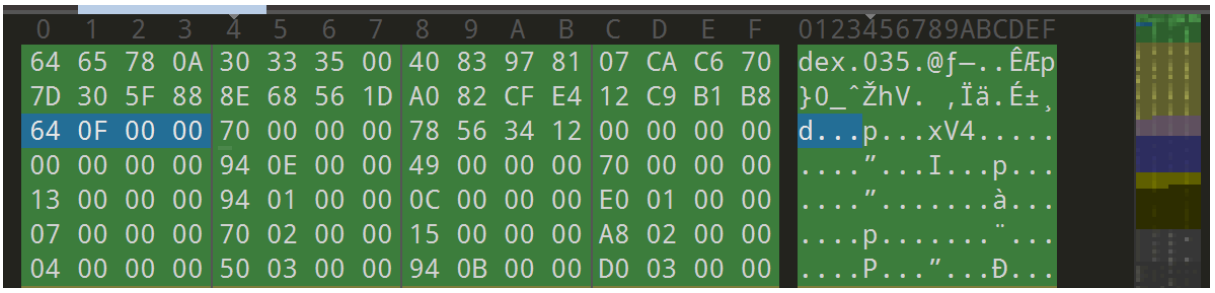
SHA-1签名，用于唯一标识该文件。它是定长的20字节



对后面整体文件进行摘要

`u4 fileSize;`

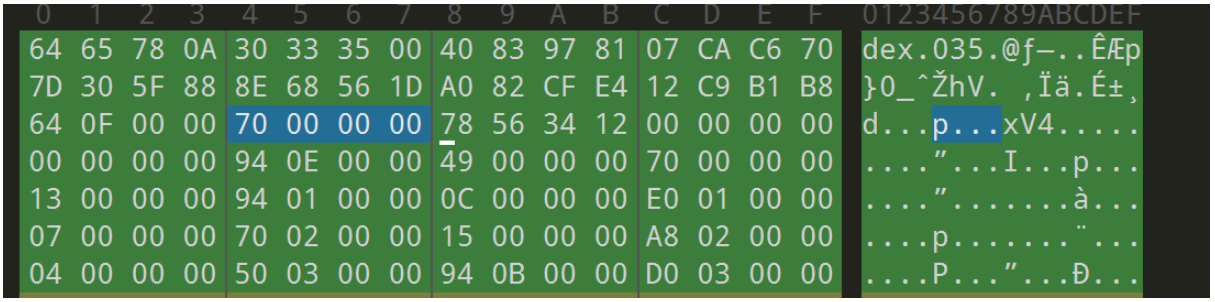
整个dex文件的大小（dex文件为小端字节序）



所以该文件的整体长度为0xF64，也就是3940个字节。

`u4 headerSize;`

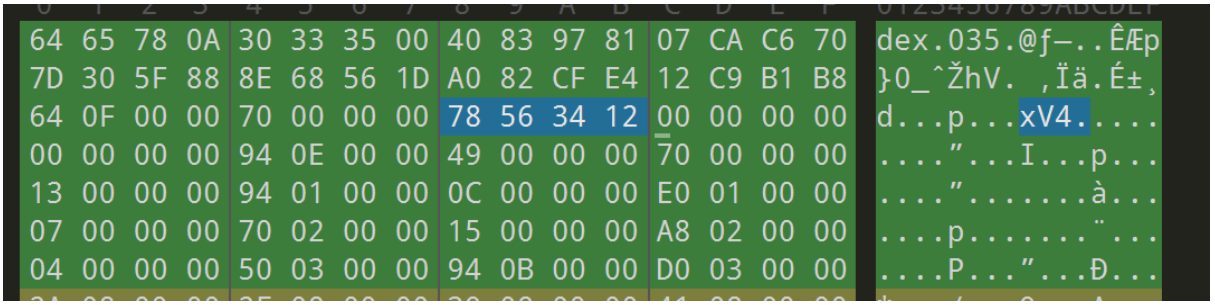
DexHeader结构体大小



0x70，该大小也是固定的。

`u4 endianTag;`

表示字节序列标记

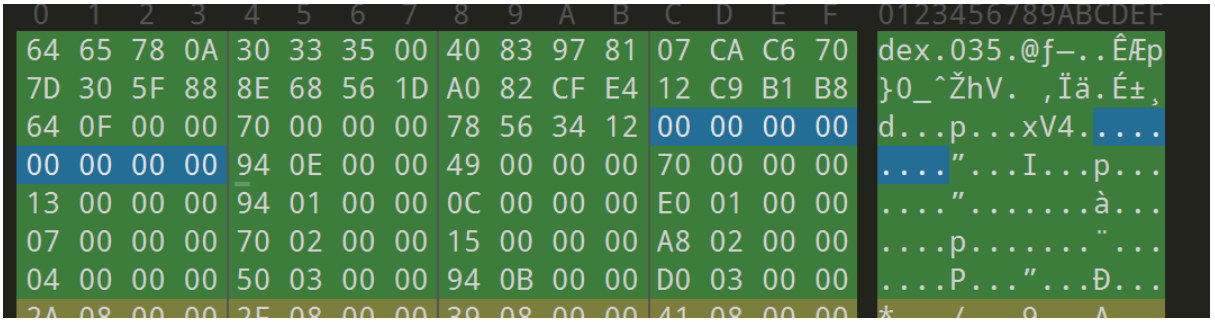


代表小端字节序，真实值是0x12345678

`u4 linkSize;`

`u4 linkOff;`

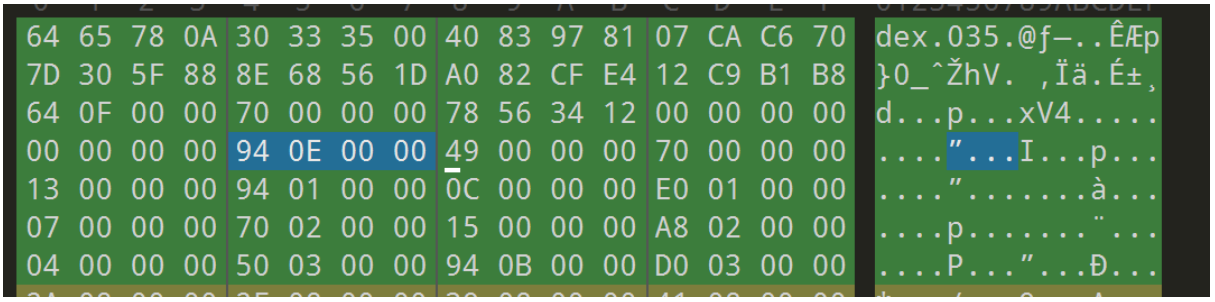
分别表示链接段大小和链接段偏移



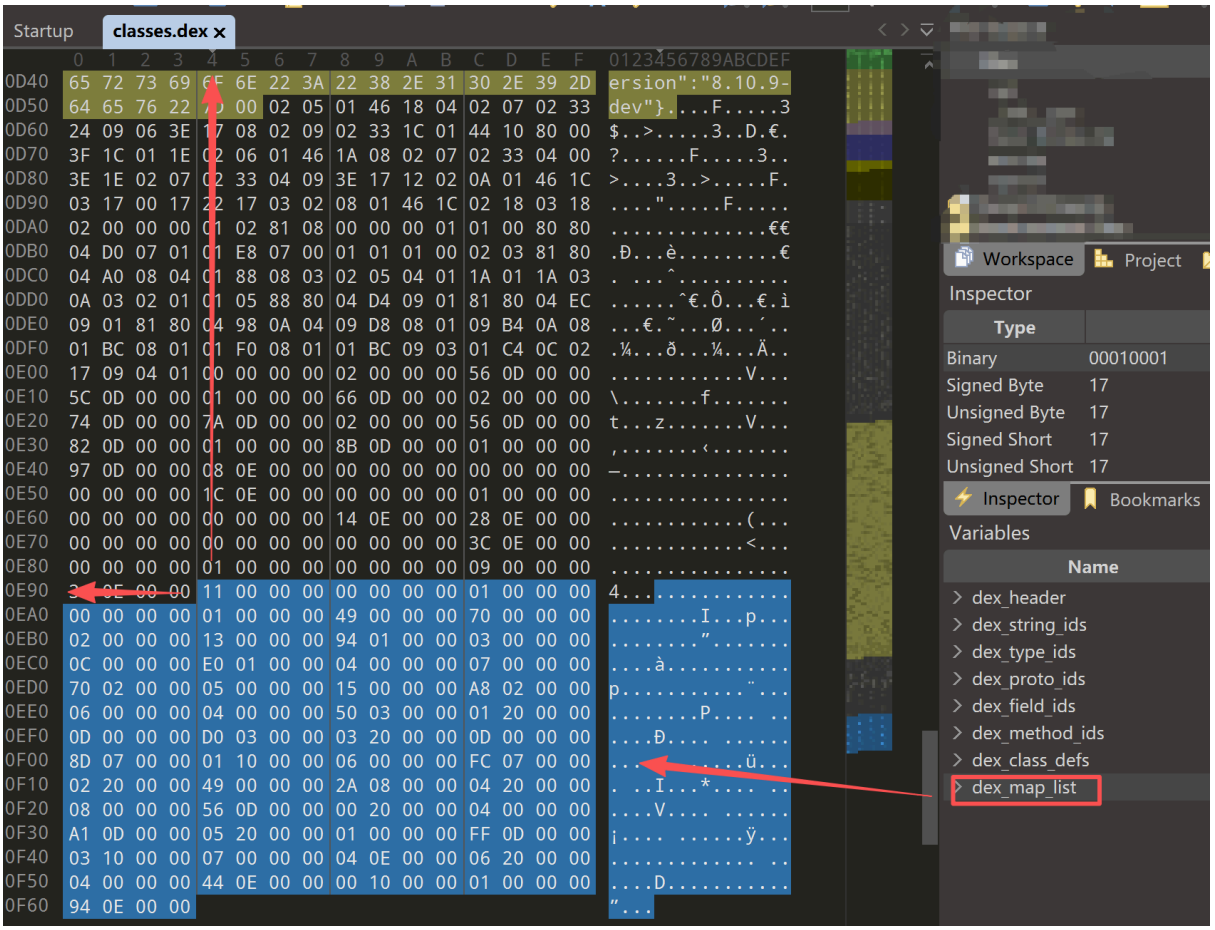
一般都是为0.

`u4 mapOff;`

这里的off指的是偏移



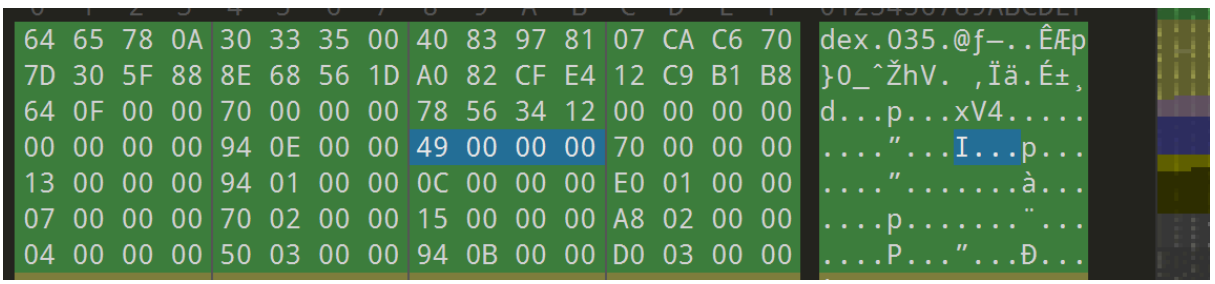
指向DexMapList结构体偏移，对应位置在0x0E94，从这里开始就是它指向的数据



其实就是内存的最后一部分，内存映射

```
u4 stringIdsSize;
```

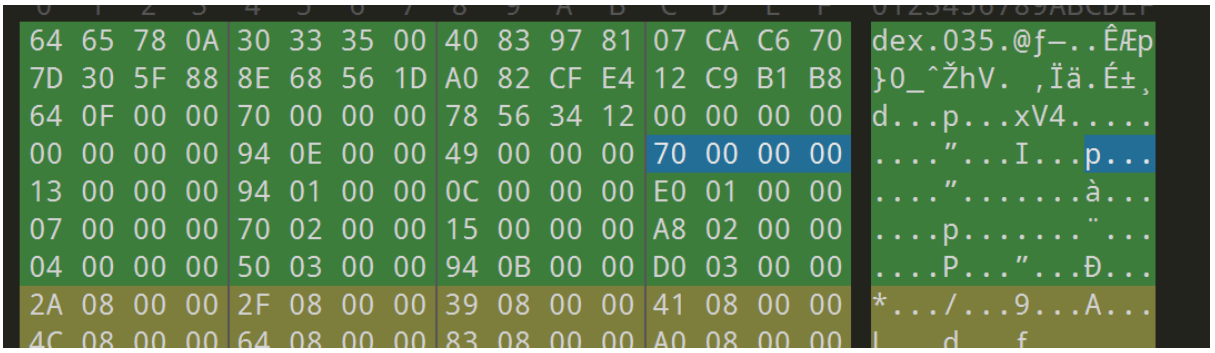
DexStringId个数



相当于有73个字符串，为MUTF-8编码

`u4 stringIdsOff;`

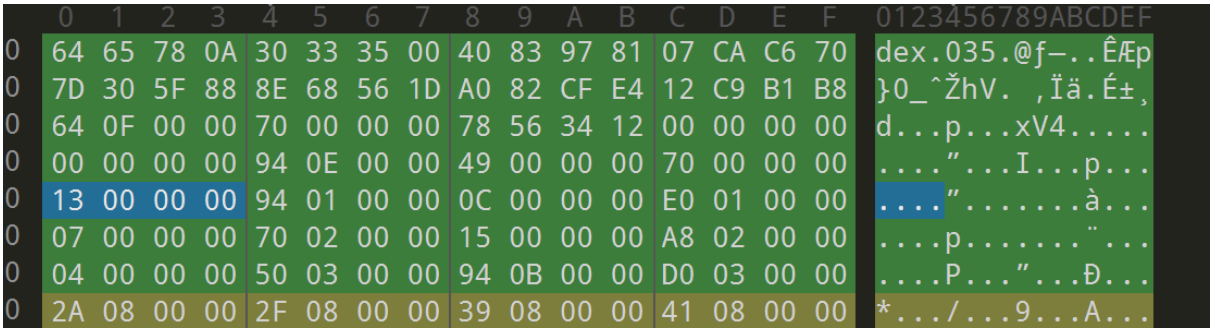
DexStringId的偏移，也代表在0X70位置DexHeader结束.



对应开始位置在0x70，**这里记录的是指针**。有172个字符串，采用MUTF-8编码

`u4 typeIdsSize;`

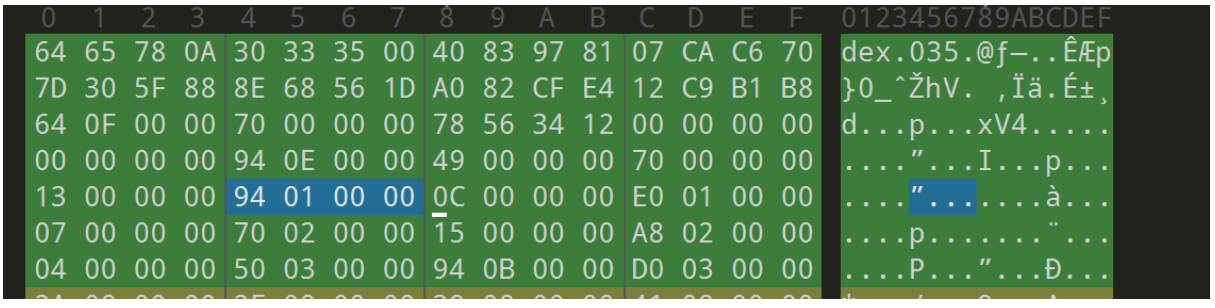
DexTypeId个数



这里对应的是19个类型

`u4 typeIdsOff;`

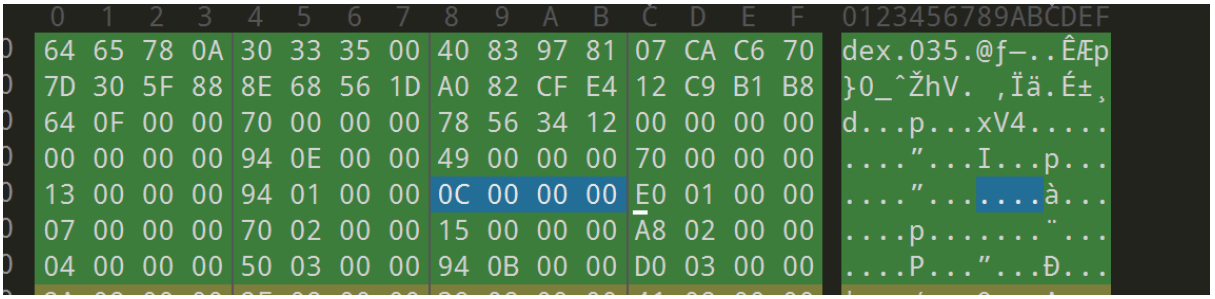
DexTypeId的偏移



对应位置在0x0194，我们进入DexTypeId进行查看

`u4 protoIdsSize;`

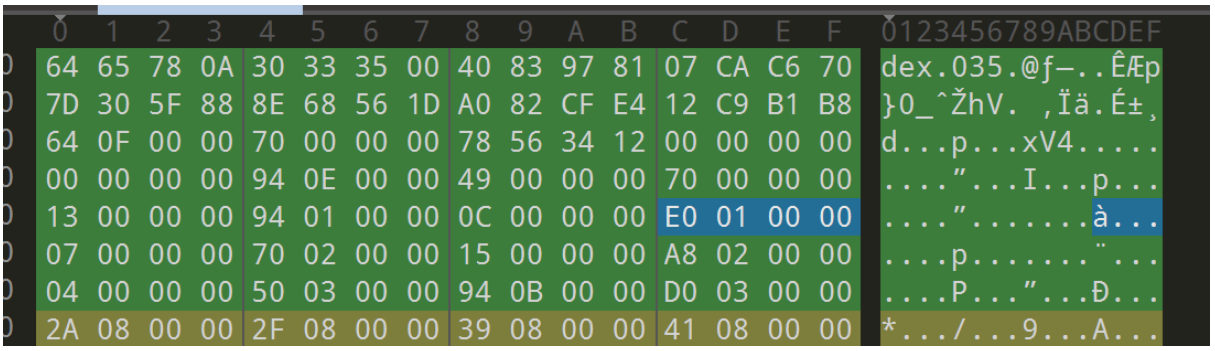
DexProtoId个数



也就是说接下来里面有12个方法原型。

```
u4 protoIdsOff;
```

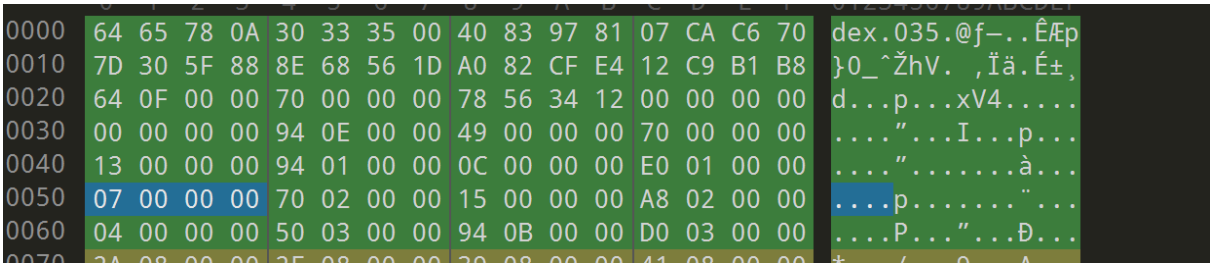
DexProtoId的偏移



指向0x1E0，我们到DexProtoId区域进行查看

```
u4 fieldIdsSize;
```

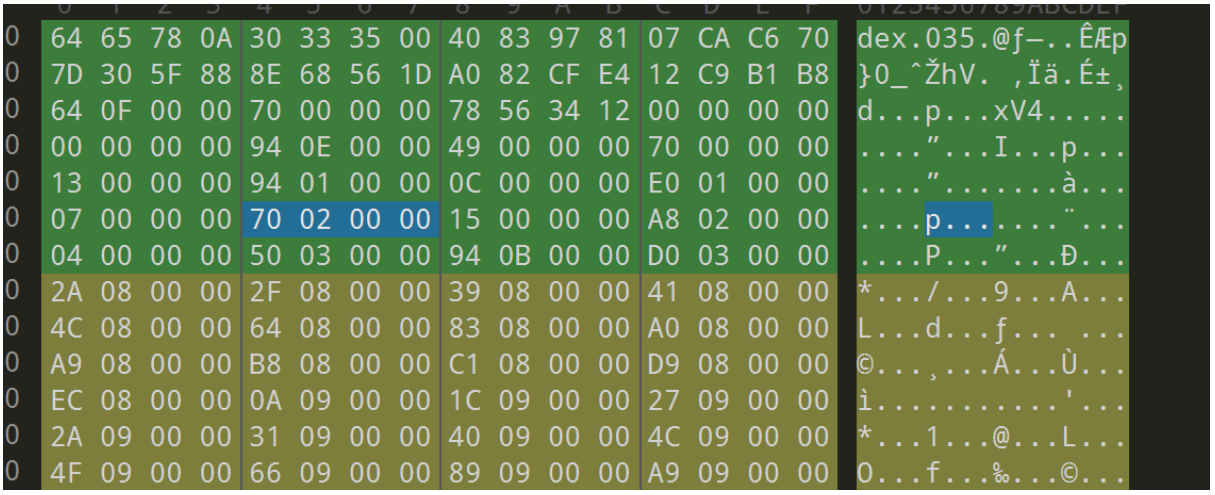
DexFieldId个数



代表有7个DexFieldId

```
u4 fieldIdsOff;
```

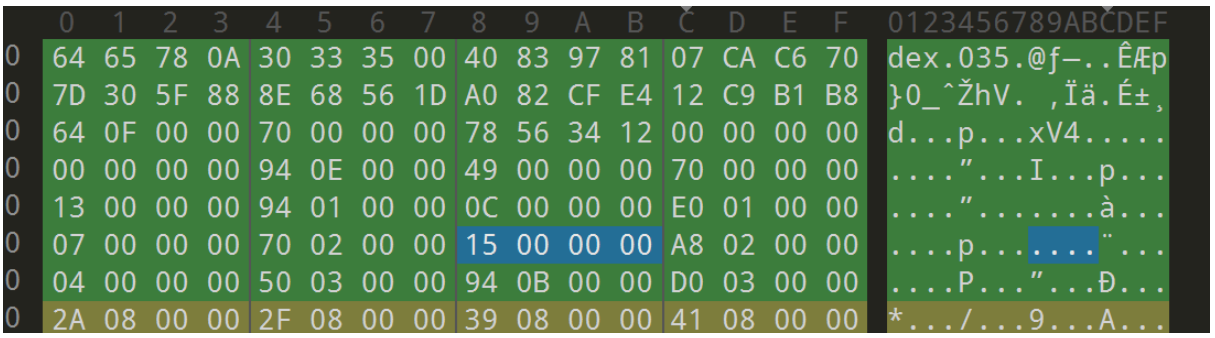
DexFieldId的偏移



偏移为0x270，我们进入DexFieldId进行查看

```
u4 methodIdsSize;
```

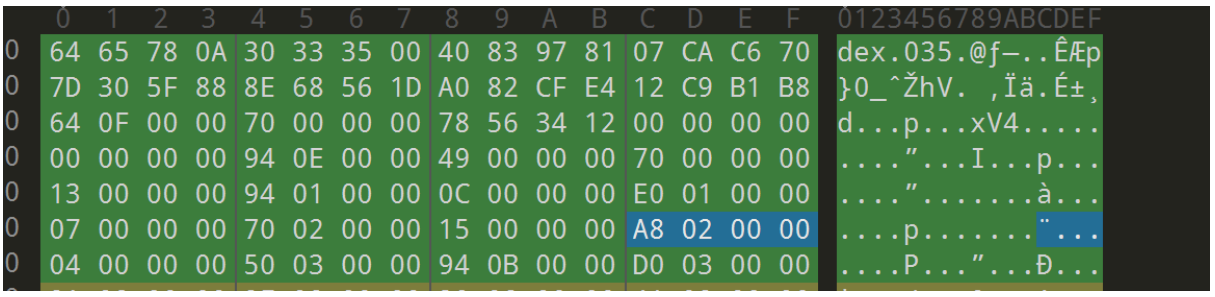
DexMethoId个数



这里有31个 DexMethodId ，它是 DEX 文件中描述一个方法“身份”的结构体，包含它属于哪个类、它的签名（返回类型+参数）以及它的名字，是方法调用的基础引用。

```
u4 methodIdsOff;
```

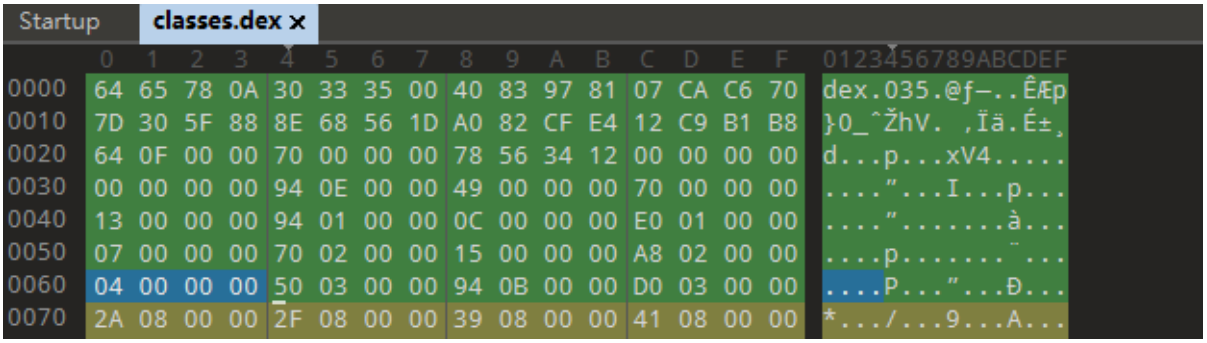
DexMethoId的偏移



这里指向0x2A8，我们进入DexMethoId查看。

```
u4 classDefsSize;
```

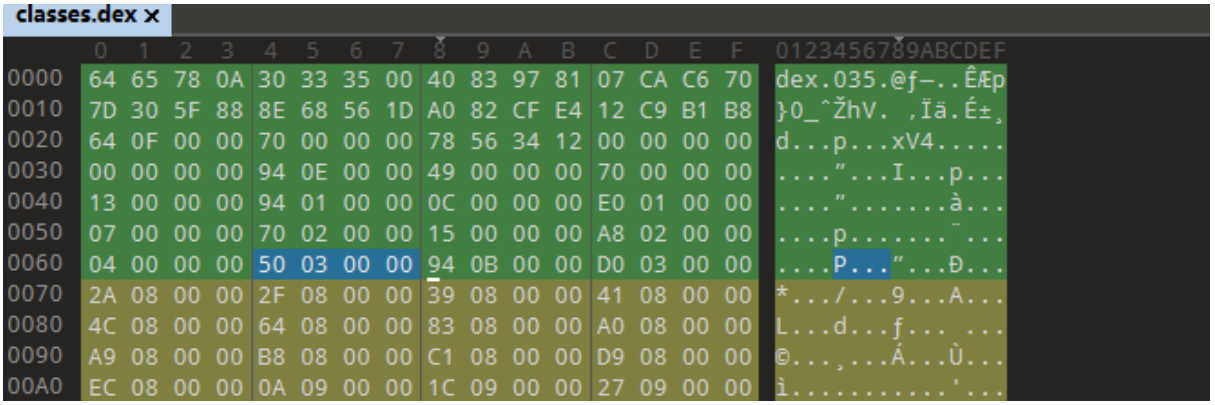
DexClassDef个数



这里有4个DexClassDef

```
u4 classDefsOff;
```

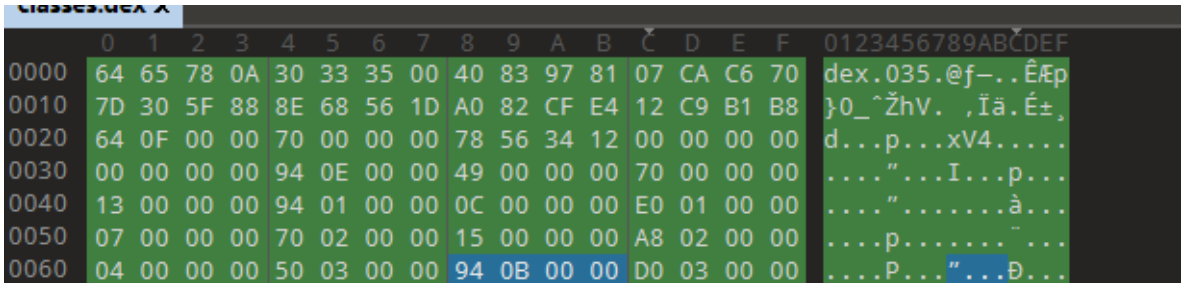
DexClassDef的偏移



这里指向0x350，我们进入DexClassDef查看。

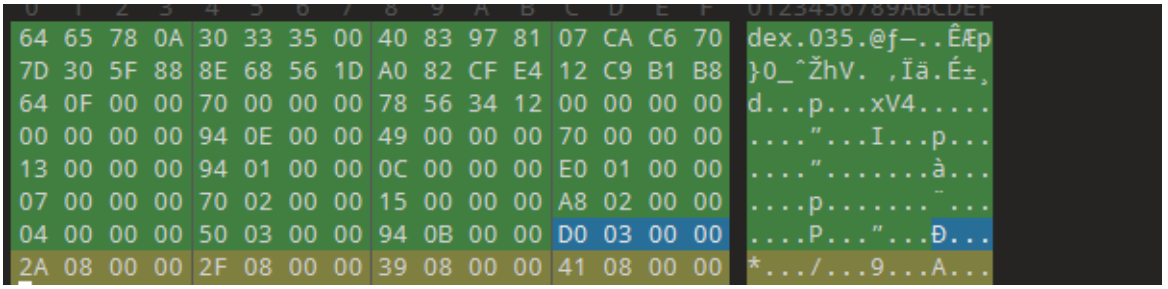
```
u4 dataSize;
```

数据段大小




```
u4 dataOff;
```

数据段的偏移



头的作用：解析器首先读取头，根据头中记录的各个段的偏移量和大小，就能快速定位到文件中的任何数据。

2. 索引表区 (Index Tables / Lookup Tables)

这些表都是**索引表**，它们不存储实际数据，而是存储一个指向实际数据的**偏移量**或**另一个索引**。

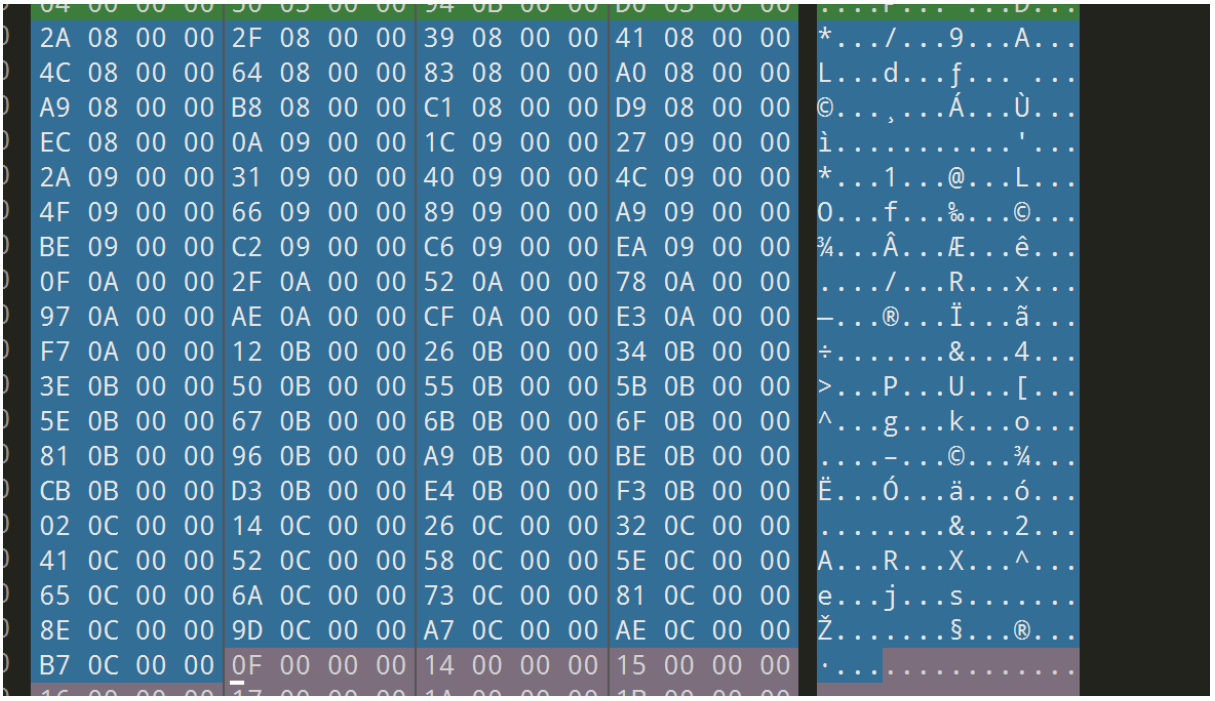
DexStringId(字符串表)

字符串表 (`string_ids`)：每个条目是一个偏移量，指向 `string_data` 区中某个字符串的实际位置。

```
260 /*
261  * Direct-mapped "string_id_item".
262  */
263 struct DexStringId {
264     u4 stringDataOff;      /* file offset to string_data_item */
265 };
```

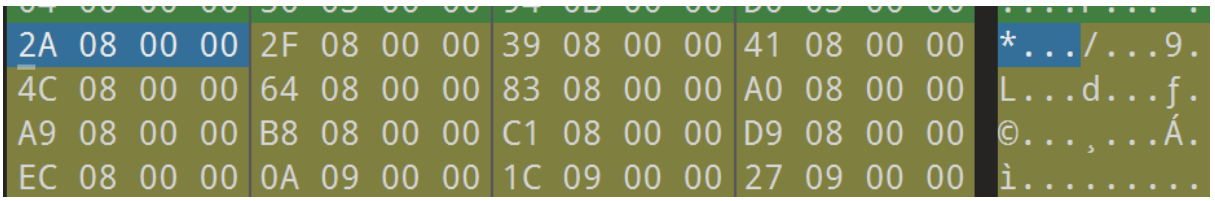
`u4 stringDataOff;`

stringDataOff指向的是真正的数据



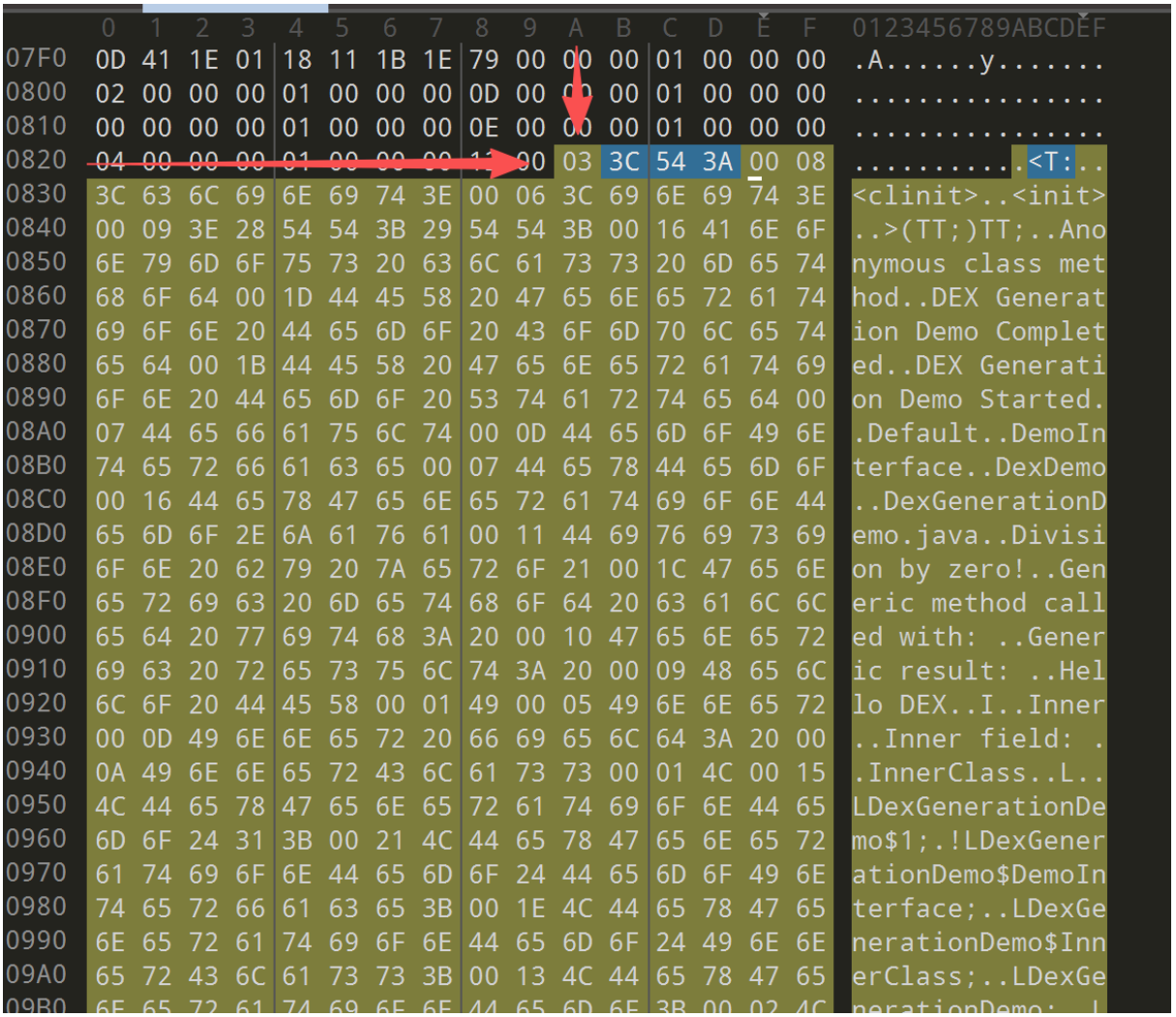
2A	08	00	00	2F	08	00	00	39	08	00	00	41	08	00	00
4C	08	00	00	64	08	00	00	83	08	00	00	A0	08	00	00
A9	08	00	00	B8	08	00	00	C1	08	00	00	D9	08	00	00
EC	08	00	00	0A	09	00	00	1C	09	00	00	27	09	00	00
2A	09	00	00	31	09	00	00	40	09	00	00	4C	09	00	00
4F	09	00	00	66	09	00	00	89	09	00	00	A9	09	00	00
BE	09	00	00	C2	09	00	00	C6	09	00	00	EA	09	00	00
0F	0A	00	00	2F	0A	00	00	52	0A	00	00	78	0A	00	00
97	0A	00	00	AE	0A	00	00	CF	0A	00	00	E3	0A	00	00
F7	0A	00	00	12	0B	00	00	26	0B	00	00	34	0B	00	00
3E	0B	00	00	50	0B	00	00	55	0B	00	00	5B	0B	00	00
5E	0B	00	00	67	0B	00	00	6B	0B	00	00	6F	0B	00	00
81	0B	00	00	96	0B	00	00	A9	0B	00	00	BE	0B	00	00
CB	0B	00	00	D3	0B	00	00	E4	0B	00	00	F3	0B	00	00
02	0C	00	00	14	0C	00	00	26	0C	00	00	32	0C	00	00
41	0C	00	00	52	0C	00	00	58	0C	00	00	5E	0C	00	00
65	0C	00	00	6A	0C	00	00	73	0C	00	00	81	0C	00	00
8E	0C	00	00	9D	0C	00	00	A7	0C	00	00	AE	0C	00	00
B7	0C	00	00	0F	00	00	00	14	00	00	00	15	00	00	00

例如第一个位置是在0x082A，同理后边每4个字节代表一个字符串的地址。



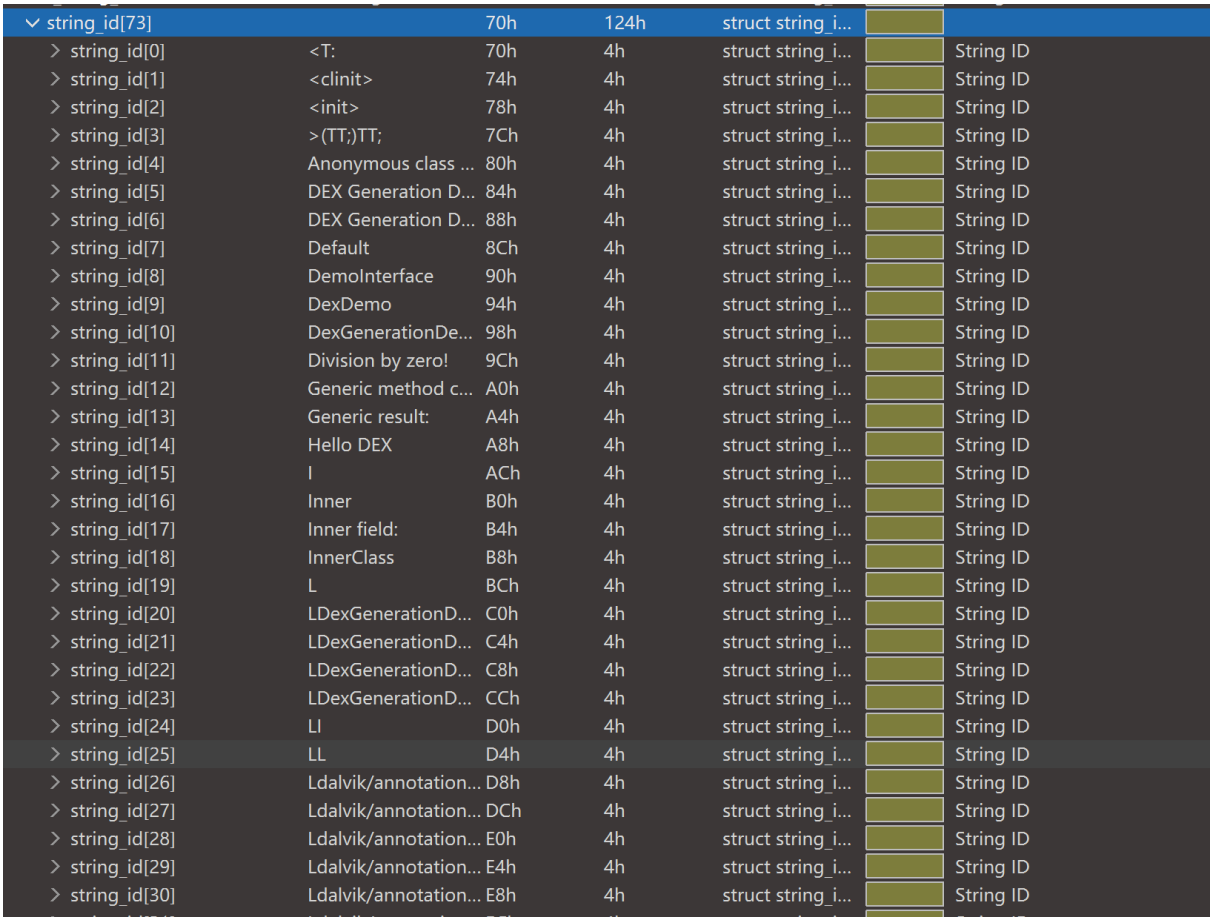
2A	08	00	00	2F	08	00	00	39	08	00	00	41	08	00	00
4C	08	00	00	64	08	00	00	83	08	00	00	A0	08	00	00
A9	08	00	00	B8	08	00	00	C1	08	00	00	D9	08	00	00
EC	08	00	00	0A	09	00	00	1C	09	00	00	27	09	00	00

字符串为MUTF-8编码，开头用uleb128记录字符个数，之后是Utf-8编码字符串。



字符串以00结尾，这中间就是Utf-8编码的内容。

解析172个字符串，从0开始依次排序，得到一张字符串索引表。

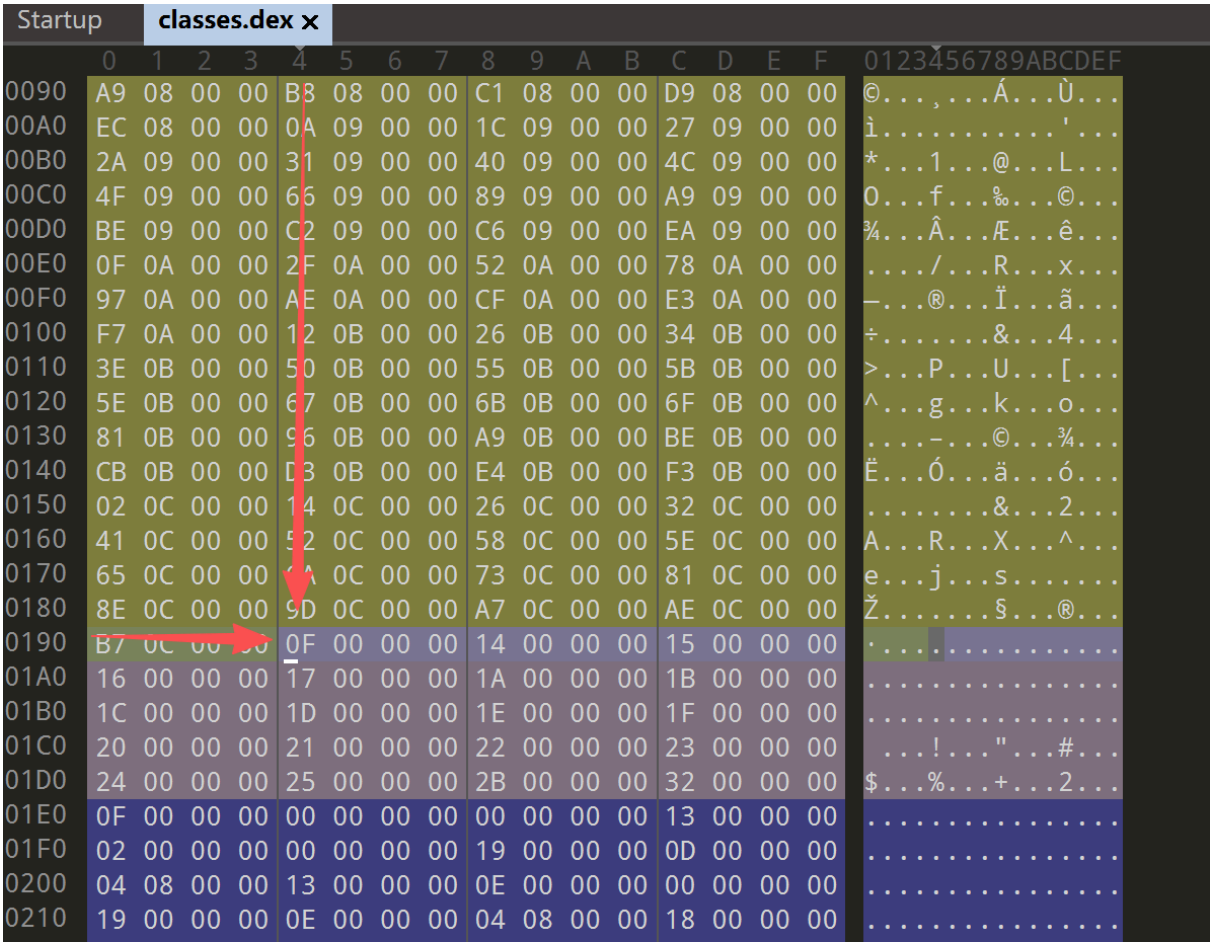


string_id[73]		70h	124h	struct string_i...		
string_id[0]	<T:	70h	4h	struct string_i...		String ID
string_id[1]	<clinit>	74h	4h	struct string_i...		String ID
string_id[2]	<init>	78h	4h	struct string_i...		String ID
string_id[3]	>(TT;)TT;	7Ch	4h	struct string_i...		String ID
string_id[4]	Anonymous class ...	80h	4h	struct string_i...		String ID
string_id[5]	DEX Generation D...	84h	4h	struct string_i...		String ID
string_id[6]	DEX Generation D...	88h	4h	struct string_i...		String ID
string_id[7]	Default	8Ch	4h	struct string_i...		String ID
string_id[8]	DemoInterface	90h	4h	struct string_i...		String ID
string_id[9]	DexDemo	94h	4h	struct string_i...		String ID
string_id[10]	DexGenerationDe...	98h	4h	struct string_i...		String ID
string_id[11]	Division by zero!	9Ch	4h	struct string_i...		String ID
string_id[12]	Generic method c...	A0h	4h	struct string_i...		String ID
string_id[13]	Generic result:	A4h	4h	struct string_i...		String ID
string_id[14]	Hello DEX	A8h	4h	struct string_i...		String ID
string_id[15]	I	ACh	4h	struct string_i...		String ID
string_id[16]	Inner	B0h	4h	struct string_i...		String ID
string_id[17]	Inner field:	B4h	4h	struct string_i...		String ID
string_id[18]	InnerClass	B8h	4h	struct string_i...		String ID
string_id[19]	L	BCh	4h	struct string_i...		String ID
string_id[20]	LDexGenerationD...	C0h	4h	struct string_i...		String ID
string_id[21]	LDexGenerationD...	C4h	4h	struct string_i...		String ID
string_id[22]	LDexGenerationD...	C8h	4h	struct string_i...		String ID
string_id[23]	LDexGenerationD...	CCh	4h	struct string_i...		String ID
string_id[24]	LI	D0h	4h	struct string_i...		String ID
string_id[25]	LL	D4h	4h	struct string_i...		String ID
string_id[26]	Ldalvik/annotation...	D8h	4h	struct string_i...		String ID
string_id[27]	Ldalvik/annotation...	DCh	4h	struct string_i...		String ID
string_id[28]	Ldalvik/annotation...	E0h	4h	struct string_i...		String ID
string_id[29]	Ldalvik/annotation...	E4h	4h	struct string_i...		String ID
string_id[30]	Ldalvik/annotation...	E8h	4h	struct string_i...		String ID

DexTypeId(类型表)

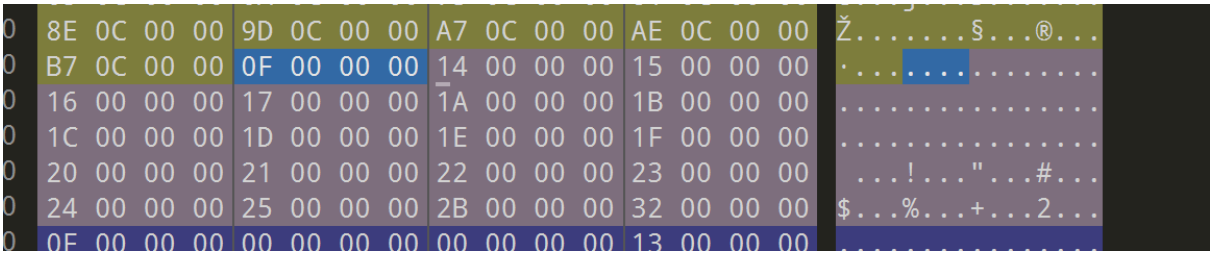
类型表 (`type_ids`)：每个条目是一个 `string_id` 索引，这个字符串就是类的描述符（如 `Ljava/lang/String;`）。

```
267 /*
268  * Direct-mapped "type_id_item".
269  */
270 struct DexTypeId {
271     u4  descriptorIdx;      /* index into stringIds list for type descriptor */
272 };
```

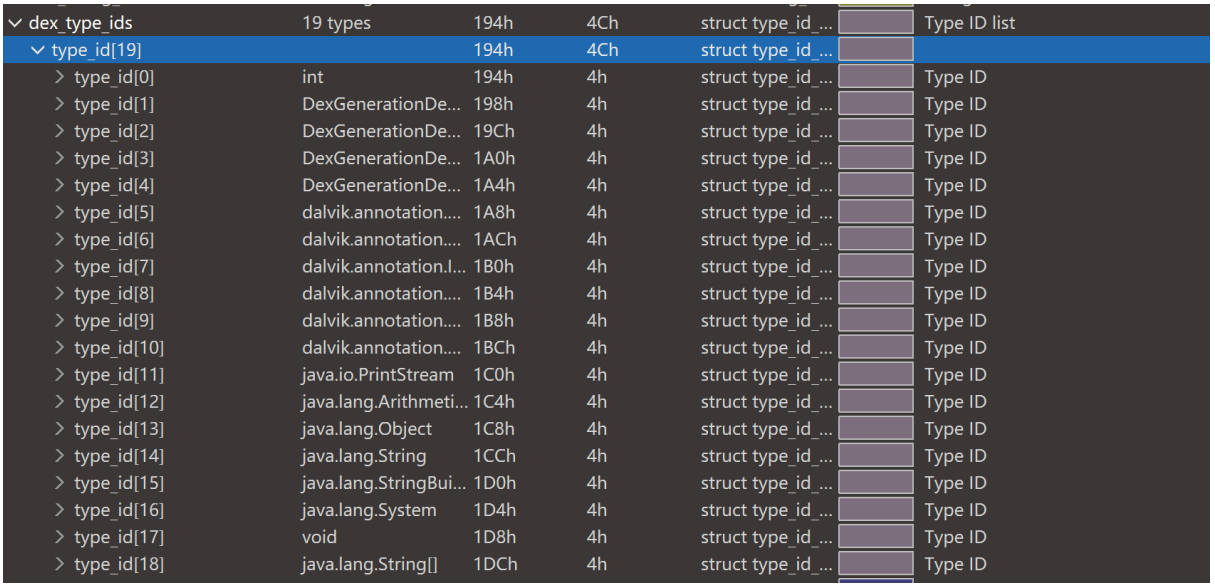



```
u4 descriptorIdx;
```

指向DexStringID列表的索引



代表索引值为F的字符串，也就代表处理后可以得出19个类型，从0开始依次排序，从而得到一张类型索引表。



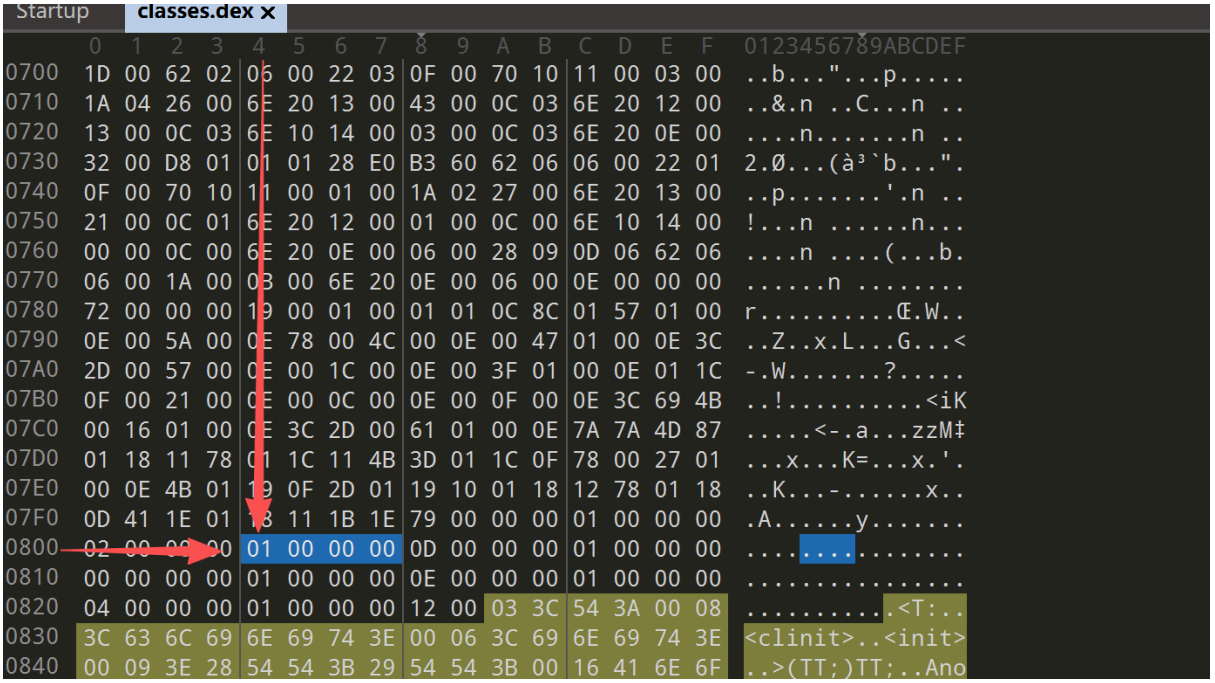
dex_type_ids	19 types	194h	4Ch	struct type_id_...		Type ID list
type_id[19]		194h	4Ch	struct type_id_...		
type_id[0]	int	194h	4h	struct type_id_...		Type ID
type_id[1]	DexGenerationDe...	198h	4h	struct type_id_...		Type ID
type_id[2]	DexGenerationDe...	19Ch	4h	struct type_id_...		Type ID
type_id[3]	DexGenerationDe...	1A0h	4h	struct type_id_...		Type ID
type_id[4]	DexGenerationDe...	1A4h	4h	struct type_id_...		Type ID
type_id[5]	dalvik.annotation....	1A8h	4h	struct type_id_...		Type ID
type_id[6]	dalvik.annotation....	1ACh	4h	struct type_id_...		Type ID
type_id[7]	dalvik.annotation.I...	1B0h	4h	struct type_id_...		Type ID
type_id[8]	dalvik.annotation....	1B4h	4h	struct type_id_...		Type ID
type_id[9]	dalvik.annotation....	1B8h	4h	struct type_id_...		Type ID
type_id[10]	dalvik.annotation....	1BCh	4h	struct type_id_...		Type ID
type_id[11]	java.io.PrintStream	1C0h	4h	struct type_id_...		Type ID
type_id[12]	java.lang.Arithmeti...	1C4h	4h	struct type_id_...		Type ID
type_id[13]	java.lang.Object	1C8h	4h	struct type_id_...		Type ID
type_id[14]	java.lang.String	1CCh	4h	struct type_id_...		Type ID
type_id[15]	java.lang.StringBui...	1D0h	4h	struct type_id_...		Type ID
type_id[16]	java.lang.System	1D4h	4h	struct type_id_...		Type ID
type_id[17]	void	1D8h	4h	struct type_id_...		Type ID
type_id[18]	java.lang.String[]	1DCh	4h	struct type_id_...		Type ID

DexTypeList

```
322  /*
323   * Direct-mapped "type_list".
324   */
325  struct DexTypeList {
326      u4  size;               /* #of entries in list */
327      DexTypeItem list[1];    /* entries */
328  };
```

u4 size;

DexTypeItem的个数



指明由DexProtoId分析来的参数有多少个，这里明显为1个参数，也就是一个DexTypeItem 结构体。

DexTypeItem list[1];

DexTypeItem结构，为直接跟在后面的可以理解为DexTypeItem list[size];

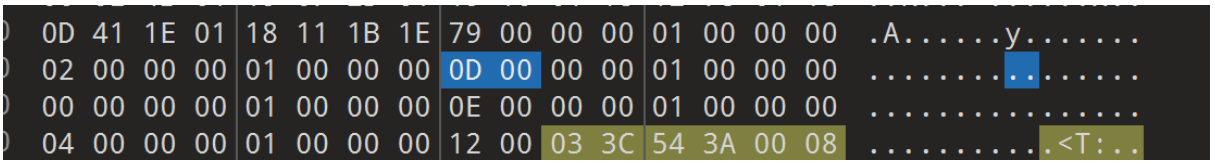
里面放的是真正的函数类型

DexTypeItem

```
315  /*
316   * Direct-mapped "type_item".
317   */
318  struct DexTypeItem {
319      u2  typeIdx;          /* index into typeIds */
320  };
```

u2 typeIdx;

指向DexTypeId列表的索引

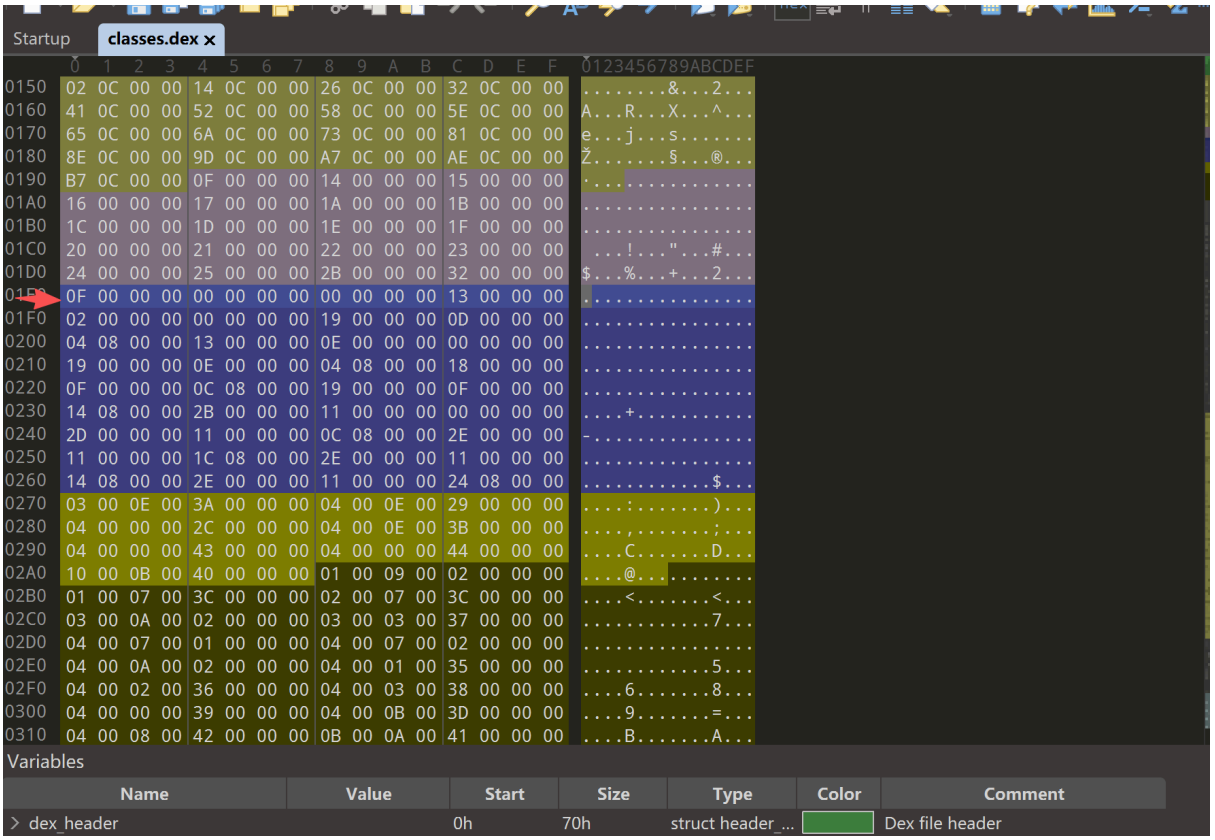


结构体占两个字节，TypeId中索引为D

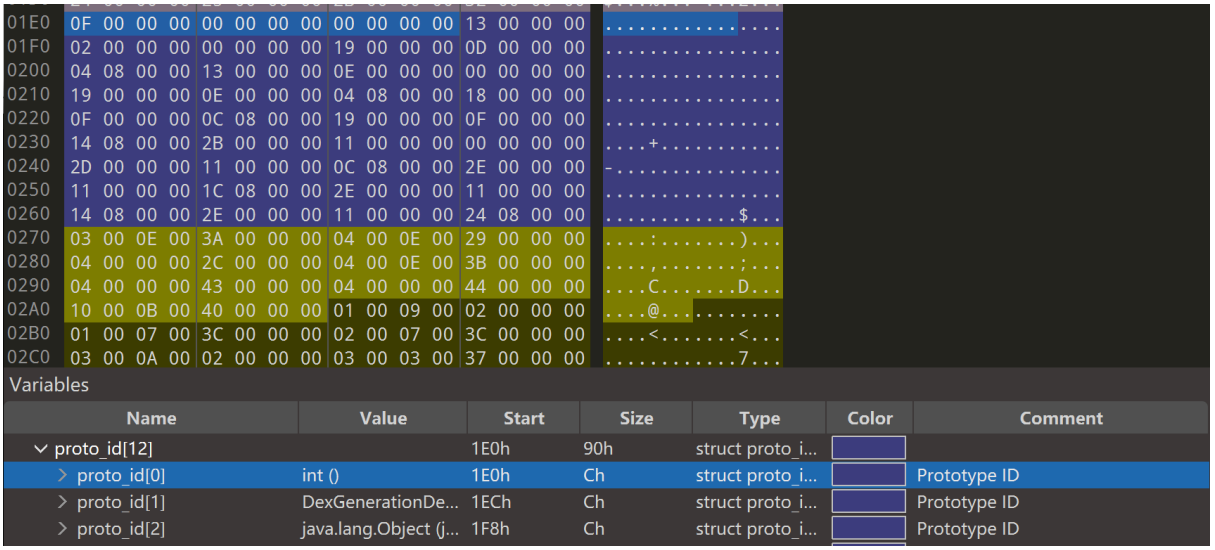
DexProtoId(原型表)

原型表 (proto_ids)：每个条目包含一个返回类型索引和一个指向参数列表的偏移量，共同构成方法的签名（如 (II)I）。

```
292 /*
293  * Direct-mapped "proto_id_item".
294  */
295 struct DexProtoId {
296     u4  shortyIdx;         /* index into stringIds for shorty descriptor */
297     u4  returnTypeIdx;     /* index into typeIds list for return type */
298     u4  parametersOff;     /* file offset to type_list for parameter types */
299 };
```



由上得每个ProtoId占12个字节



`u4 shortyIdx;`

指向DexStringId列表的索引，这里指向0F

我们发现是“I”，代表我们方法原型里有一个int类型

`u4 returnTypeIdx;`

指向DexTypeId列表的索引

要表明一个方法，需要知道方法返回值类型

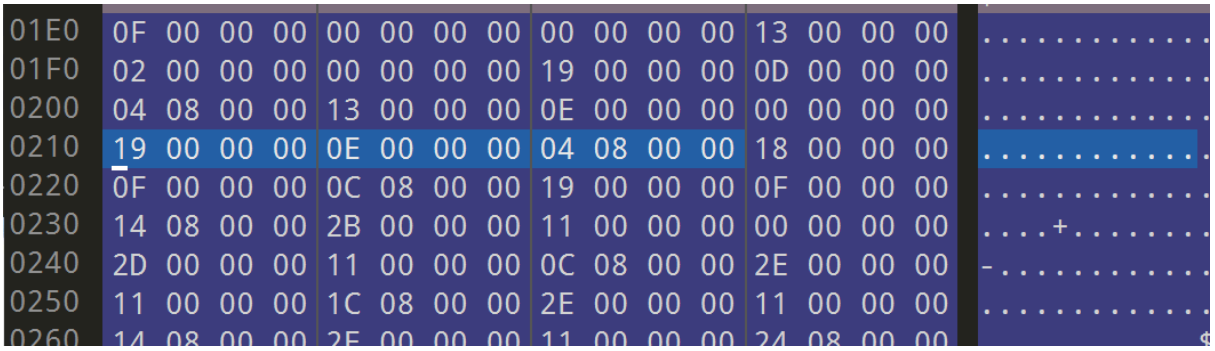
这里指向00 00 00 00，也就是说指向我们DexTypeId里面的0，说明返回值是int

`u4 parametersOff;`

指向DexTypeList的位置偏移

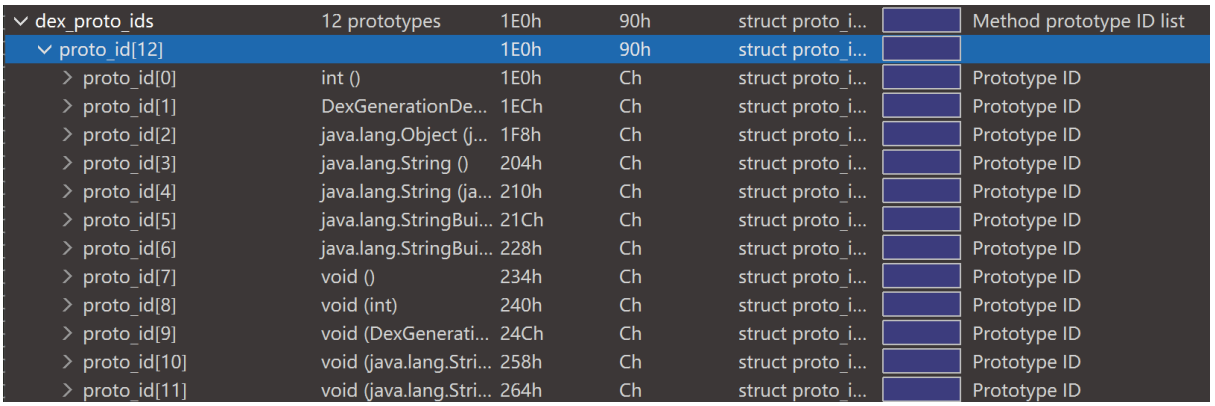
表明方法自然也需要参数类型，而参数可能为多个，所以需要一个参数列表结构体去表示，DexTypeList。

这里为000000000，代表方法没有参数，不指向任何 `type_list`。



我们换ProtoId[4]继续进行演示，这里指向0804的位置。我们进入DexTypeList继续查看。

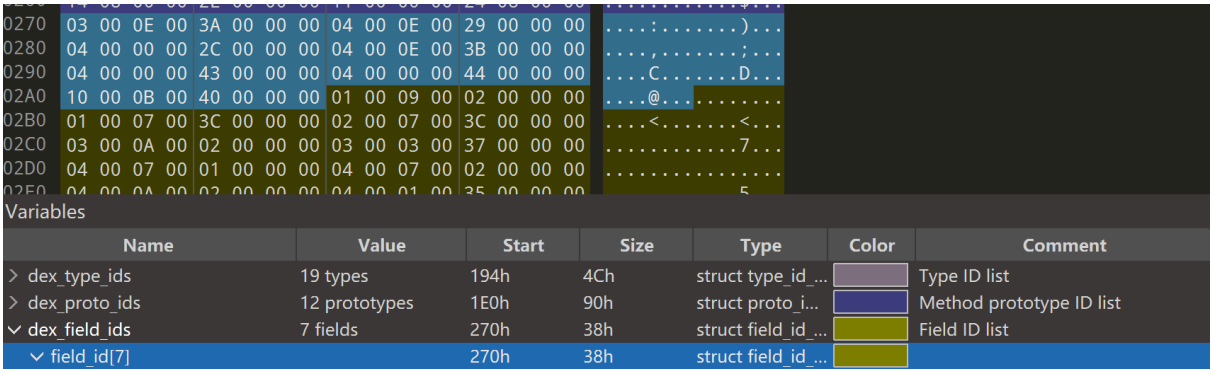
全部解析完后进行组和后我们会得到所有方法原型列表



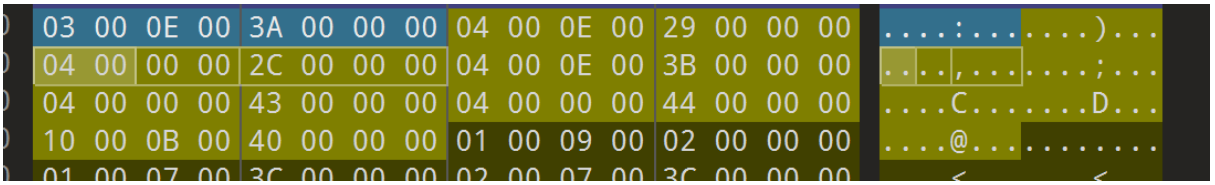
DexFieldId(字段表)

字段表 (field_ids)：每个条目由**所属类索引**、**字段类型索引**和**字段名索引**组成，共同唯一确定一个字段（如 `MyClass.myField:I`）。

```
274  /*
275   * Direct-mapped "field_id_item".
276   */
277  struct DexFieldId {
278      u2  classIdx;         /* index into typeIds list for defining class */
279      u2  typeIdx;          /* index into typeIds for field type */
280      u4  nameIdx;           /* index into stringIds for field name */
281  };
```

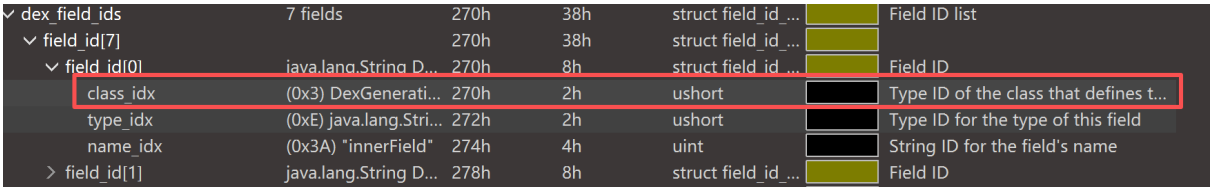


由上得每个DexFieldId(占8个字节




```
u2 classIdx;
```

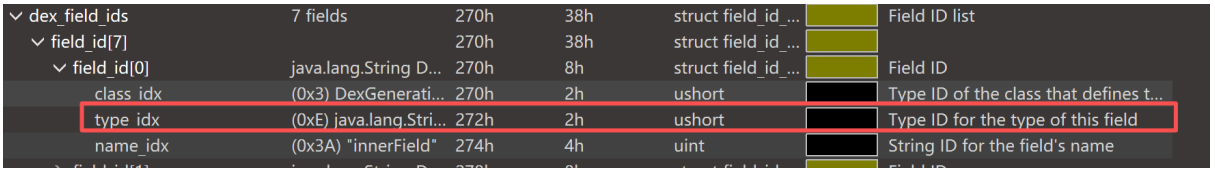
类的类型，指向DexTypeId列表的索引



要描述一个类的属性，自然也要有类的相关索引，这里指向3

```
u2 typeIdx;
```

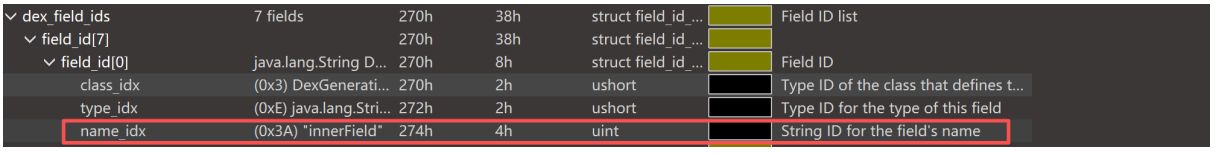
字段类型，指向DexTypeId列表的索引



这里指向0E

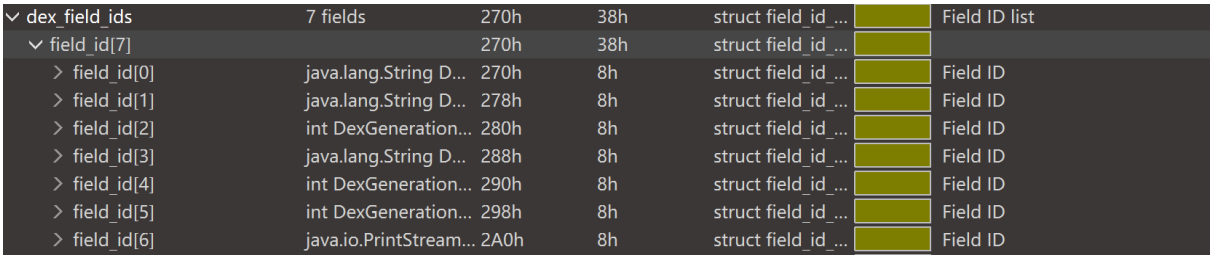
```
u4 nameIdx;
```

字段名，指向DexStringId列表的索引



处理过就可以显现，也就是该字段ID的名字。

全部解析完后进行组和后我们会得到所有字段列表

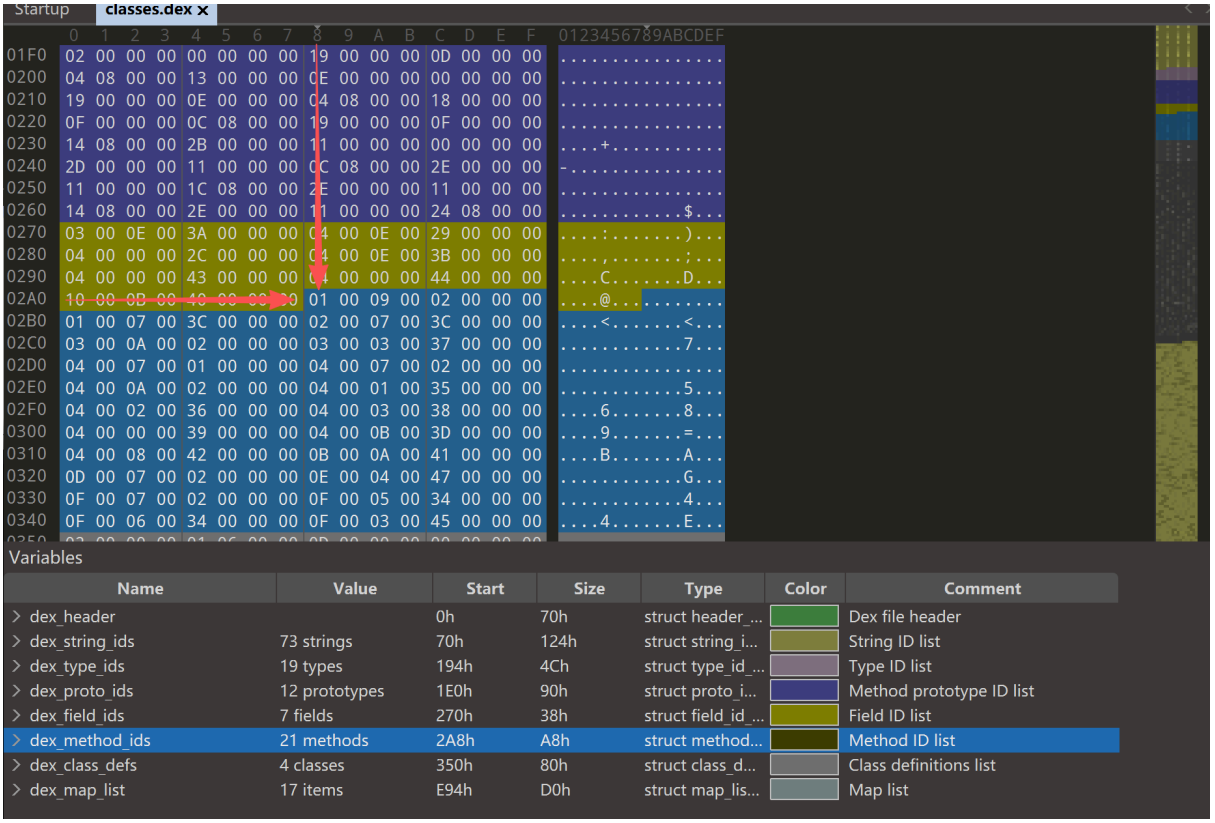


DexMethodId(方法表)

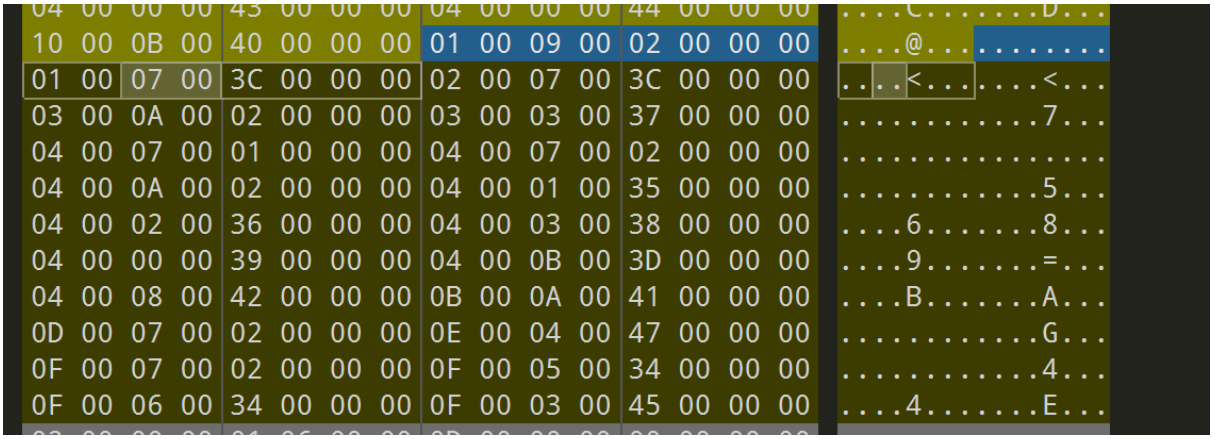
方法表 (`method_ids`)：每个条目包含：所属类的 `type_id`、方法原型的 `proto_id`、方法名的 `string_id`。

```
283  /*
284   * Direct-mapped "method_id_item".
285   */
286  struct DexMethodId {
287      u2  classIdx;           /* index into typeIds list for defining class */
288      u2  protoIdx;           /* index into protoIds for method prototype */
289      u4  nameIdx;            /* index into stringIds for method name */
290  };
```

这种设计使得一个字符串（如 `"toString"`）或一个类型（如 `Ljava/lang/Object;`）在整个文件中**只存储一次**，所有引用它的地方都使用索引，极大地节省了空间。

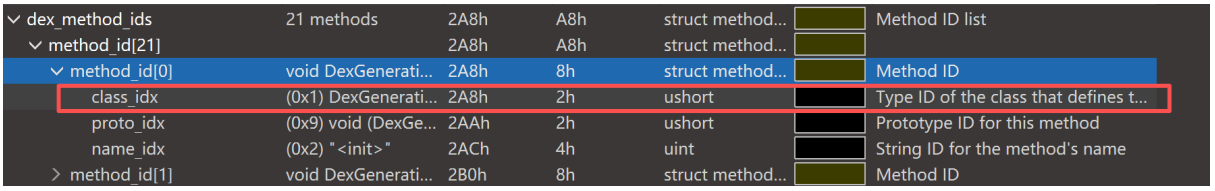


由上得每个DexMethodId占8个字节



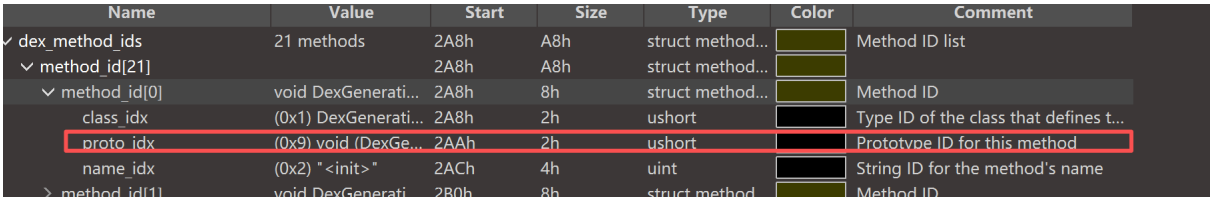
u2 classIdx;

类的类型，指向DexTypeId列表的索引



u2 protoIdx;

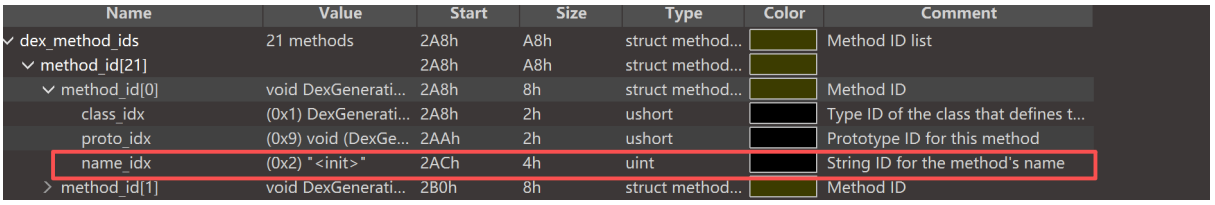
声明类型，指向DexProtoId列表的索引



Name	Value	Start	Size	Type	Color	Comment
dex_method_ids	21 methods	2A8h	A8h	struct method...		Method ID list
method_id[21]		2A8h	A8h	struct method...		
method_id[0]	void DexGenerati...	2A8h	8h	struct method...		Method ID
class_idx	(0x1) DexGenerati...	2A8h	2h	ushort		Type ID of the class that defines t...
proto_idx	(0x9) void (DexGe...	2AAh	2h	ushort		Prototype ID for this method
name_idx	(0x2) "<init>"	2ACh	4h	uint		String ID for the method's name
method_id[1]	void DexGenerati...	2B0h	8h	struct method...		Method ID

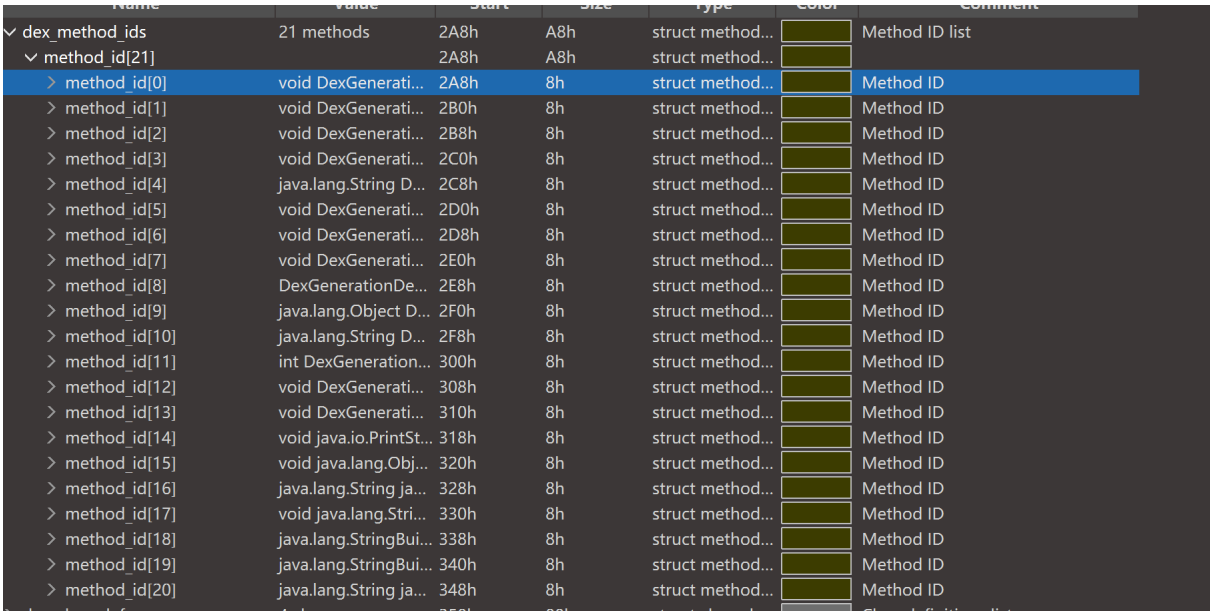
u4 nameIdx;

方法名，指向DexStringId列表的索引



Name	Value	Start	Size	Type	Color	Comment
dex_method_ids	21 methods	2A8h	A8h	struct method...		Method ID list
method_id[21]		2A8h	A8h	struct method...		
method_id[0]	void DexGenerati...	2A8h	8h	struct method...		Method ID
class_idx	(0x1) DexGenerati...	2A8h	2h	ushort		Type ID of the class that defines t...
proto_idx	(0x9) void (DexGe...	2AAh	2h	ushort		Prototype ID for this method
name_idx	(0x2) "<init>"	2ACh	4h	uint		String ID for the method's name
method_id[1]	void DexGenerati...	2B0h	8h	struct method...		Method ID

全部解析完后进行组和后我们会得到所有方法表



dex_method_ids	21 methods	2A8h	A8h	struct method...		Method ID list
method_id[21]		2A8h	A8h	struct method...		
method_id[0]	void DexGenerati...	2A8h	8h	struct method...		Method ID
method_id[1]	void DexGenerati...	2B0h	8h	struct method...		Method ID
method_id[2]	void DexGenerati...	2B8h	8h	struct method...		Method ID
method_id[3]	void DexGenerati...	2C0h	8h	struct method...		Method ID
method_id[4]	java.lang.String D...	2C8h	8h	struct method...		Method ID
method_id[5]	void DexGenerati...	2D0h	8h	struct method...		Method ID
method_id[6]	void DexGenerati...	2D8h	8h	struct method...		Method ID
method_id[7]	void DexGenerati...	2E0h	8h	struct method...		Method ID
method_id[8]	DexGenerationDe...	2E8h	8h	struct method...		Method ID
method_id[9]	java.lang.Object D...	2F0h	8h	struct method...		Method ID
method_id[10]	java.lang.String D...	2F8h	8h	struct method...		Method ID
method_id[11]	int DexGeneration...	300h	8h	struct method...		Method ID
method_id[12]	void DexGenerati...	308h	8h	struct method...		Method ID
method_id[13]	void DexGenerati...	310h	8h	struct method...		Method ID
method_id[14]	void java.io.PrintSt...	318h	8h	struct method...		Method ID
method_id[15]	void java.lang.Obj...	320h	8h	struct method...		Method ID
method_id[16]	java.lang.String ja...	328h	8h	struct method...		Method ID
method_id[17]	void java.lang.Stri...	330h	8h	struct method...		Method ID
method_id[18]	java.lang.StringBui...	338h	8h	struct method...		Method ID
method_id[19]	java.lang.StringBui...	340h	8h	struct method...		Method ID
method_id[20]	java.lang.String ja...	348h	8h	struct method...		Method ID

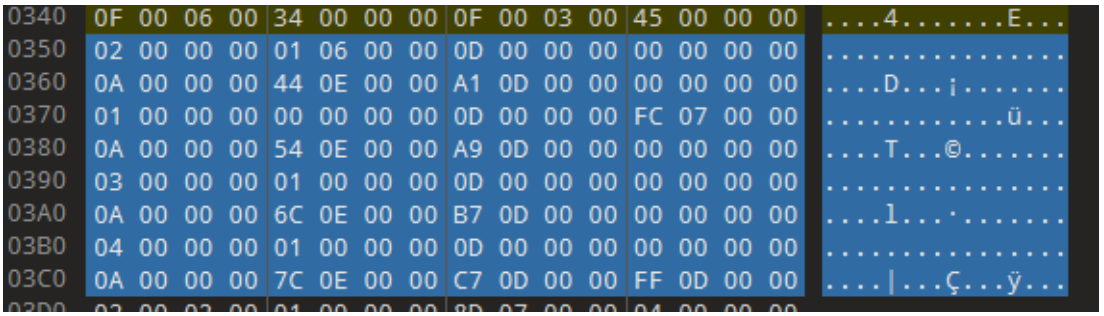
3. 数据区 (Data Section)

存储了实际的代码、数据和其他内容。为了节省空间，此区域大量使用 `uleb128` / `sleb128` 变长编码。结构不是固定大小的。

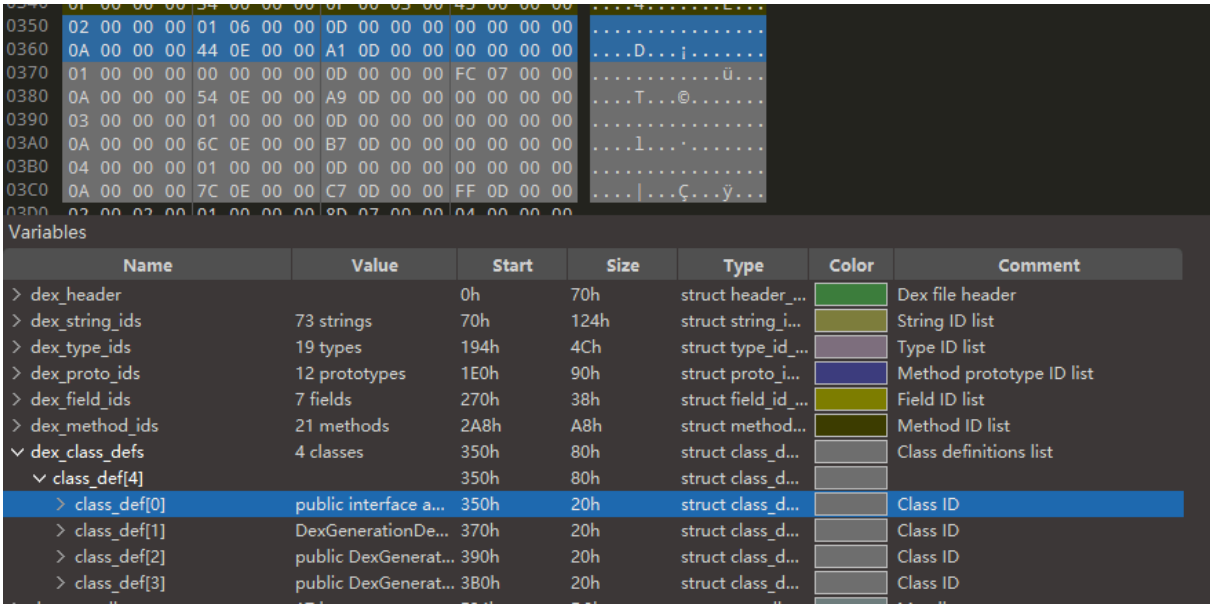
DexClassDef

是 DEX 里**“一个类/接口/枚举”的唯一入口描述**，它只存“元数据 + 偏移”，**不含字节码、字段列表本身**，真正的运行时数据通过偏移再去数据区拿。

```
301 /*
302  * Direct-mapped "class_def_item".
303  */
304 struct DexClassDef {
305     u4  classIdx;           /* index into typeIds for this class */
306     u4  accessFlags;
307     u4  superclassIdx;      /* index into typeIds for superclass */
308     u4  interfacesOff;      /* file offset to DexTypeList */
309     u4  sourceFileIdx;      /* index into stringIds for source file name */
310     u4  annotationsOff;     /* file offset to annotations_directory_item */
311     u4  classDataOff;       /* file offset to class_data_item */
312     u4  staticValuesOff;    /* file offset to DexEncodedArray */
313 };
```

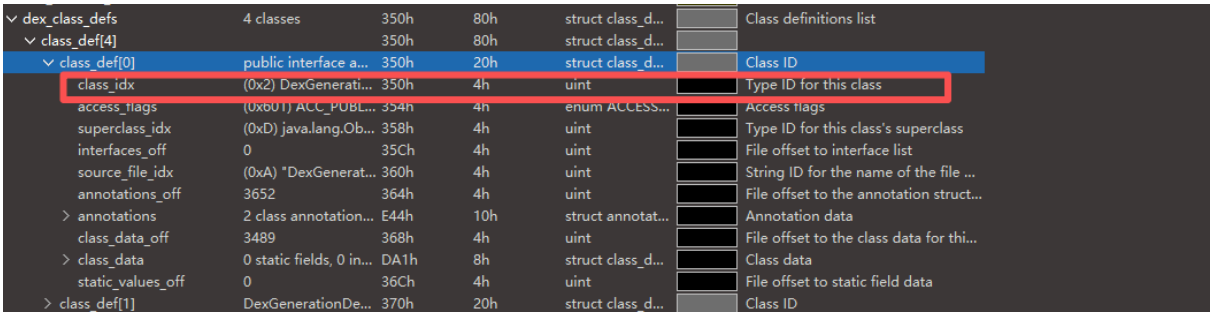


由上得每个DexMethodId占32个字节



`u4  classIdx;`

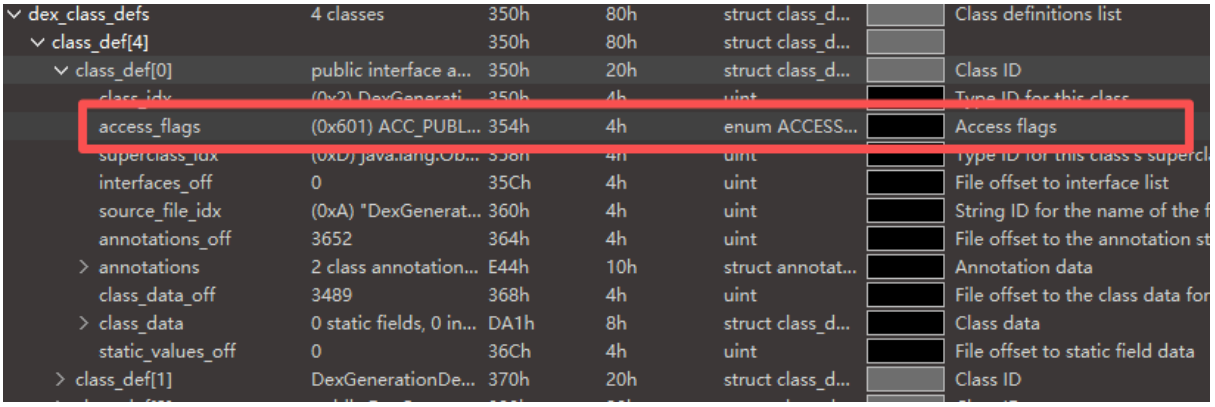
类的类型。这是一个指向 `type_ids` 表的索引。该索引处的类型描述符。



这里指向 `type_ids` 索引为2的类型

`u4 accessFlags;`

类的访问和属性标志。是一个位掩码，定义了类的修饰符（public这类就是修饰符）。



实际就是枚举的常量，这里是0x601

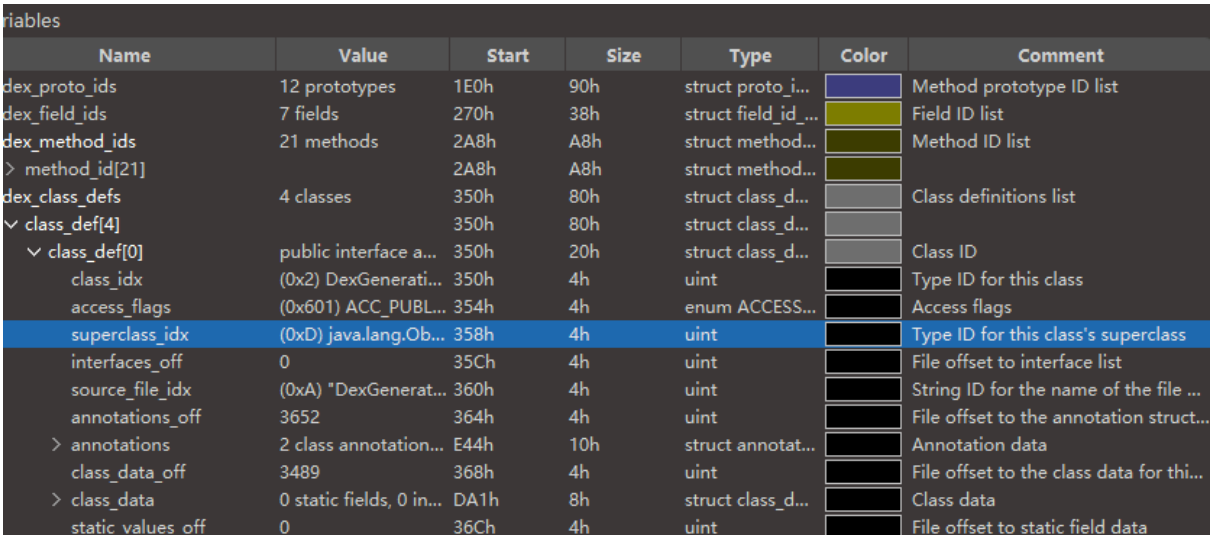
枚举常量如下：

```
96  /*
97   * access flags and masks; the "standard" ones are all <= 0x4000
98   *
99   * Note: There are related declarations in vm/oo/Object.h in the ClassFlags
100   * enum.
101   */
102  enum {
103      ACC_PUBLIC       = 0x00000001,       // class, field, method, ic
104      ACC_PRIVATE      = 0x00000002,       // field, method, ic
105      ACC_PROTECTED    = 0x00000004,       // field, method, ic
106      ACC_STATIC       = 0x00000008,       // field, method, ic
107      ACC_FINAL        = 0x00000010,       // class, field, method, ic
108      ACC_SYNCHRONIZED = 0x00000020,       // method (only allowed on n
```

```
atives)
109     ACC_SUPER       = 0x00000020,       // class (not used in Dalvik)
110     ACC_VOLATILE    = 0x00000040,       // field
111     ACC_BRIDGE      = 0x00000040,       // method (1.5)
112     ACC_TRANSIENT   = 0x00000080,       // field
113     ACC_VARARGS     = 0x00000080,       // method (1.5)
114     ACC_NATIVE      = 0x00000100,       // method
115     ACC_INTERFACE   = 0x00000200,       // class, ic
116     ACC_ABSTRACT    = 0x00000400,       // class, method, ic
117     ACC_STRICT      = 0x00000800,       // method
118     ACC_SYNTHETIC   = 0x00001000,       // field, method, ic
119     ACC_ANNOTATION  = 0x00002000,       // class, ic (1.5)
120     ACC_ENUM        = 0x00004000,       // class, field, ic (1.5)
121     ACC_CONSTRUCTOR  = 0x00010000,      // method (Dalvik only)
122     ACC_DECLARED_SYNCHRONIZED =
123                 0x00020000,      // method (Dalvik only)
124     ACC_CLASS_MASK =
125         (ACC_PUBLIC | ACC_FINAL | ACC_INTERFACE | ACC_ABSTRACT
126             | ACC_SYNTHETIC | ACC_ANNOTATION | ACC_ENUM),
127     ACC_INNER_CLASS_MASK =
128         (ACC_CLASS_MASK | ACC_PRIVATE | ACC_PROTECTED | ACC_STATI
C),
129     ACC_FIELD_MASK =
130         (ACC_PUBLIC | ACC_PRIVATE | ACC_PROTECTED | ACC_STATIC | AC
C_FINAL
131             | ACC_VOLATILE | ACC_TRANSIENT | ACC_SYNTHETIC | ACC_EN
UM),
132     ACC_METHOD_MASK =
133         (ACC_PUBLIC | ACC_PRIVATE | ACC_PROTECTED | ACC_STATIC | AC
C_FINAL
134             | ACC_SYNCHRONIZED | ACC_BRIDGE | ACC_VARARGS | ACC_N
ATIVE
135             | ACC_ABSTRACT | ACC_STRICT | ACC_SYNTHETIC | ACC_CON
STRUCTOR
136             | ACC_DECLARED_SYNCHRONIZED),
137  };
```

`u4 superclassIdx;`

父类的类型。同样是一个指向 `type_ids` 表的索引。对于所有类，这都必须指向一个非final类（`java.lang.Object` 除外）。对于 `java.lang.Object` 本身，这个值为 `NO_INDEX` （0）。

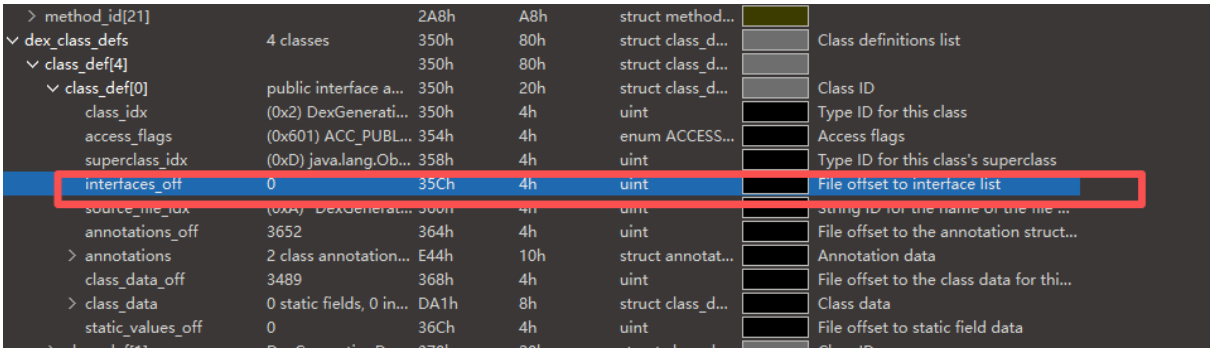


Name	Value	Start	Size	Type	Color	Comment
dex_proto_ids	12 prototypes	1E0h	90h	struct proto_i...		Method prototype ID list
dex_field_ids	7 fields	270h	38h	struct field_id_...		Field ID list
dex_method_ids	21 methods	2A8h	A8h	struct method...		Method ID list
> method_id[21]		2A8h	A8h	struct method...		
dex_class_defs	4 classes	350h	80h	struct class_d...		Class definitions list
v class_def[4]		350h	80h	struct class_d...		
v class_def[0]	public interface a...	350h	20h	struct class_d...		Class ID
class_idx	(0x2) DexGenerati...	350h	4h	uint		Type ID for this class
access_flags	(0x601) ACC_PUBL...	354h	4h	enum ACCESS...		Access flags
superclass_idx	(0xD) java.lang.Ob...	358h	4h	uint		Type ID for this class's superclass
interfaces_off	0	35Ch	4h	uint		File offset to interface list
source_file_idx	(0xA) "DexGenerat...	360h	4h	uint		String ID for the name of the file ...
annotations_off	3652	364h	4h	uint		File offset to the annotation struct...
> annotations	2 class annotation...	E44h	10h	struct annotat...		Annotation data
class_data_off	3489	368h	4h	uint		File offset to the class data for thi...
> class_data	0 static fields, 0 in...	DA1h	8h	struct class_d...		Class data
static_values_off	0	36Ch	4h	uint		File offset to static field data

`Object`是所有类的父类

`u4 interfacesOff;`

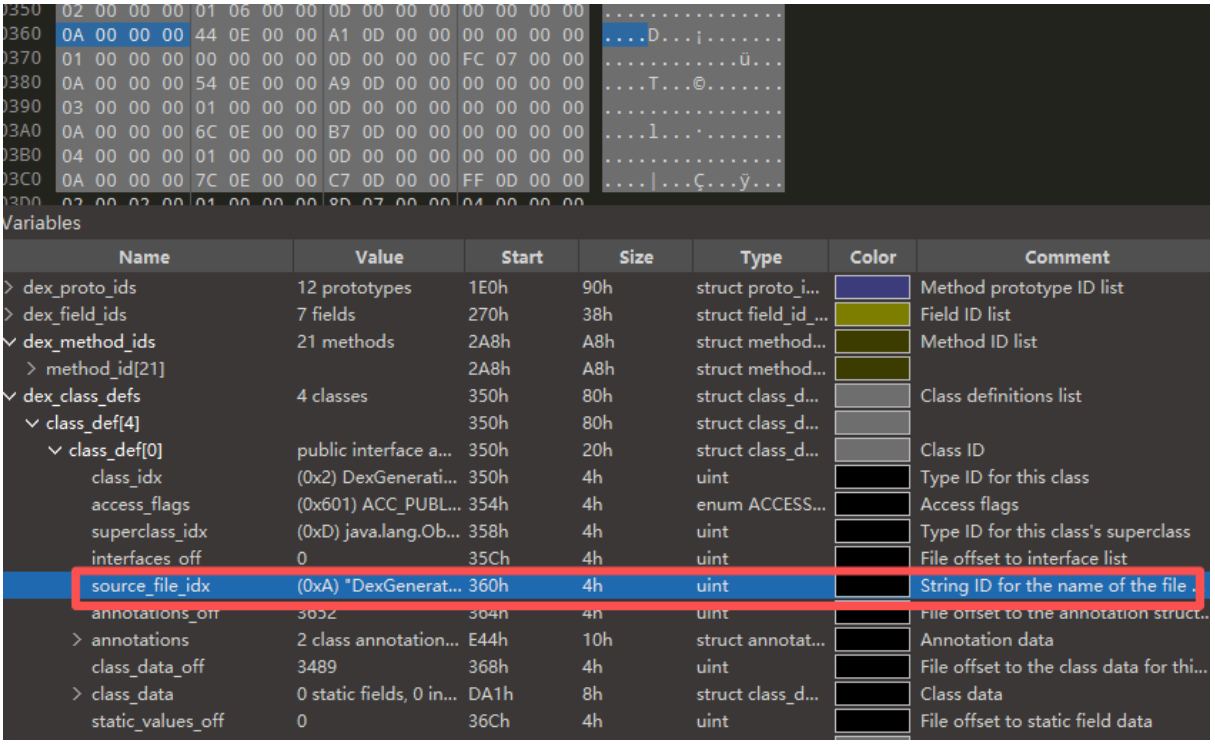
实现的接口列表。这是一个指向 `DexTypeList` 结构的偏移量。如果该类没有实现任何接口，此值为 0。`DexTypeList` 包含了所有接口类型的 `type_ids` 索引。



> method_id[21]		2A8h	A8h	struct method...		
v dex_class_defs	4 classes	350h	80h	struct class_d...		Class definitions list
v class_def[4]		350h	80h	struct class_d...		
v class_def[0]	public interface a...	350h	20h	struct class_d...		Class ID
class_idx	(0x2) DexGenerati...	350h	4h	uint		Type ID for this class
access_flags	(0x601) ACC_PUBL...	354h	4h	enum ACCESS...		Access flags
superclass_idx	(0xD) java.lang.Ob...	358h	4h	uint		Type ID for this class's superclass
interfaces_off	0	35Ch	4h	uint		File offset to interface list
annotations_off	3652	364h	4h	uint		File offset to the annotation struct...
> annotations	2 class annotation...	E44h	10h	struct annotat...		Annotation data
class_data_off	3489	368h	4h	uint		File offset to the class data for thi...
> class_data	0 static fields, 0 in...	DA1h	8h	struct class_d...		Class data
static_values_off	0	36Ch	4h	uint		File offset to static field data

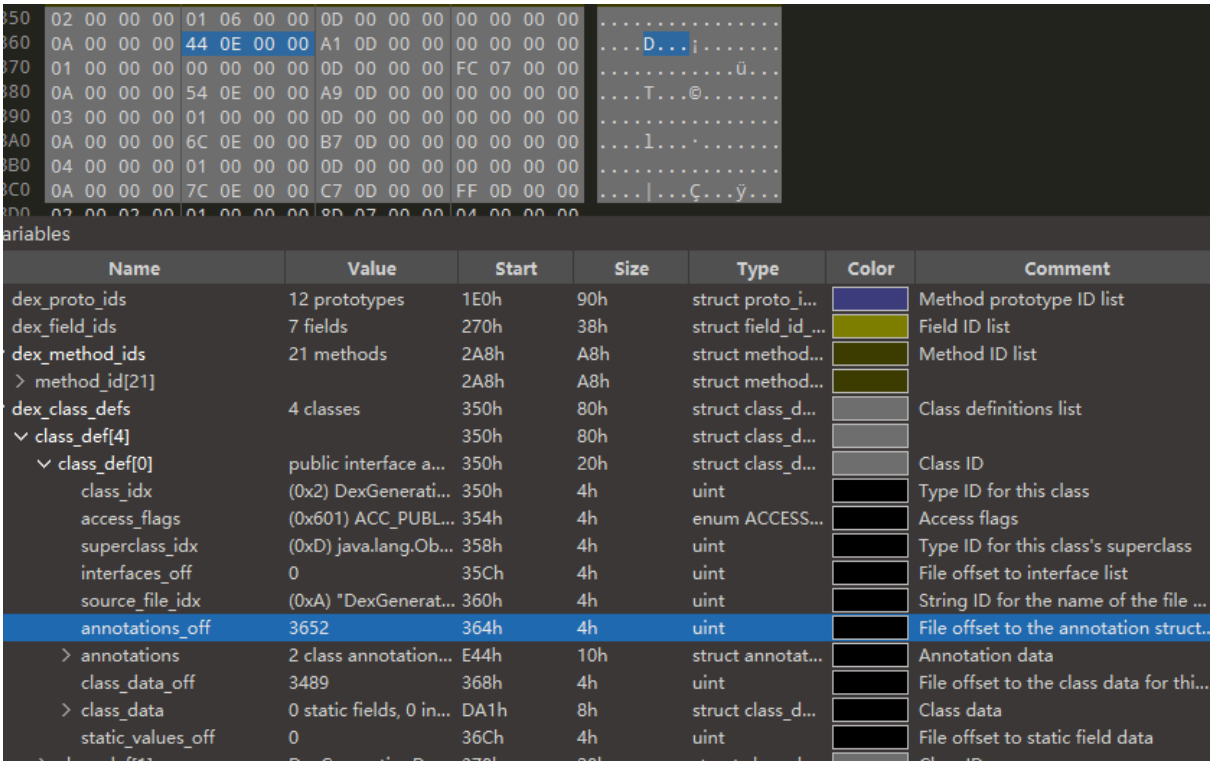
`u4 sourceFileIdx;`

源文件名。这是一个指向 `string_ids` 表的索引，字符串是原始的源代码文件名（如 `"MyClass.java"`）。对于匿名类或某些混淆后的类，此值可能为 `NO_INDEX` （0）。



`u4 annotationsOff;`

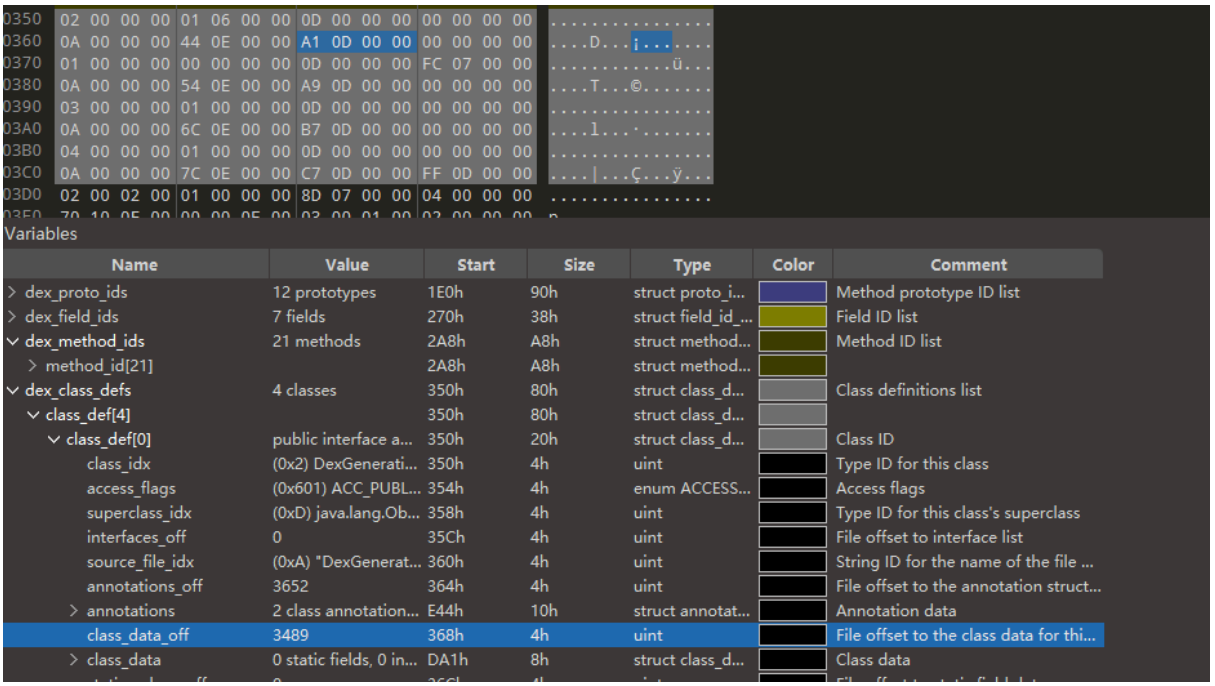
类注解信息。这是一个指向 `annotations_directory_item` 的偏移量。该结构包含了类、字段、方法上的注解的指针。如果没有任何注解，此值为 0。



指向DexAnnotationsDirectoryItem（注解目录），根据类型不同注解类，注解字段，注解方法，注解参数。

`u4 classDataOff;`

类的数据偏移。这是最重要的字段。这是一个指向 `class_data_item` 结构的偏移量。

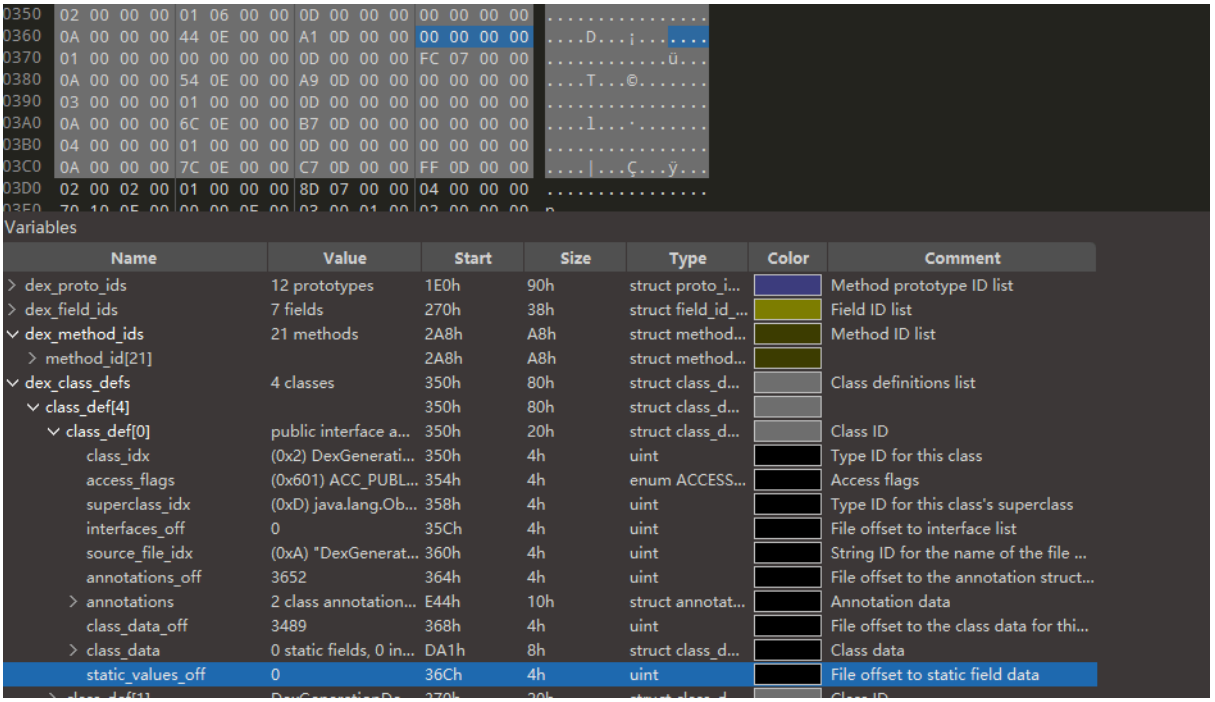


这里偏移位置在0xA1，我们进入DexClassDataItem查看

`class_data_item` 位于数据区，是一个使用 `uleb128` 编码的变长结构，包含了该类所有的**静态字段、实例字段、直接方法**和**虚方法**的具体定义。**如果这个值为 0，则表示该类没有任何数据（例如是一个接口或抽象类）。**

`u4 staticValuesOff;`

静态字段初始值。这是一个指向 `encoded_array_item` 的偏移量。



该结构存储了类中静态字段的初始值（仅包含那些在声明时被显式赋值的原始类型和String类型的静态字段）。如果没有这样的静态字段，此值为 0。

DexAnnotationsDirectoryItem（注解目录）

```
368  /*
369   * Direct-mapped "annotations_directory_item".
370   */
371  struct DexAnnotationsDirectoryItem {
372      u4  classAnnotationsOff;  /* offset to DexAnnotationSetItem */
373      u4  fieldsSize;           /* count of DexFieldAnnotationsItem */
374      u4  methodsSize;          /* count of DexMethodAnnotationsItem */
375      u4  parametersSize;       /* count of DexParameterAnnotationsItem */
376      /* followed by DexFieldAnnotationsItem[fieldsSize] */
377      /* followed by DexMethodAnnotationsItem[methodsSize] */
378      /* followed by DexParameterAnnotationsItem[parametersSize] */
379  };
```

`u4 classAnnotationsOff;`

指向***DexAnnotationsDirectoryItem***结构，注解类

`u4 fieldsSize;`

指向***DexFieldAnnotationsItem***结构，注解字段

`u4 methodsSize;`

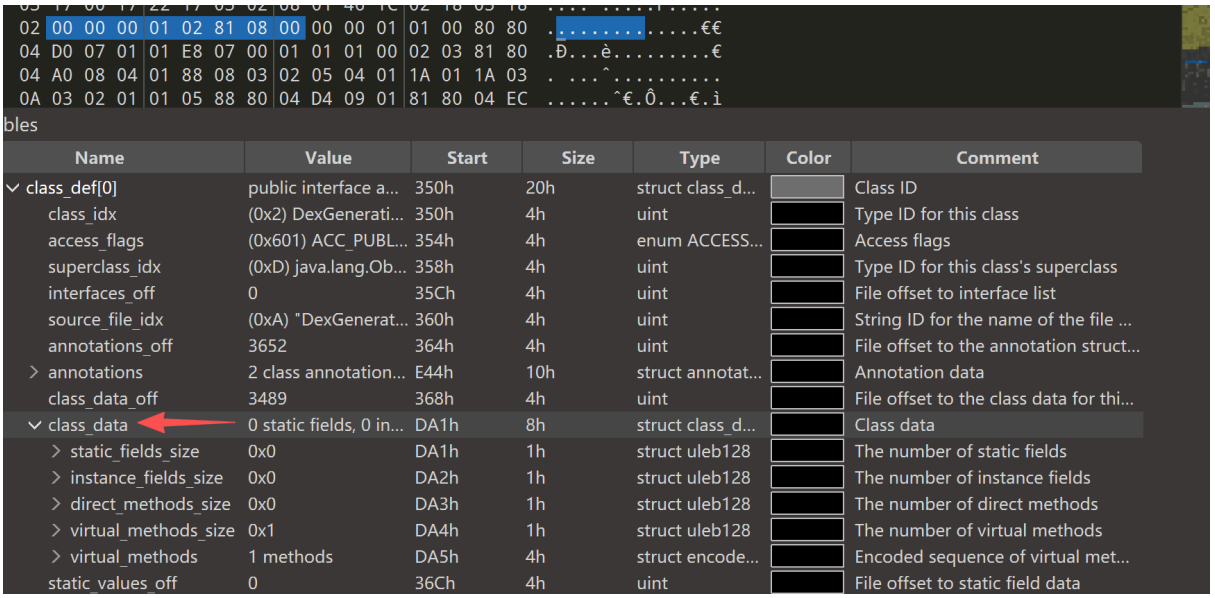
指向***DexMethodAnnotationsItem***结构，注解方法

`u4 parametersSize;`

指向***DexParameterAnnotationsItem*** 结构，注解方法

DexClassData

```
48  /* expanded form of class_data_item. Note: If a particular item is
49   * absent (e.g., no static fields), then the corresponding pointer
50   * is set to NULL. */
51  struct DexClassData {
52      DexClassDataHeader header;
53      DexField*       staticFields;
54      DexField*       instanceFields;
55      DexMethod*      directMethods;
56      DexMethod*      virtualMethods;
57  };
```



因为这里没有字段和直接方法（进入DexClassDataHeader查看），所以后面直接跟的就是虚方法，我们直接进入`DexMethod`查看。

`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;
```

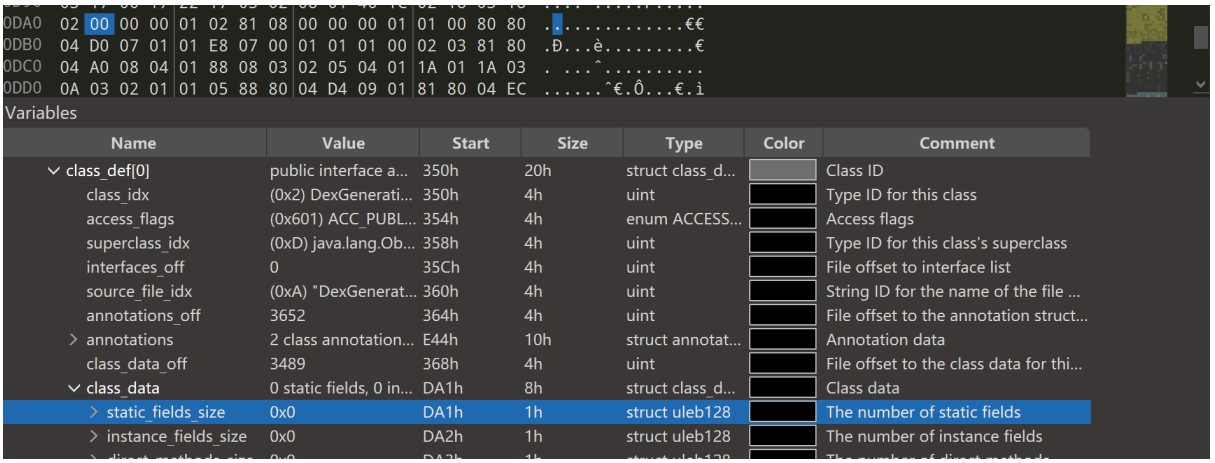
虚方法，DexMethod结构

DexClassDataHeader

```
27  /* expanded form of a class_data_item header */
28  struct DexClassDataHeader {
29      u4 staticFieldsSize;
30      u4 instanceFieldsSize;
31     u4 directMethodsSize;
32      u4 virtualMethodsSize;
33  };
```

```
u4 staticFieldsSize;
```

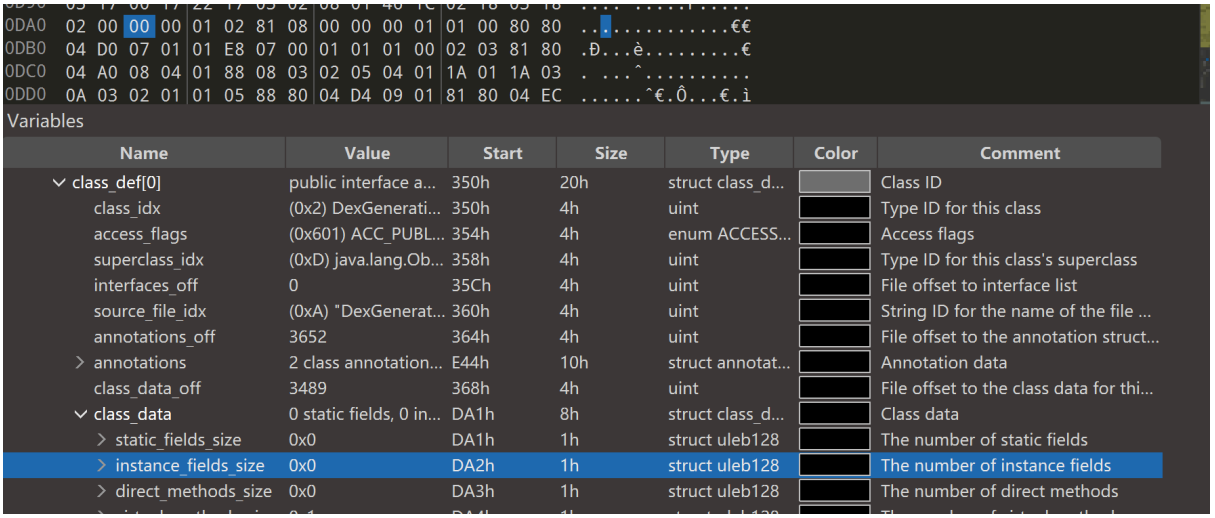
静态字段个数 uleb128



因为是uleb128，首位是0，所以只有1个字节，说明0个静态

```
u4 instanceFieldsSize;
```

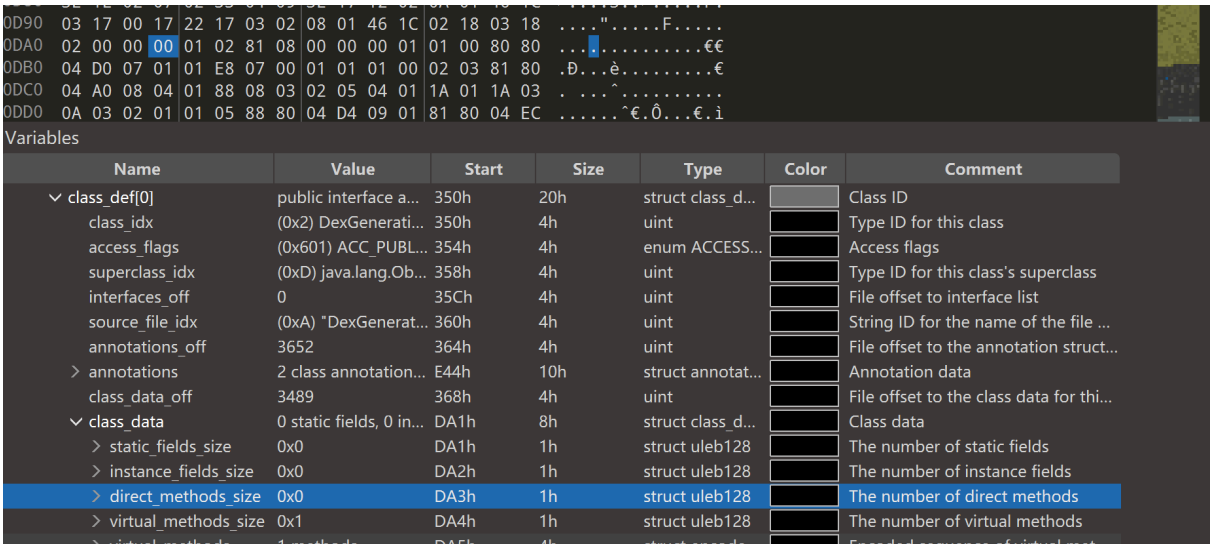
实例字段个数 uleb128



0个实例

```
u4 directMethodsSize;
```

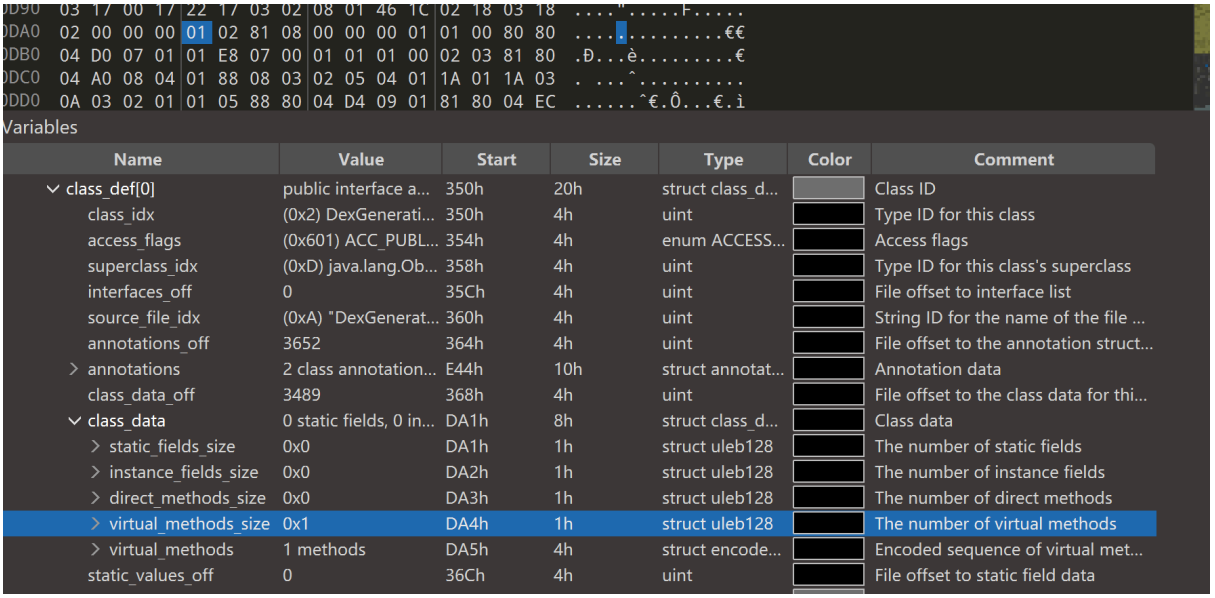
直接方法个数 uleb128



0个直接方法

```
u4 virtualMethodsSize;
```

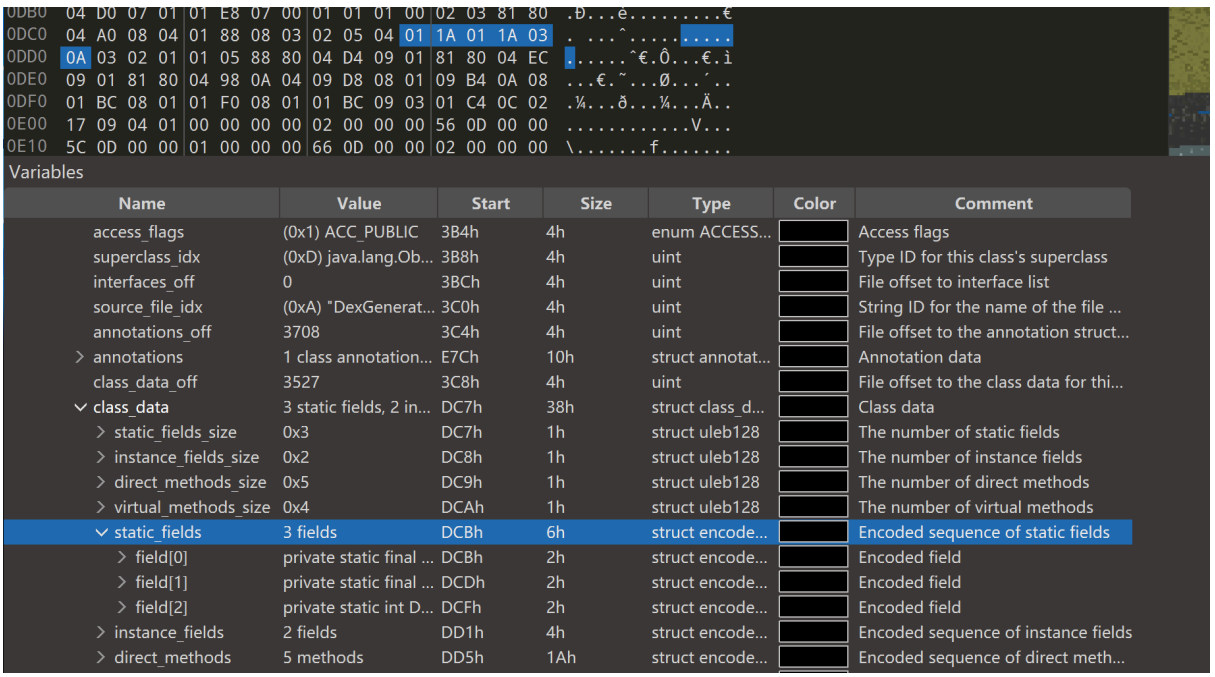
虚方法个数 uleb128



1个虚方法

DexField

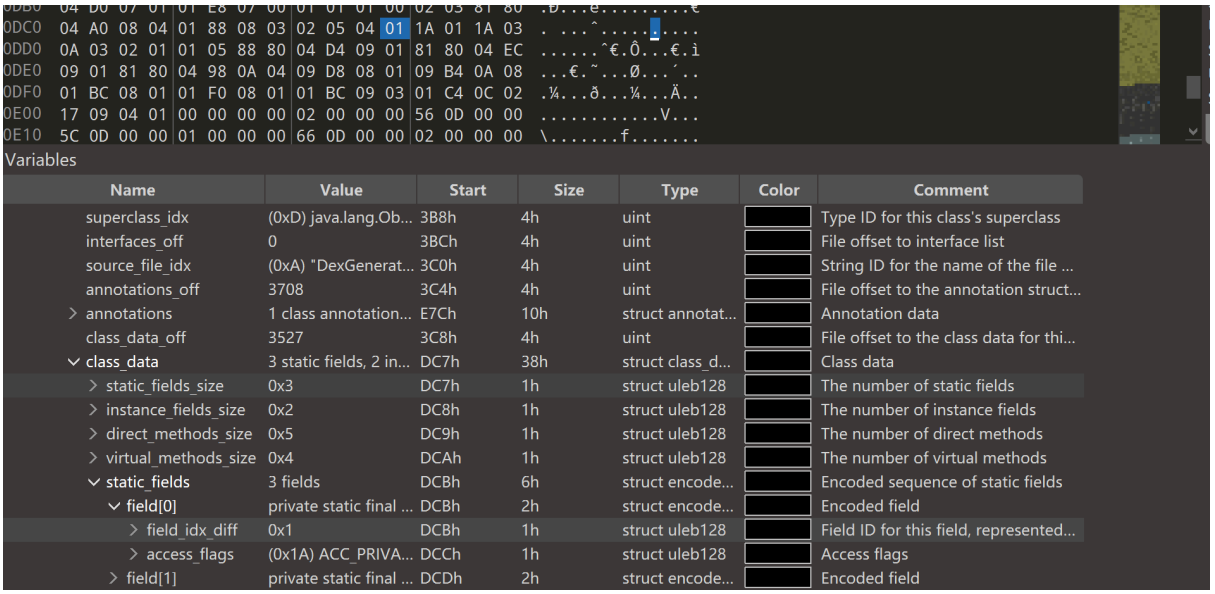
```
35  /* expanded form of encoded_field */
36  struct DexField {
37      u4 fieldIdx;    /* index to a field_id_item */
38      u4 accessFlags;
39  };
```



因为ClassDef[0]中没有字段，这里用ClassDef[3]作演示。

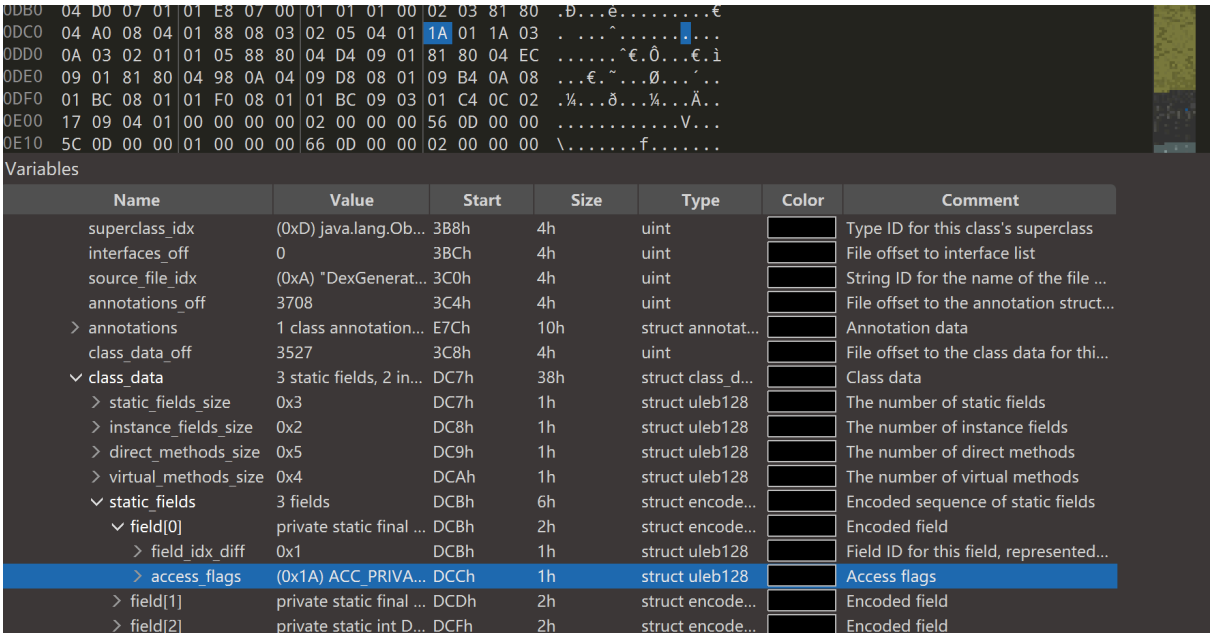
`u4 fieldIdx;`

指向DexFiledId的索引 uleb128



`u4 accessFlags;`

访问标志 uleb128



接着往下排列剩余字段

static_fields	3 fields	DCBh	6h	struct encode...		Encoded sequence of static fields
field[0]	private static final ...	DCBh	2h	struct encode...		Encoded field
field[1]	private static final ...	DCDh	2h	struct encode...		Encoded field
field[2]	private static int D...	DCFh	2h	struct encode...		Encoded field
instance_fields	2 fields	DD1h	4h	struct encode...		Encoded sequence of instance fields
field[0]	private java.lang.S...	DD1h	2h	struct encode...		Encoded field
field[1]	public int DexGen...	DD3h	2h	struct encode...		Encoded field

多个字段听类型的偏移需要相加

DexMethod

```
41  /* expanded form of encoded_method */
42  struct DexMethod {
43      u4 methodIdx;    /* index to a method_id_item */
44      u4 accessFlags;
45      u4 codeOff;      /* file offset to a code_item */
46  };
```

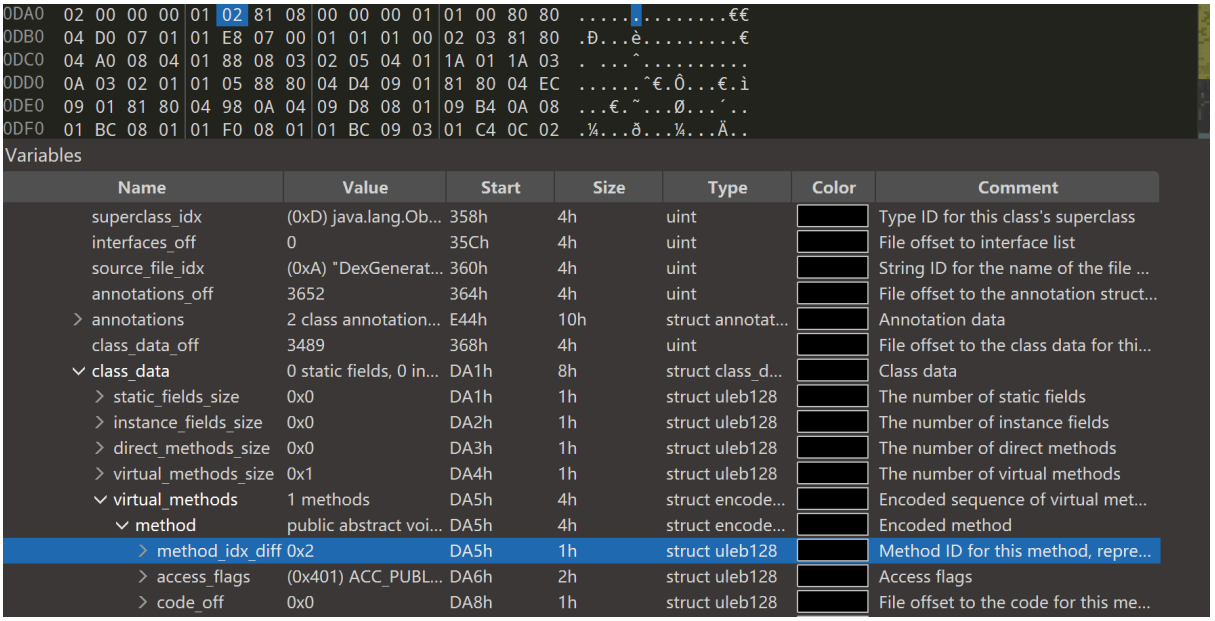
```
0DA0  02 00 00 00 01 02 81 08 00 00 00 01 01 00 80 80  .....¤..........€€
0DB0  04 D0 07 01 01 E8 07 00 01 01 01 00 02 03 81 80  .Ð...è.........€
0DC0  04 A0 08 04 01 88 08 03 02 05 04 01 1A 01 1A 03  . ...ˆ..........
0DD0  0A 03 02 01 01 05 88 80 04 D4 09 01 81 80 04 EC  ......ˆ€.Ô...€.ì
0DE0  09 01 81 80 04 98 0A 04 09 D8 08 01 09 B4 0A 08  ...€.˜...Ø...´..
0DF0  01 BC 08 01 01 F0 08 01 01 BC 09 03 01 C4 0C 02  .¼...ð...¼...Ä..
```

Variables

Name	Value	Start	Size	Type	Color	Comment
superclass_idx	(0xD) java.lang.Ob...	358h	4h	uint		Type ID for this class's superclass
interfaces_off	0	35Ch	4h	uint		File offset to interface list
source_file_idx	(0xA) "DexGenerat...	360h	4h	uint		String ID for the name of the file ...
annotations_off	3652	364h	4h	uint		File offset to the annotation struct...
annotations	2 class annotation...	E44h	10h	struct annotat...		Annotation data
class_data_off	3489	368h	4h	uint		File offset to the class data for thi...
class_data	0 static fields, 0 in...	DA1h	8h	struct class_d...		Class data
static_fields_size	0x0	DA1h	1h	struct uleb128		The number of static fields
instance_fields_size	0x0	DA2h	1h	struct uleb128		The number of instance fields
direct_methods_size	0x0	DA3h	1h	struct uleb128		The number of direct methods
virtual_methods_size	0x1	DA4h	1h	struct uleb128		The number of virtual methods
virtual_methods	1 methods	DA5h	4h	struct encode...		Encoded sequence of virtual met...
method	public abstract voi...	DA5h	4h	struct encode...		Encoded method
method_idx_diff	0x2	DA5h	1h	struct uleb128		Method ID for this method, repre...
access_flags	(0x401) ACC_PUBL...	DA6h	2h	struct uleb128		Access flags
code_off	0x0	DA8h	1h	struct uleb128		File offset to the code for this me...
static_values_off	0	36Ch	4h	uint		File offset to static field data

`u4 methodIdx;`

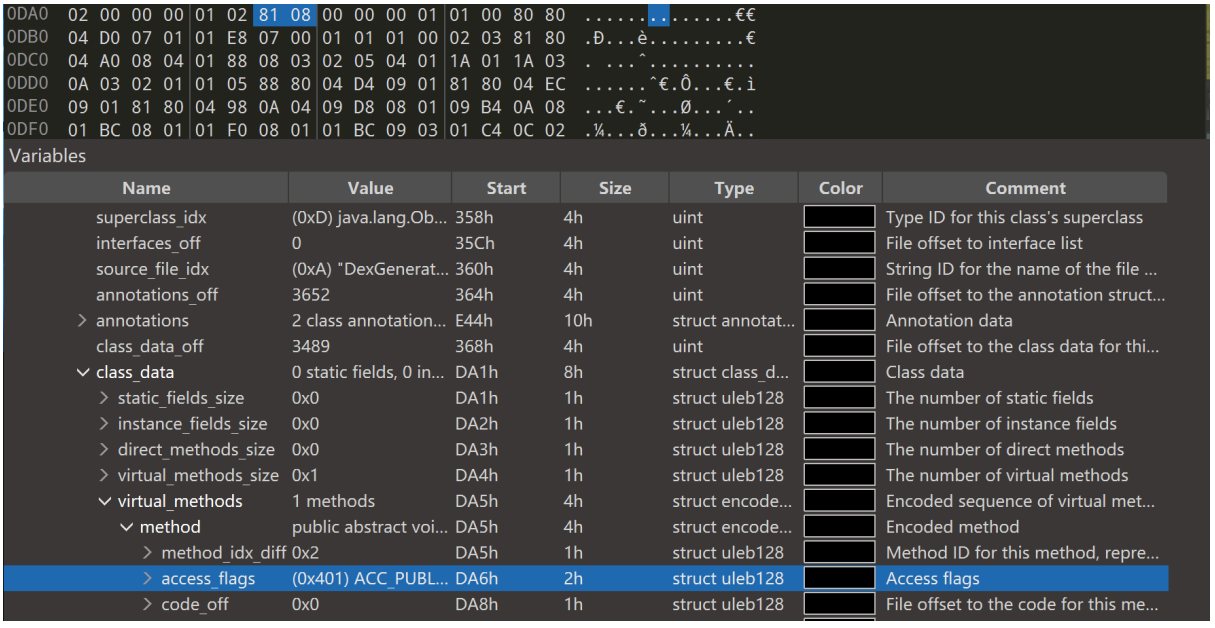
指向DexMethodId的索引 uleb128



这里指向DexMethodId的索引2

`u4 accessFlags;`

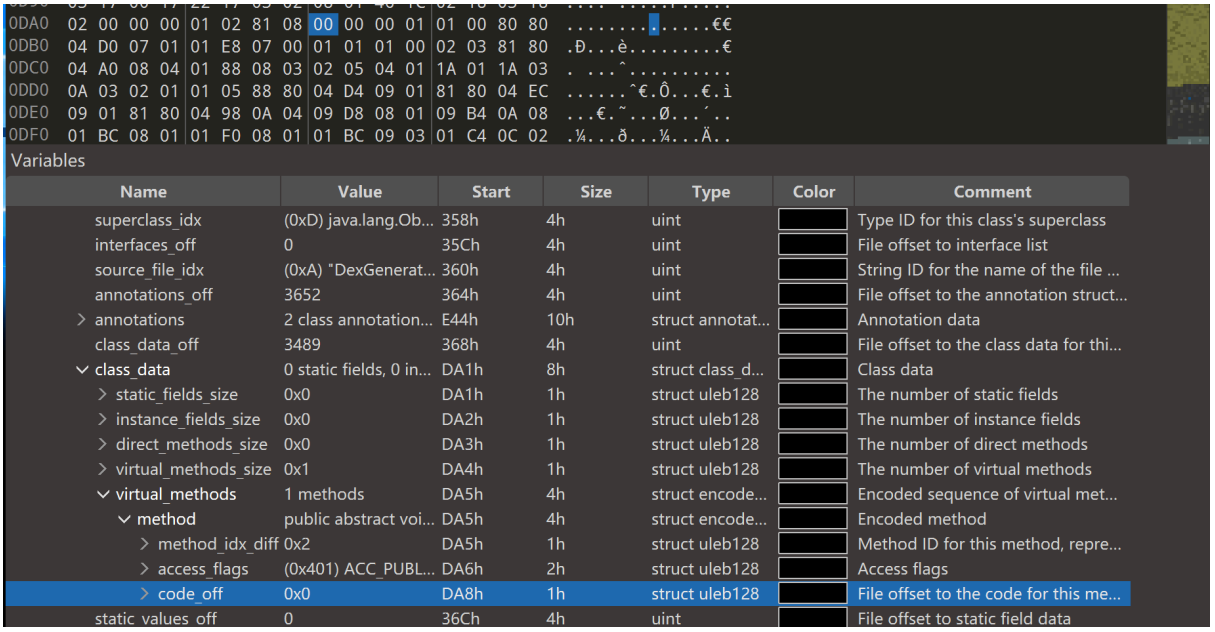
访问标志 uleb128



uleb128它的值转换后是0x401

`u4 codeOff;`

指向DexCode结构的偏移 uleb128



表示没有方法体

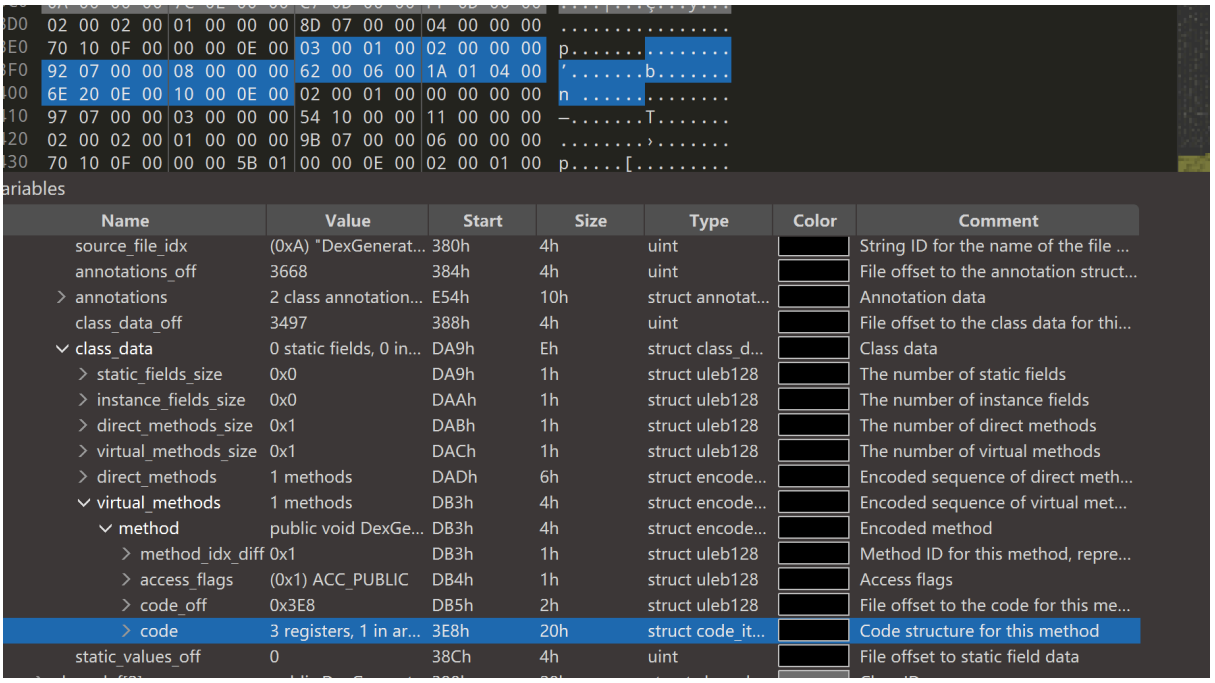
后续多个方法计算索引的时候是需要与前边的同类型方法进行相加

DexCode

它存储了一个方法的**所有执行所需信息**。这包括使用的寄存器数量、参数信息、实际的 Dalvik 字节码指令流以及异常处理（try-catch）信息。

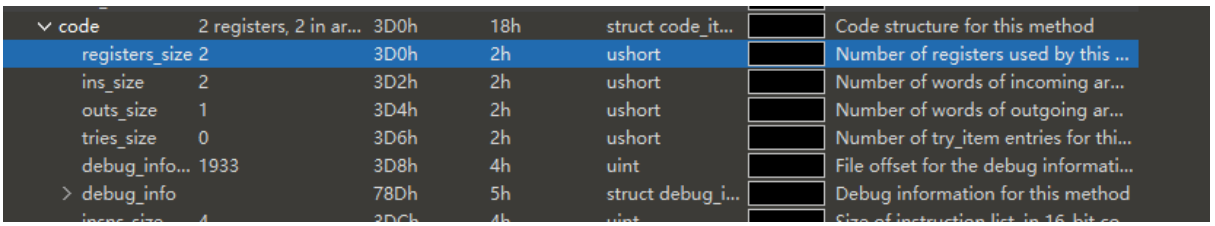
```
330 /*
331  * Direct-mapped "code_item".
332  *
333  * The "catches" table is used when throwing an exception,
334  * "debugInfo" is used when displaying an exception stack trace or
335  * debugging. An offset of zero indicates that there are no entries.
336  */
337 struct DexCode {
338     u2  registersSize;
339     u2  insSize;
340     u2  outsSize;
341     u2  triesSize;
342     u4  debugInfoOff;       /* file offset to debug info stream */
343     u4  insnsSize;          /* size of the insns array, in u2 units */
344     u2  insns[1];
345     /* followed by optional u2 padding */两个字节填充用来对齐
346     /* followed by try_item[triesSize] */try 块描述符数组（可选）。
347     /* followed by uleb128 handlersSize */
```

```
348     /* followed by catch_handler_item[handlersSize] */异常处理程序列表（可选）
349 };
```



`u2 registersSize;`

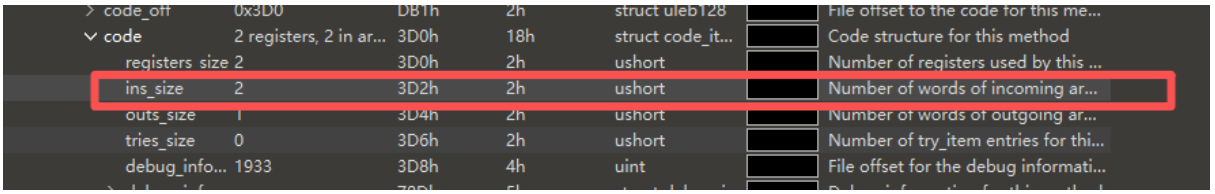
该方法使用寄存器个数



包括所有局部变量和参数占用的寄存器。虚拟机根据这个值来分配运行时栈帧。

`u2 insSize;`

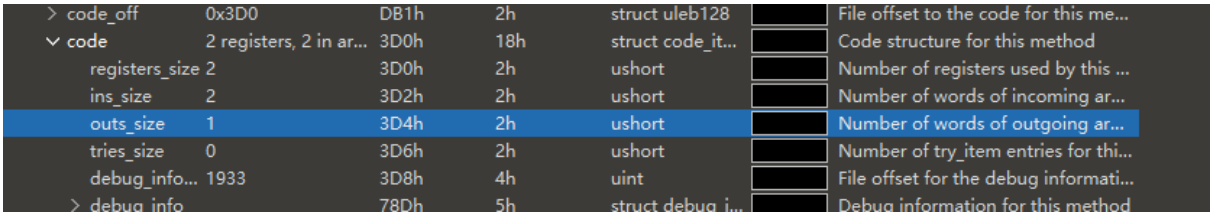
方法的参数个数



对于实例方法，`this` 引用算作第一个参数（在 `v0` 寄存器）。

`u2 outsSize;`

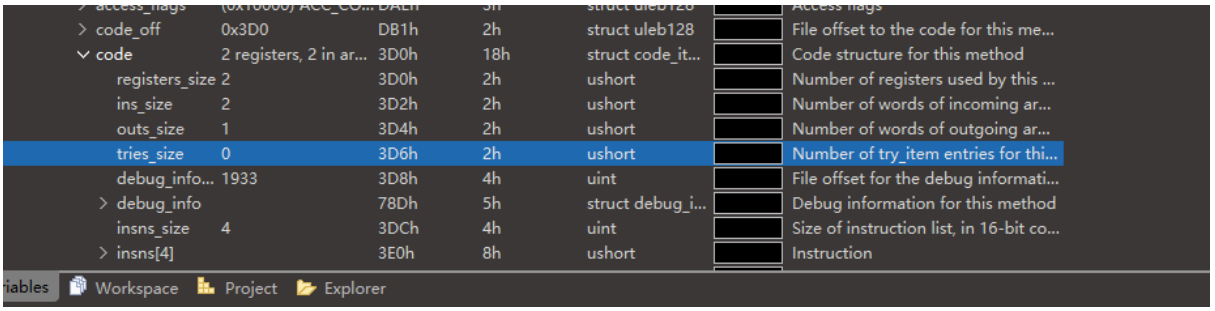
调用其他方法是使用的寄存器个数



输出参数所占的寄存器个数。需要为那个方法的参数提供多少寄存器。这决定了调用方法时栈帧的大小。

```
u2  triesSize;
```

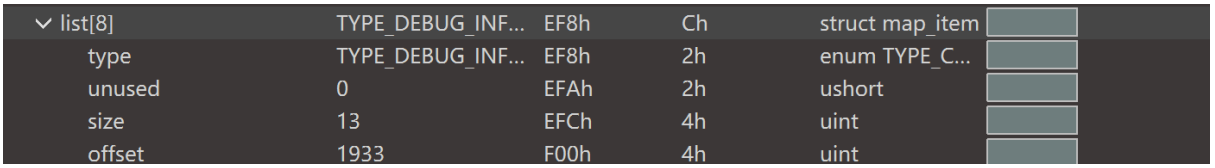
try/catch语句的个数

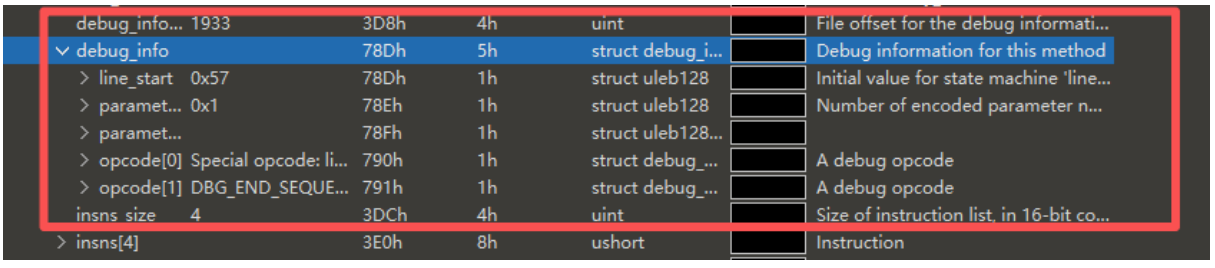


try 块的数量。如果方法中没有 try-catch 块，此值为 0。

```
u4  debugInfoOff;
```

调试信息偏移量dexDexCodeDebugInfo

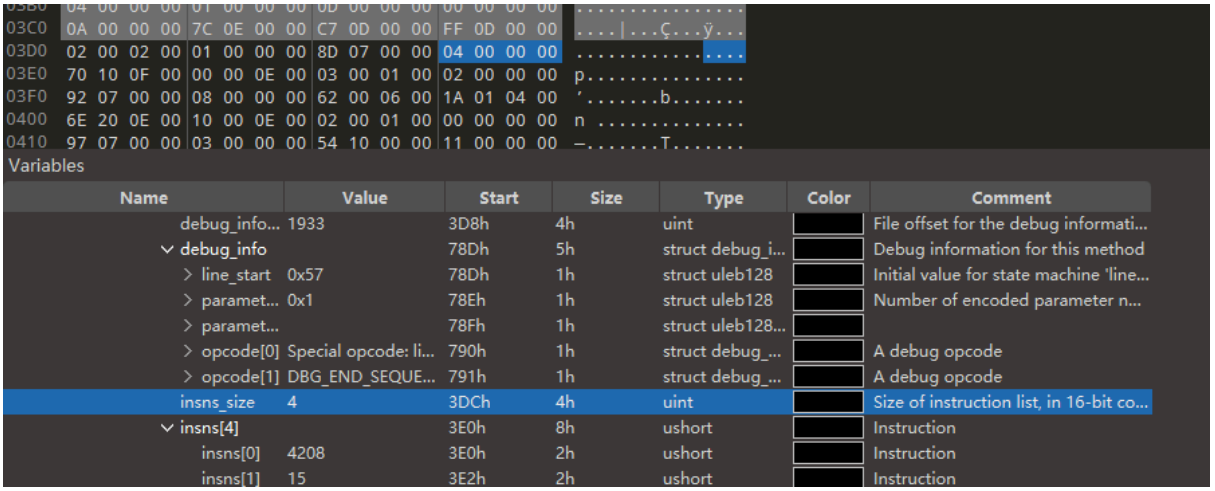




指向一个 `debug_info_item`。该结构包含了行号、局部变量名等调试信息，发布版本中可能被移除或压缩。如果不存在，则为 0。

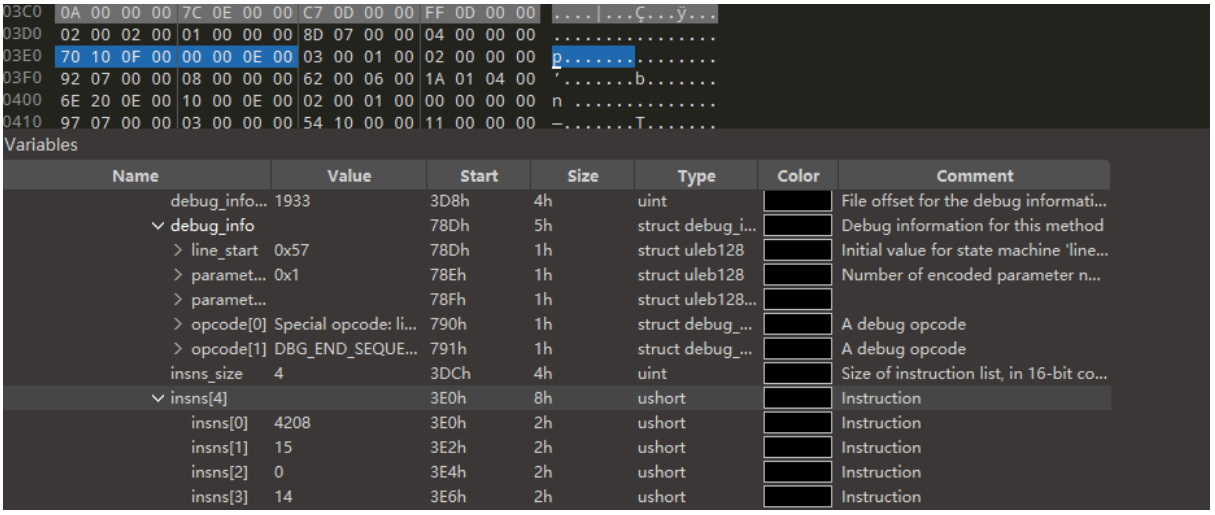
```
u4  insnsSize;
```

指令集个数，两个字节为单位。注意，其单位是 `u2` **（2 字节）**，而不是字节数



指令集的大小。指令集的总字节数为 `insnsSize * 2`。
`u2 insns[1];`

指令集



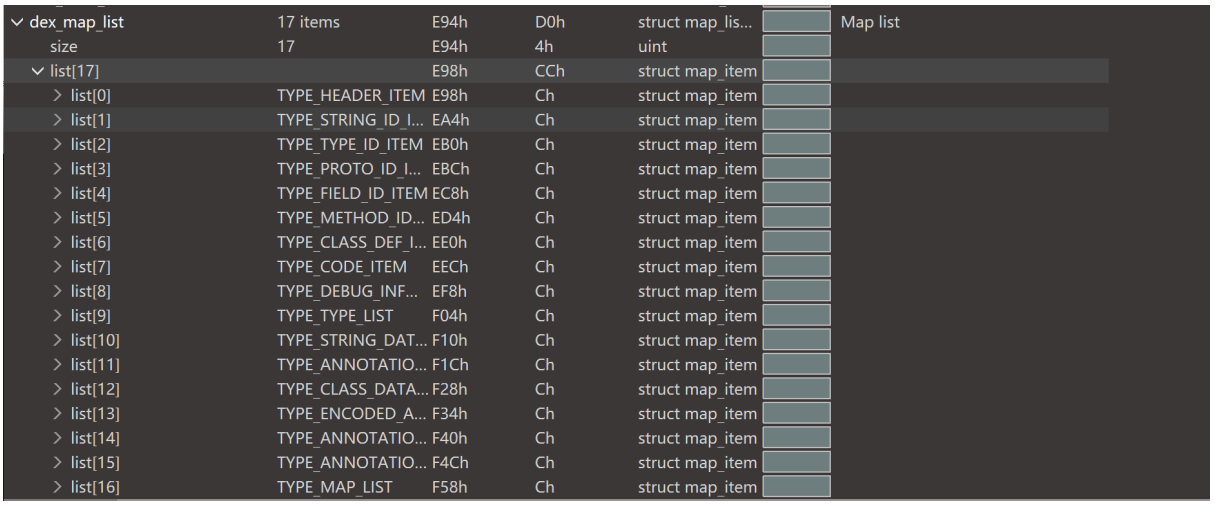
Dalvik 字节码指令流。这是真正的可执行代码。它是一个 `u2` 类型的数组。每条指令的第一个 `u2` 是操作码（opcode），后面跟着数量不等的操作数。Dalvik 是基于寄存器的虚拟机，所以指令操作的是寄存器（如 `v0`, `v1`）而不是 JVM 的操作数栈。

4. Map块(元数据区)

DexMapItem

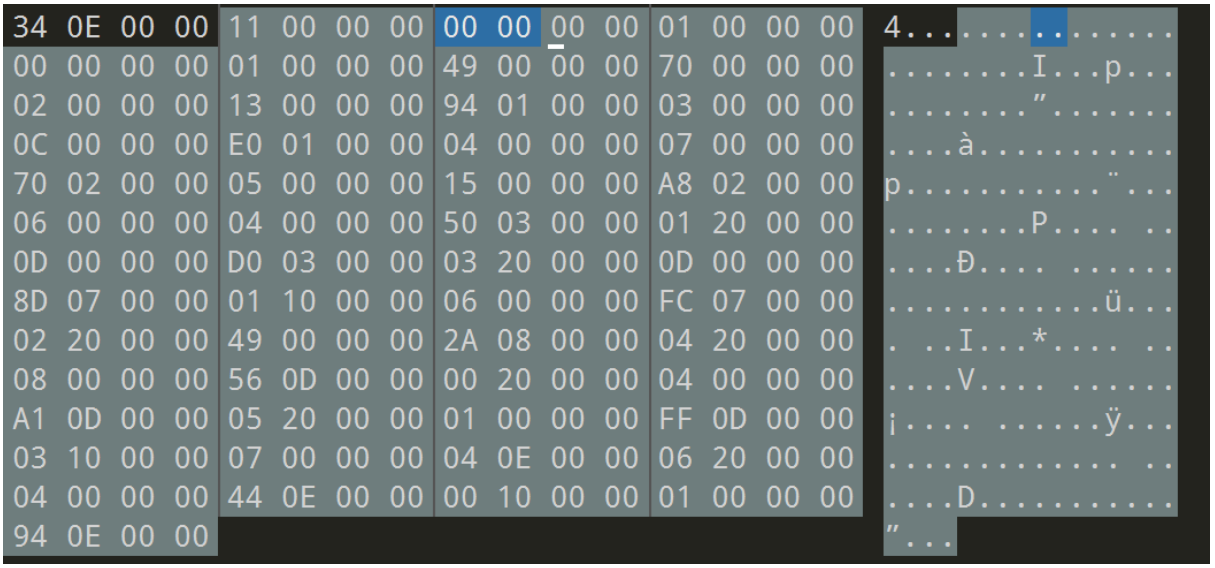
```
242 /*
243  * Direct-mapped "map_item".
244  */
245 struct DexMapItem {
```

```
246     u2 type;            /* type code (see kDexType* above) */
247     u2 unused;
248     u4 size;            /* count of items of the indicated type */
249     u4 offset;          /* file offset to the start of data */
250 };
```



`u2 type;`

kDexType开头的类型



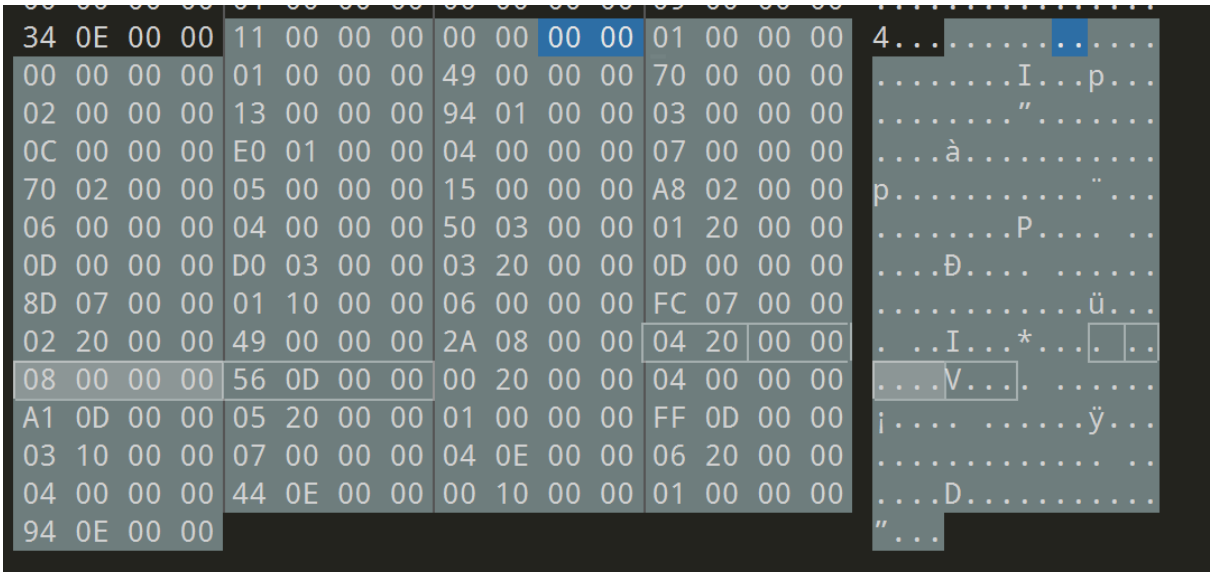
我们可以通过枚举查看代表的是文件头

```
166 /* map item type codes */
167 enum {
168     kDexTypeHeaderItem          = 0x0000,
```

```
169     kDexTypeStringIdItem          = 0x0001,
170     kDexTypeTypeIdItem            = 0x0002,
171     kDexTypeProtoIdItem           = 0x0003,
172     kDexTypeFieldIdItem           = 0x0004,
173     kDexTypeMethodIdItem          = 0x0005,
174     kDexTypeClassDefItem          = 0x0006,
175     kDexTypeMapList               = 0x1000,
176     kDexTypeTypeList              = 0x1001,
177     kDexTypeAnnotationSetRefList  = 0x1002,
178     kDexTypeAnnotationSetItem     = 0x1003,
179     kDexTypeClassDataItem         = 0x2000,
180     kDexTypeCodeItem              = 0x2001,
181     kDexTypeStringDataItem        = 0x2002,
182     kDexTypeDebugInfoItem         = 0x2003,
183     kDexTypeAnnotationItem        = 0x2004,
184     kDexTypeEncodedArrayItem      = 0x2005,
185     kDexTypeAnnotationsDirectoryItem = 0x2006,
186 };
```

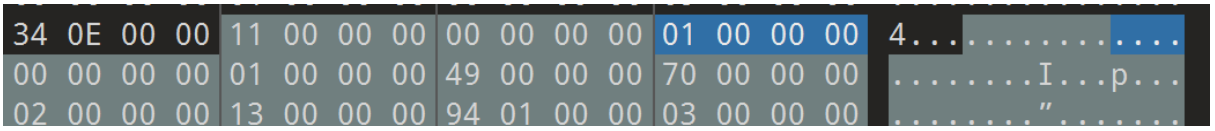
`u2 unused;`

未使用，用于字节对齐



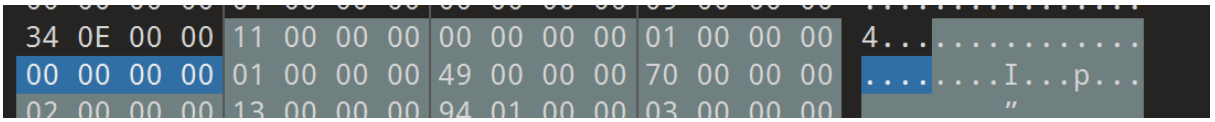
`u4 size;`

DexMapItem中size字段指定特定类型的个数，它们以特定类型在dex文件中连续存放



`u4 offset;`

指定类型数据的文件拍内衣offset为该类型文件起始偏移地址



DexMapList(地图)

地图列表。由 `DexHeader.map_off` 指向。它包含一个 `DexMapItem` 的列表，详细列出了文件中**每一个段的位置和大小**，用于快速验证和遍历。

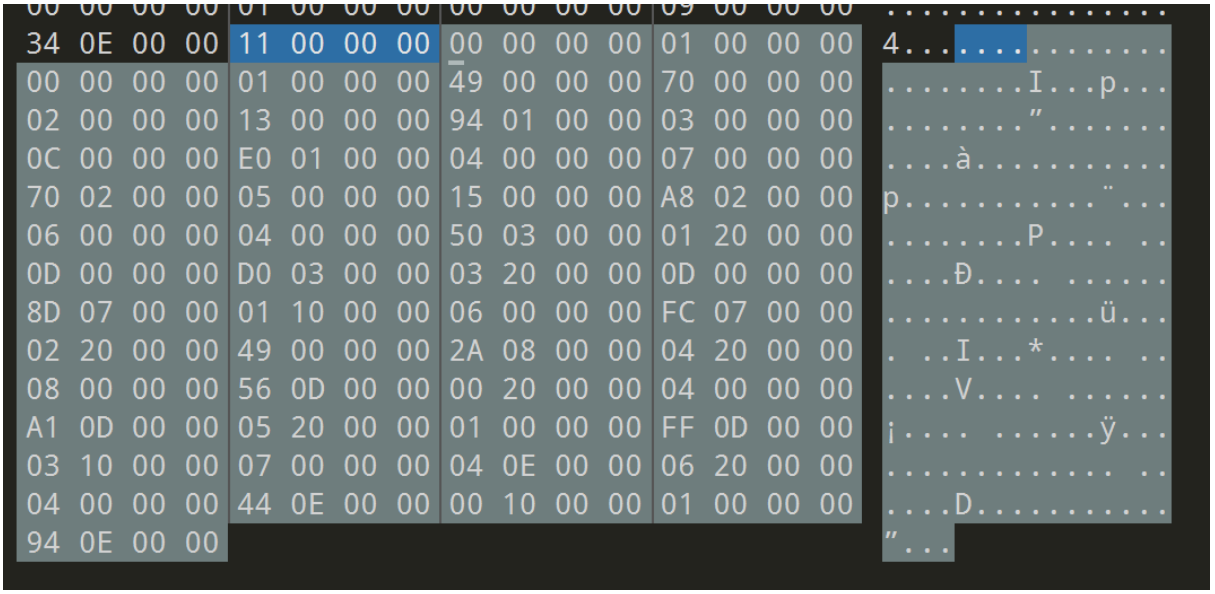
```
252 /*
253  * Direct-mapped "map_list".
254  */
255 struct DexMapList {
256     u4  size;               /* #of entries in list */
257     DexMapItem list[1];     /* entries */
258 };
```

DexHeader 已经包含了各个主要段（如 `string_ids`, `type_ids`）的偏移量和大小。为什么还需要一个 `map_list`？

1. **完整性验证**：提供了一种快速遍历和验证整个文件结构完整性的方法。系统或工具可以读取 `map_list`，然后检查每个段是否在文件边界内，以及段与段之间是否有重叠。
2. **非标准顺序**：虽然 DexHeader 中定义的段有约定的顺序，但 `map_list` 描述的文件段可以以任何顺序出现（尽管标准顺序是优化的）。`map_list` 确保任何解析器都能正确找到它们。
3. **可扩展性**：允许 DEX 文件包含 DexHeader 中未定义的新类型的段。旧的解析器可以跳过它们不认识的段类型，而新的解析器可以通过 `map_list` 来定位这些新段。
4. **快速概览**：为工具（如 010 Editor, dexdump）提供了一种快速获取文件整体布局概览的方法，而无需解析整个文件。

`u4 size;`

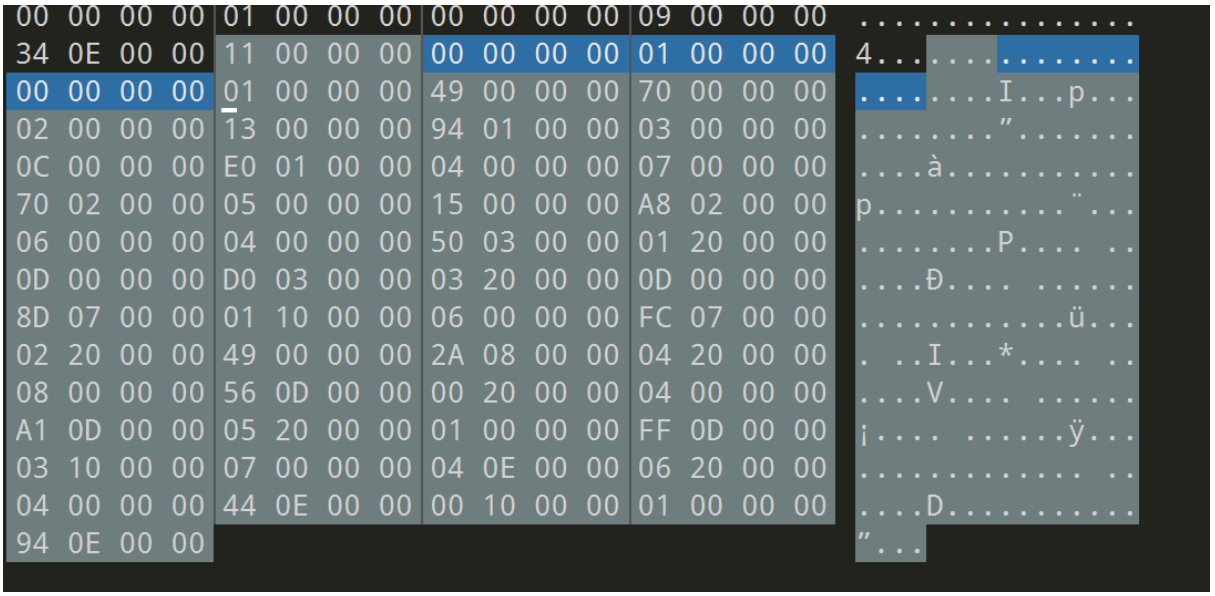
代表一共有多少个`DexMapItem`结构



说明我们接下来会有17个 `DexMapItem` 结构体，一个 `DexMapItem` 结构体占12个字节，也就是说接下来会占204个字节

`DexMapItem list[1];`

`DexMapItem` 结构



我们进入 `DexMapItem` 进行查看

解析流程简例

假设我们要查找 `MainActivity` 类的 `onCreate` 方法。

1. **读取 Header**：获取 `class_defs_off` 和 `class_defs_size`。

2. **遍历 Class Defs Table**：逐个比较每个 `class_def_item` 的 `class_idx` 所指向的字符串是否为 `"Lcom/example/app/MainActivity;"`。
3. **找到目标类**：找到对应的 `class_def_item`，读取其 `class_data_off`。
4. **解析 Class Data**：跳到 `class_data_off` 位置，解析 LEB128 数据，找到方法列表。逐个比较每个方法的 `method_idx`，这个 `method_idx` 指向 `method_ids` 表。
5. **在 Method Table 中确认**：通过 `method_idx` 在 `method_ids` 中找到对应的方法项，检查其 `name_idx` 指向的字符串是否为 `"onCreate"`，其 `proto_idx` 指向的原型是否为 `"(Landroid/os/Bundle;)V"`。
6. **获取字节码**：在 `Class Data` 的方法列表中找到该方法的项，其中包含了指向 `Code Item` 的偏移量 `code_off`。
7. **解析 Code Item**：跳到 `code_off`，读取 `registers_size`，`ins_size`，`insns` 等字段。`insns` 就是 `onCreate` 方法的 Dalvik 字节码指令流。