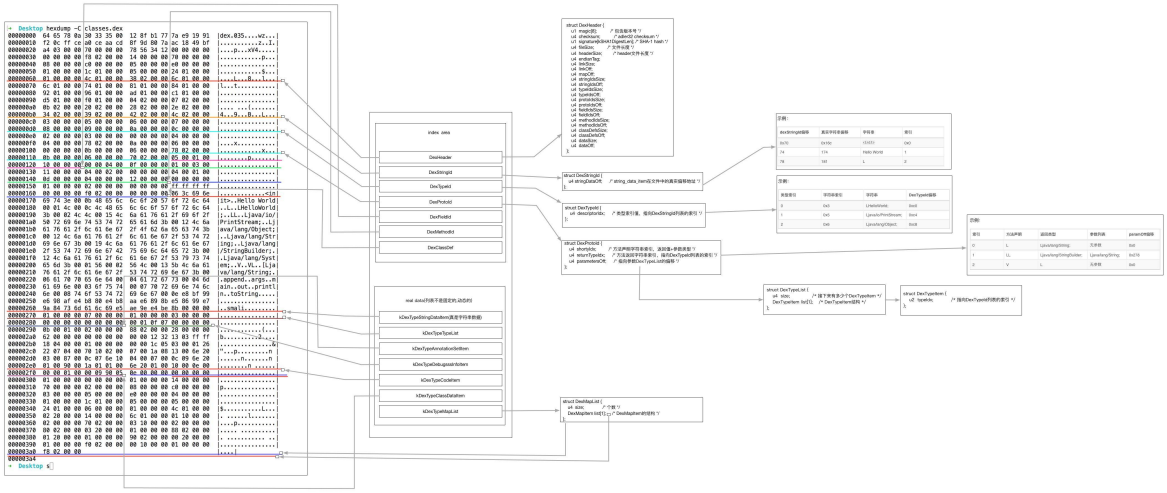
一张图搞懂dex

[大图这里]



当然也可以通过下面的图12 DexFile的文件格式，了解更清楚。

DEX文件详解

- 什么是dex文件?
- 如何生成一个dex文件
- dex文件的作用
- dex文件格式详解

什么是dex文件?

dex文件是Android系统中的一种文件，是一种特殊的数据格式，和APK、jar 等格式文件类似。
能够被DVM识别，加载并执行的文件格式。
简单说就是优化后的android版.exe。每个apk安装包里都有。包含应用程序的全部操作指令以及运行时数据。
相对于PC上的java虚拟机能运行.class；android上的Davlik虚拟机能运行.dex。

当java程序编译成class后，还需要使用dx工具将所有的class文件整合到一个dex文件，目的是其中各个类能够共享数据，在一定程度上降低了冗余，同时也是文件结构更加经凑，实验表明，dex文件是传统jar文件大小的50%左右

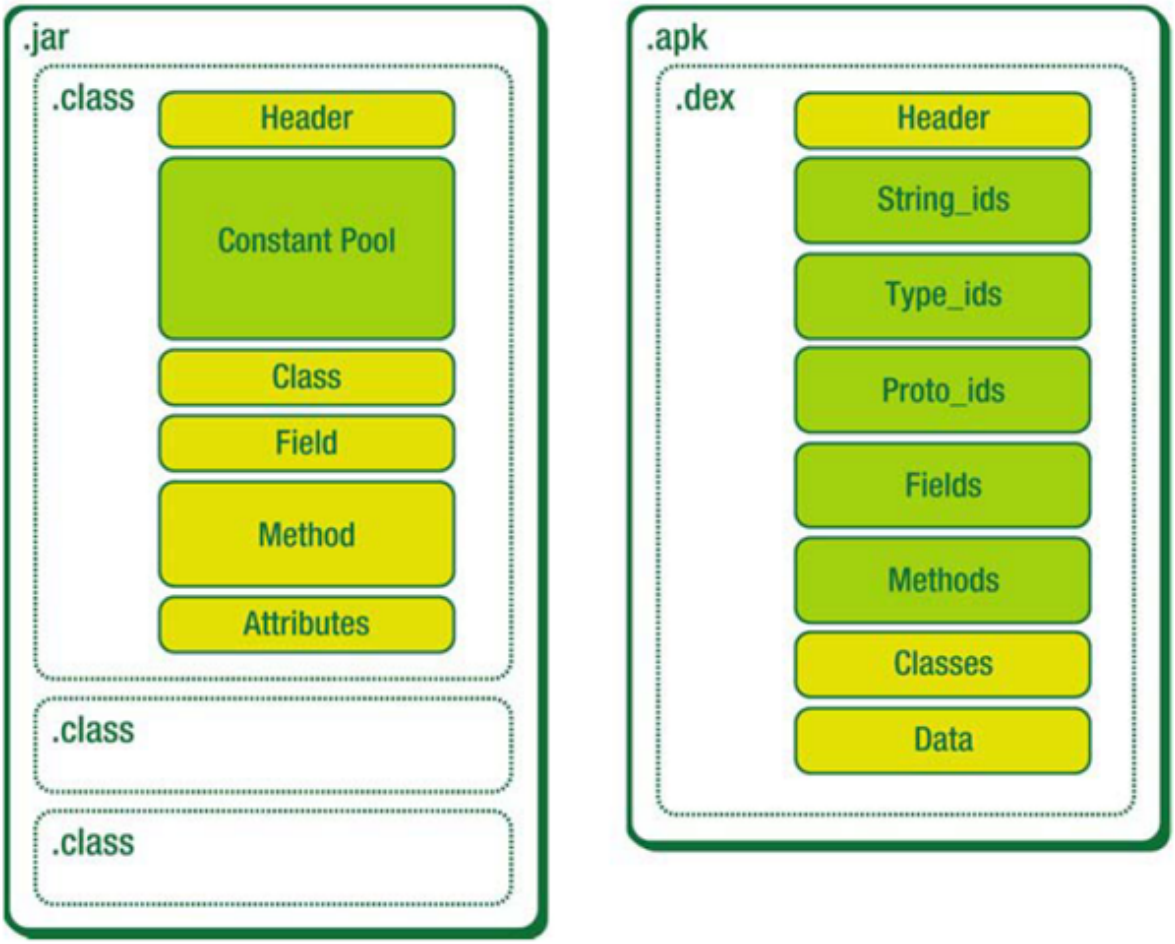


Figure 3-2. *Class file vs DEX file*

图2 apk中的dex文件

为何要研究dex格式？因为dex里面包含了所有app代码，利用反编译工具可以获取java源码。理解并修改dex文件，就能更好的apk破解和防破解。

使用dex文件的最大目的是实现安全管理，但在追求安全的前提下，一定要注意对dex文件实现优化处理。

注意：并不是只有Java才可以生成dex文件，C和C++也可以生成dex文件

如何生成一个dex文件?

1. 通过IDE自动帮我们build生成
2. 手动通过dx命令去生成dex文件
   在待测试的class文件目录下（我将TestMain.class放到了F盘根目录下）,执行命令`dx --dex --output TestMain.dex TestMain.class`,就会生成TestMain.dex文件。
3. 手动运行dex文件在手机
   在待测试的dex文件目录下（我将TestMain.class放到了F盘根目录下），通过`adb push TestMain.dex /storage/emulated/0`命令，然后通过`adb shell`命令进入手机，后执行`dalvikvm -cp /sdcard/TestMain.dex TestMain`，就会打印出

```
Hello World!
```

如下图3所示（使用AS的终端，没有用Windows的cmd命令）

```
E:\AndroidStudioProjects\MyTest>F:

F:\>adb push TestMain.dex /storage/emulated/0
TestMain.dex: 1 file pushed. 0.1 MB/s (892 bytes in 0.011s)

F:\>adb shell
aqua:/ $ dalvikvm -cp /sdcard/TestMain.dex TestMain
Hello World!
aqua:/ $
```

图3 手动运行dex文件

注意：

- 环境变量的配置，dex在SDK目录下的build-tools目录下有很多版本，这里可以选择最新版本目录下dx.bat配置到环境变量path路径下；同样，adb命令也同样配置。
- 运行完dex文件，可以通过exit退出手机，电脑本地盘符

查看dex文件

大图这里

图4 dex文件概貌

通过010Editor工具（图片来自网络）

大图这里

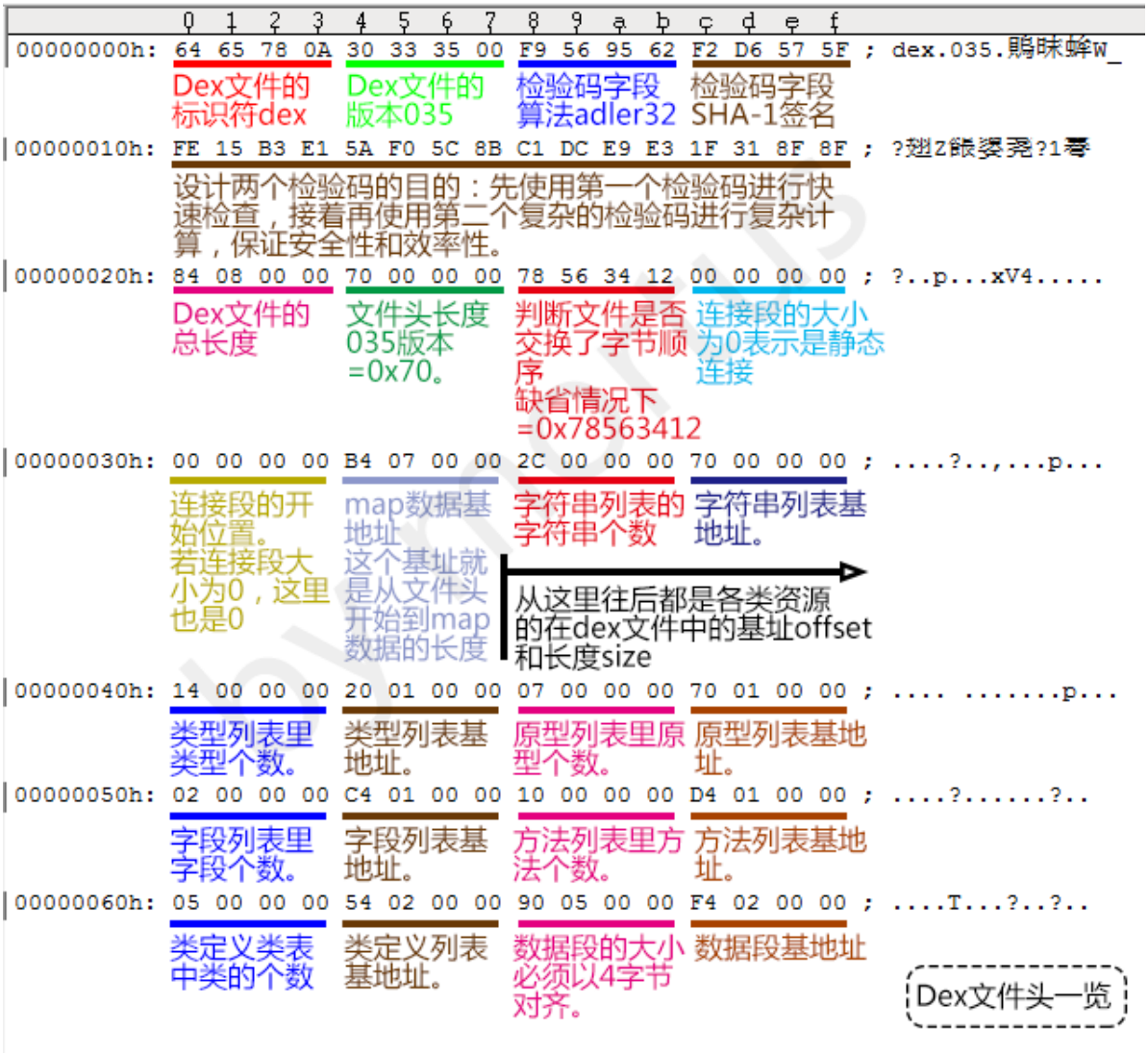
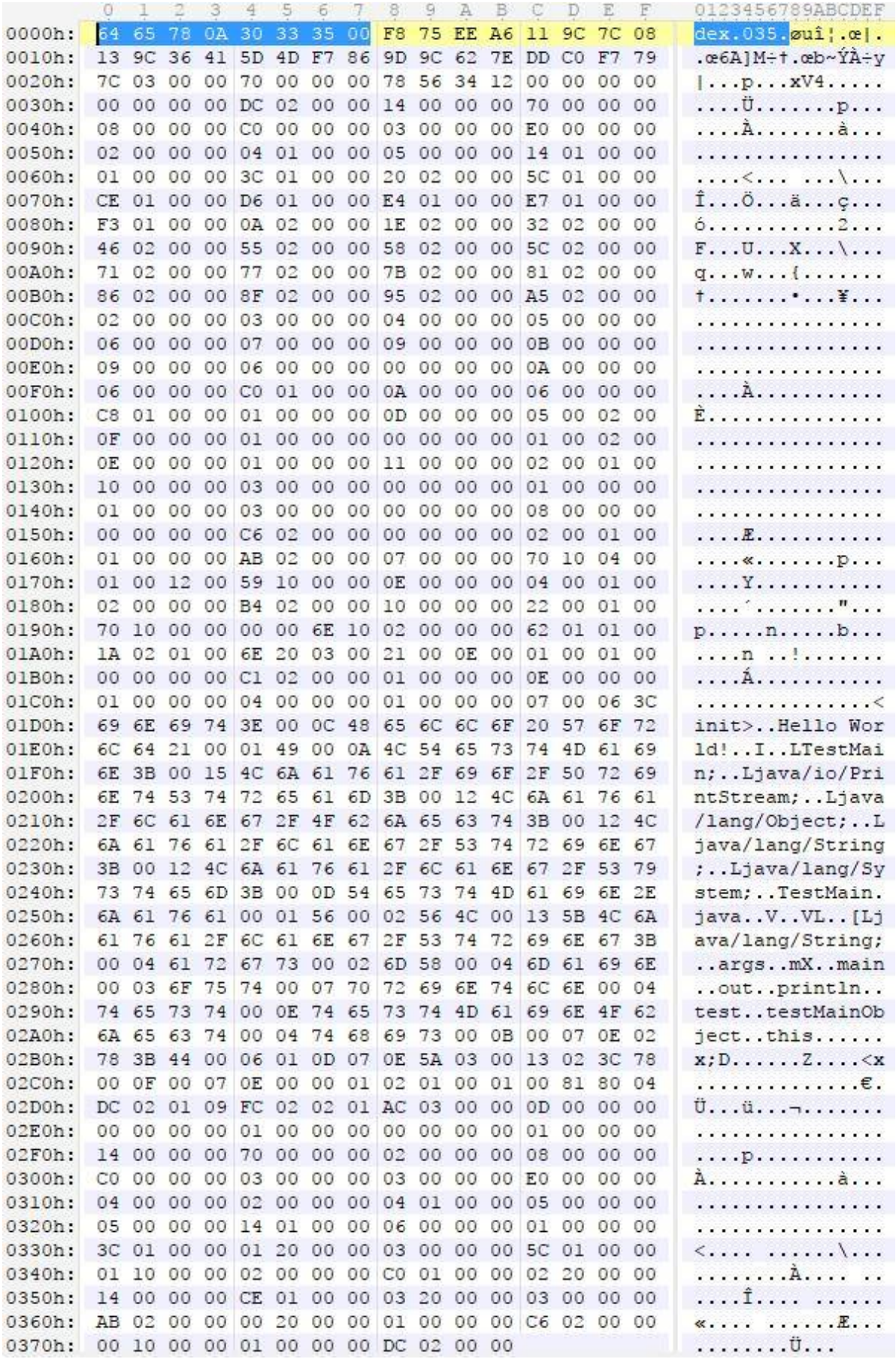


图5 注：图片来自网络

下图6是TestMain.dex通过010Editor工具得到的

大图这里



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0000h:	64	65	78	0A	30	33	35	00	F8	75	EE	A6	11	9C	7C	08	dex.035.øuî¦.œ|.
0010h:	13	9C	36	41	5D	4D	F7	86	9D	9C	62	7E	DD	C0	F7	79	.œ6A]M÷†.œb~ÝÀ÷y
0020h:	7C	03	00	00	70	00	00	00	78	56	34	12	00	00	00	00	|...p...xV4.....
0030h:	00	00	00	00	DC	02	00	00	14	00	00	00	70	00	00	00	....Ü.......p...
0040h:	08	00	00	00	C0	00	00	00	03	00	00	00	E0	00	00	00	....À.......à...
0050h:	02	00	00	00	04	01	00	00	05	00	00	00	14	01	00	00	....
0060h:	01	00	00	00	3C	01	00	00	20	02	00	00	5C	01	00	00	....<... ...\...
0070h:	CE	01	00	00	D6	01	00	00	E4	01	00	00	E7	01	00	00	Î...Ö...ä...ç...
0080h:	F3	01	00	00	0A	02	00	00	1E	02	00	00	32	02	00	00	ó...........2...
0090h:	46	02	00	00	55	02	00	00	58	02	00	00	5C	02	00	00	F...U...X...\...
00A0h:	71	02	00	00	77	02	00	00	7B	02	00	00	81	02	00	00	q...w...{.......
00B0h:	86	02	00	00	8F	02	00	00	95	02	00	00	A5	02	00	00	†.......•...¥...
00C0h:	02	00	00	00	03	00	00	00	04	00	00	00	05	00	00	00	....
00D0h:	06	00	00	00	07	00	00	00	09	00	00	00	0B	00	00	00	....
00E0h:	09	00	00	00	06	00	00	00	00	00	00	00	0A	00	00	00	....
00F0h:	06	00	00	00	C0	01	00	00	0A	00	00	00	06	00	00	00	....À...........
0100h:	C8	01	00	00	01	00	00	00	0D	00	00	00	05	00	02	00	È...............
0110h:	0F	00	00	00	01	00	00	00	00	00	00	00	01	00	02	00	....
0120h:	0E	00	00	00	01	00	00	00	11	00	00	00	02	00	01	00	....
0130h:	10	00	00	00	03	00	00	00	00	00	00	00	01	00	00	00	....
0140h:	01	00	00	00	03	00	00	00	00	00	00	00	08	00	00	00	....
0150h:	00	00	00	00	C6	02	00	00	00	00	00	00	02	00	01	00	....Æ...........
0160h:	01	00	00	00	AB	02	00	00	07	00	00	00	70	10	04	00	....«.......p...
0170h:	01	00	12	00	59	10	00	00	0E	00	00	00	04	00	01	00	....Y...........
0180h:	02	00	00	00	B4	02	00	00	10	00	00	00	22	00	01	00	....´......."....
0190h:	70	10	00	00	00	00	6E	10	02	00	00	00	62	01	01	00	p.....n.....b...
01A0h:	1A	02	01	00	6E	20	03	00	21	00	0E	00	01	00	01	00	....n ..!.......
01B0h:	00	00	00	00	C1	02	00	00	01	00	00	00	0E	00	00	00	....Á...........
01C0h:	01	00	00	00	04	00	00	00	01	00	00	00	07	00	06	3C	...............<
01D0h:	69	6E	69	74	3E	00	0C	48	65	6C	6C	6F	20	57	6F	72	init>..Hello Wor
01E0h:	6C	64	21	00	01	49	00	0A	4C	54	65	73	74	4D	61	69	ld!..I..LTestMai
01F0h:	6E	3B	00	15	4C	6A	61	76	61	2F	69	6F	2F	50	72	69	n;..Ljava/io/Pri
0200h:	6E	74	53	74	72	65	61	6D	3B	00	12	4C	6A	61	76	61	ntStream;..Ljava
0210h:	2F	6C	61	6E	67	2F	4F	62	6A	65	63	74	3B	00	12	4C	/lang/Object;..L
0220h:	6A	61	76	61	2F	6C	61	6E	67	2F	53	74	72	69	6E	67	java/lang/String
0230h:	3B	00	12	4C	6A	61	76	61	2F	6C	61	6E	67	2F	53	79	;..Ljava/lang/Sy
0240h:	73	74	65	6D	3B	00	0D	54	65	73	74	4D	61	69	6E	2E	stem;..TestMain.
0250h:	6A	61	76	61	00	01	56	00	02	56	4C	00	13	5B	4C	6A	java..V..VL..[Lj
0260h:	61	76	61	2F	6C	61	6E	67	2F	53	74	72	69	6E	67	3B	ava/lang/String;
0270h:	00	04	61	72	67	73	00	02	6D	58	00	04	6D	61	69	6E	..args..mX..main
0280h:	00	03	6F	75	74	00	07	70	72	69	6E	74	6C	6E	00	04	..out..println..
0290h:	74	65	73	74	00	0E	74	65	73	74	4D	61	69	6E	4F	62	test..testMainOb
02A0h:	6A	65	63	74	00	04	74	68	69	73	00	0B	00	07	0E	02	ject..this......
02B0h:	78	3B	44	00	06	01	0D	07	0E	5A	03	00	13	02	3C	78	x;D......Z....<x
02C0h:	00	0F	00	07	0E	00	00	01	02	01	00	01	00	81	80	04	..............€.
02D0h:	DC	02	01	09	FC	02	02	01	AC	03	00	00	0D	00	00	00	Ü...ü...¬.......
02E0h:	00	00	00	00	01	00	00	00	00	00	00	00	01	00	00	00	....
02F0h:	14	00	00	00	70	00	00	00	02	00	00	00	08	00	00	00	....p...........
0300h:	C0	00	00	00	03	00	00	00	03	00	00	00	E0	00	00	00	À...........à...
0310h:	04	00	00	00	02	00	00	00	04	01	00	00	05	00	00	00	....
0320h:	05	00	00	00	14	01	00	00	06	00	00	00	01	00	00	00	....
0330h:	3C	01	00	00	01	20	00	00	03	00	00	00	5C	01	00	00	<.... ......\...
0340h:	01	10	00	00	02	00	00	00	C0	01	00	00	02	20	00	00	........À.... ..
0350h:	14	00	00	00	CE	01	00	00	03	20	00	00	03	00	00	00	....Î.... ......
0360h:	AB	02	00	00	00	20	00	00	01	00	00	00	C6	02	00	00	«.... ......Æ...
0370h:	00	10	00	00	01	00	00	00	DC	02	00	00					........Ü...

图6 010Editor 检测TestMain.dex结果

图7是通过010Editor工具检测TestMain.dex得到的 Template Result结果

大图这里

Template Results - DEX.bt

Name	Value	Start	Size	Color	Comment
struct header_item dex_header		0h	70h	Fg: Bg:	Dex file header
struct dex_magic magic	dex 035	0h	8h	Fg: Bg:	Magic value
uint checksum	A6EE75F8h	8h	4h	Fg: Bg:	Alder32 checksum of rest of file
SHA1 signature[20]	119C7C08139C36415D4DF7869D9C627EDDC0F779	Ch	14h	Fg: Bg:	SHA-1 signature of rest of file
uint file_size	892	20h	4h	Fg: Bg:	File size in bytes
uint header_size	112	24h	4h	Fg: Bg:	Header size in bytes
uint endian_tag	12345678h	28h	4h	Fg: Bg:	Endianness tag
uint link_size	0	2Ch	4h	Fg: Bg:	Size of link section
uint link_off	0	30h	4h	Fg: Bg:	File offset of link section
uint map_off	732	34h	4h	Fg: Bg:	File offset of map list
uint string_ids_size	20	38h	4h	Fg: Bg:	Count of strings in the string ID list
uint string_ids_off	112	3Ch	4h	Fg: Bg:	File offset of string ID list
uint type_ids_size	8	40h	4h	Fg: Bg:	Count of types in the type ID list
uint type_ids_off	192	44h	4h	Fg: Bg:	File offset of type ID list
uint proto_ids_size	3	48h	4h	Fg: Bg:	Count of items in the method prototype ID list
uint proto_ids_off	224	4Ch	4h	Fg: Bg:	File offset of method prototype ID list
uint field_ids_size	2	50h	4h	Fg: Bg:	Count of items in the field ID list
uint field_ids_off	260	54h	4h	Fg: Bg:	File offset of field ID list
uint method_ids_size	5	58h	4h	Fg: Bg:	Count of items in the method ID list
uint method_ids_off	276	5Ch	4h	Fg: Bg:	File offset of method ID list
uint class_defs_size	1	60h	4h	Fg: Bg:	Count of items in the class definitions list
uint class_defs_off	316	64h	4h	Fg: Bg:	File offset of class definitions list
uint data_size	544	68h	4h	Fg: Bg:	Size of data section in bytes
uint data_off	348	6Ch	4h	Fg: Bg:	File offset of data section
struct string_id_list dex_string_ids	20 strings	70h	50h	Fg: Bg:	String ID list
struct type_id_list dex_type_ids	8 types	C0h	20h	Fg: Bg:	Type ID list
struct proto_id_list dex_proto_ids	3 prototypes	E0h	24h	Fg: Bg:	Method prototype ID list
struct field_id_list dex_field_ids	2 fields	104h	10h	Fg: Bg:	Field ID list
struct field_id_item field_id[0]	int TestMain.mX	104h	8h	Fg: Bg:	Field ID
struct field_id_item field_id[1]	java.io.PrintStream java.lang.System.out	10Ch	8h	Fg: Bg:	Field ID
struct method_id_list dex_method_ids	5 methods	114h	28h	Fg: Bg:	Method ID list
struct method_id_item method_id[0]	void TestMain.<init>()	114h	8h	Fg: Bg:	Method ID
struct method_id_item method_id[1]	void TestMain.main(java.lang.String[])	11Ch	8h	Fg: Bg:	Method ID
struct method_id_item method_id[2]	void TestMain.test()	124h	8h	Fg: Bg:	Method ID
struct method_id_item method_id[3]	void java.io.PrintStream.println(java.lang.String)	12Ch	8h	Fg: Bg:	Method ID
struct method_id_item method_id[4]	void java.lang.Object.<init>()	134h	8h	Fg: Bg:	Method ID
struct class_def_item_list dex_class_de…	1 classes	13Ch	20h	Fg: Bg:	Class definitions list
struct class_def_item class_def	public TestMain	13Ch	20h	Fg: Bg:	Class ID
struct map_list_type dex_map_list	13 items	2DCh	A0h	Fg: Bg:	Map list
uint size	13	2DCh	4h	Fg: Bg:	
struct map_item list[13]		2E0h	9Ch	Fg: Bg:	

图7 010Editor检测TemplateResult结果

通过dexdump 命令查看（注意）

利用build-tools 下的dexdump 命令查看，`dexdump -d -l plain TestMain.dex`，得到下面的结果

```
F:\>dexdump -d -l plain TestMain.dex
Processing 'TestMain.dex'...
Opened 'TestMain.dex', DEX version '035'
Class #0            -
  Class descriptor  : 'LTestMain;'
  Access flags      : 0x0001 (PUBLIC)
  Superclass        : 'Ljava/lang/Object;'
  Interfaces        -
  Static fields     -
  Instance fields   -
    #0              : (in LTestMain;)
      name          : 'mX'
      type          : 'I'
      access        : 0x0001 (PUBLIC)
  Direct methods    -
    #0              : (in LTestMain;)
      name          : '<init>'
      type          : '()V'
      access        : 0x10001 (PUBLIC CONSTRUCTOR)
      code          -
      registers     : 2
      ins           : 1
      outs          : 1
      insns size    : 7 16-bit code units
00015c:                                        |[00015c] TestMain.<init>:()V
00016c: 7010 0400 0100                         |0000: invoke-direct {v1},
Ljava/lang/Object;.<init>:()V // method@0004
000172: 1200                                   |0003: const/4 v0, #int 0 // #0
000174: 5910 0000                              |0004: iput v0, v1,
LTestMain;.mX:I // field@0000
000178: 0e00                                   |0006: return-void
      catches       : (none)
```

```
      positions     :
        0x0000 line=11
        0x0003 line=3
        0x0006 line=12
      locals        :
        0x0000 - 0x0007 reg=1 this LTestMain;

    #1              : (in LTestMain;)
      name          : 'main'
      type          : '([Ljava/lang/String;)V'
      access        : 0x0009 (PUBLIC STATIC)
      code          -
      registers     : 4
      ins           : 1
      outs          : 2
      insns size    : 16 16-bit code units
00017c:                                        |[00017c] TestMain.main:
([Ljava/lang/String;)V
00018c: 2200 0100                              |0000: new-instance v0, LTestMain;
// type@0001
000190: 7010 0000 0000                         |0002: invoke-direct {v0},
LTestMain;.<init>:()V // method@0000
000196: 6e10 0200 0000                         |0005: invoke-virtual {v0},
LTestMain;.test:()V // method@0002
00019c: 6201 0100                              |0008: sget-object v1,
Ljava/lang/System;.out:Ljava/io/PrintStream; // field@0001
0001a0: 1a02 0100                              |000a: const-string v2, "Hello
World!" // string@0001
0001a4: 6e20 0300 2100                         |000c: invoke-virtual {v1, v2},
Ljava/io/PrintStream;.println:(Ljava/lang/String;)V // method@0003
0001aa: 0e00                                   |000f: return-void
      catches       : (none)
      positions     :
        0x0000 line=6
        0x0005 line=7
        0x0008 line=8
        0x000f line=9
      locals        :
        0x0005 - 0x0010 reg=0 testMainObject LTestMain;
        0x0000 - 0x0010 reg=3 args [Ljava/lang/String;

  Virtual methods   -
    #0              : (in LTestMain;)
      name          : 'test'
      type          : '()V'
      access        : 0x0001 (PUBLIC)
      code          -
      registers     : 1
      ins           : 1
      outs          : 0
      insns size    : 1 16-bit code units
0001ac:                                        |[0001ac] TestMain.test:()V
0001bc: 0e00                                   |0000: return-void
      catches       : (none)
      positions     :
        0x0000 line=15
      locals        :
        0x0000 - 0x0001 reg=0 this LTestMain;
```

```
  source_file_idx   : 8 (TestMain.java)
```

- registers:Dalvik 最初目标是运行在以ARM 做CPU 的机器上的，ARM 芯片的一个主要特点是寄存器多。寄存器多的话有好处，就是可以把操作数放在寄存器里，而不是像传统VM 一样放在栈中。自然，操作寄存器是比操作内存（栈嘛，其实就是一块内存区域）快。registers 变量表示该方法运行过程中会使用多少个寄存器。
- ins：输入参数对应的个数
- outs：此函数内部调用其他函数，需要的参数个数。
- insns size：以4 字节为单位，代表该函数字节码的长度（类似Class 文件的code[]数组）

更多内容参考：官网介绍----->Dalvik 可执行文件格式

dex文件的作用

记录整个工程中所有类的信息，记住的整个工程所有类的信息

dex文件格式详解

- 一种8位字节的二进制流文件
- 各个数据按顺序紧密的排列，无间隙
- 整个应用中所有的Java源文件都放在一个dex中
  大图这里

dex文件结构

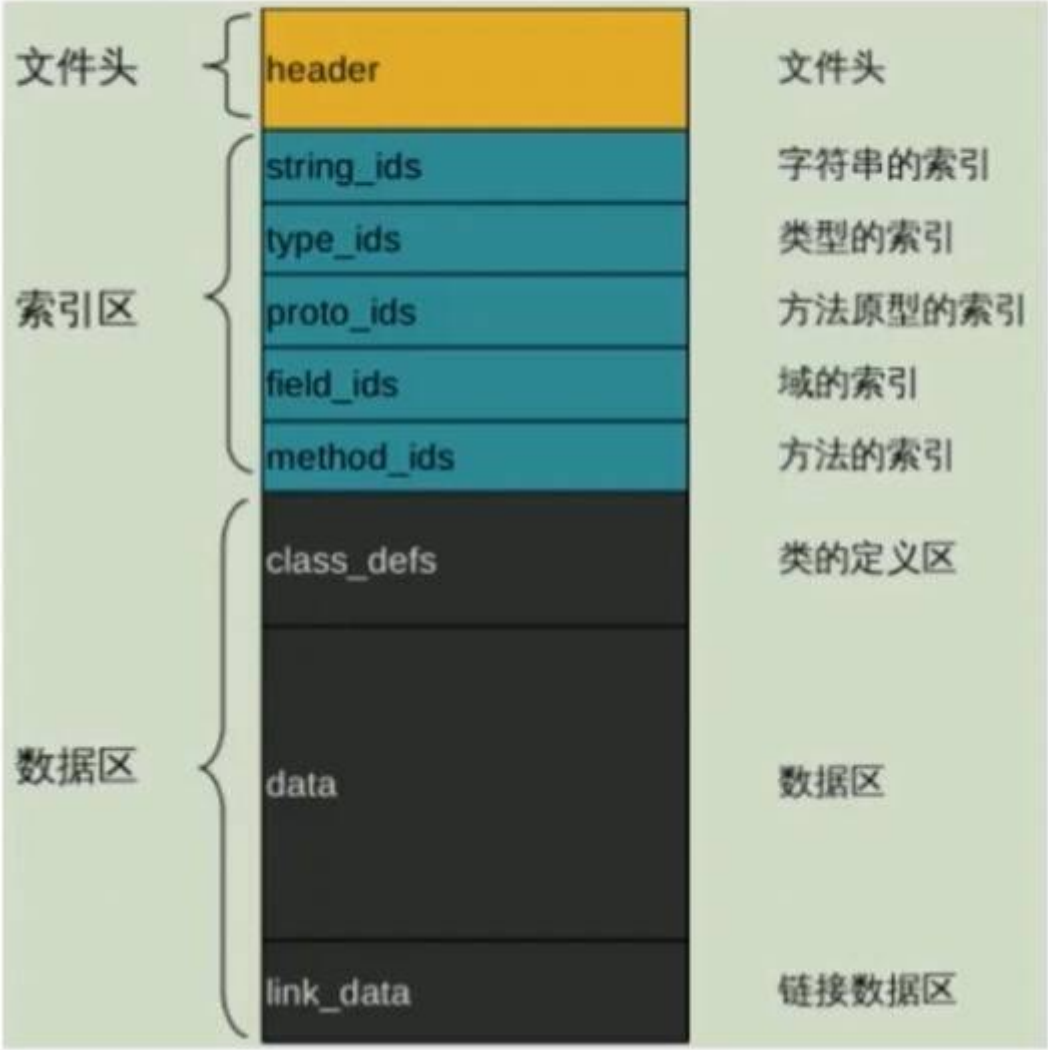


图8 dex文件结构

上图中的文件头部分，记录了dex文件的信息，所有字段大致的一个分部；索引区部分，主要包含字符串、类型、方法原型、域、方法的索引；索引区最终又被存储在数据区，其中链接数据区，主要存储动态链接库，so库的信息。

源码：/dalvik/libdex/DexFile.h:DexFile

```
struct DexFile {
    /* directly-mapped "opt" header */
    const DexOptHeader* pOptHeader;
    /* pointers to directly-mapped structs and arrays in base DEX */
    const DexHeader*    pHeader;
    const DexStringId*  pStringIds;
    const DexTypeId*    pTypeIds;
    const DexFieldId*   pFieldIds;
    const DexMethodId*  pMethodIds;
    const DexProtoId*   pProtoIds;
    const DexClassDef*  pClassDefs;
    const DexLink*      pLinkData;
};
```

具体可查看Android源码官网的关于dex文件结构的详解，如下图9（大图这里）

名称	格式	说明
header	header_item	标头
string_ids	string_id_item[]	字符串标识符列表。这些是此文件使用的所有字符串的标识符，用于内部命名（例如类型描述符）或用作代码引用的常量对象。此列表必须使用 UTF-16 代码点值按字符串内容进行排序（不采用语言区域敏感方式），且不得包含任何重复条目。
type_ids	type_id_item[]	类型标识符列表。这些是此文件引用的所有类型（类、数组或原始类型）的标识符（无论文件中是否已定义）。此列表必须按 string_id 索引进行排序，且不得包含任何重复条目。
proto_ids	proto_id_item[]	方法原型标识符列表。这些是此文件引用的所有原型的标识符。此列表必须按返回类型（按 type_id 索引排序）主要顺序进行排序，然后按参数列表（按 type_id 索引排序的各个参数，采用字典排序方法）进行排序。该列表不得包含任何重复条目。
field_ids	field_id_item[]	字段标识符列表。这些是此文件引用的所有字段的标识符（无论文件中是否已定义）。此列表必须进行排序，其中定义类型（按 type_id 索引排序）是主要顺序，字段名称（按 string_id 索引排序）是中间顺序，而类型（按 type_id 索引排序）是次要顺序。该列表不得包含任何重复条目。
method_ids	method_id_item[]	方法标识符列表。这些是此文件引用的所有方法的标识符（无论文件中是否已定义）。此列表必须进行排序，其中定义类型（按 type_id 索引排序）是主要顺序，方法名称（按 string_id 索引排序）是中间顺序，而方法原型（按 proto_id 索引排序）是次要顺序。该列表不得包含任何重复条目。
class_defs	class_def_item[]	类定义列表。这些类必须进行排序，以便所指定类的超类和已实现的接口比引用类更早出现在该列表中。此外，对于在该列表中多次出现的同名类，其定义是无效的。
call_site_ids	call_site_id_item[]	调用站点标识符列表。这些是此文件引用的所有调用站点的标识符（无论文件中是否已定义）。此列表必须按 call_site_off 的升序进行排序。
method_handles	method_handle_item[]	方法句柄列表。此文件引用的所有方法句柄的列表（无论文件中是否已定义）。此列表未进行排序，而且可能包含将在逻辑上对应于不同方法句柄实例的重复项。
data	ubyte[]	数据区，包含上面所列表格的所有支持数据。不同的项有不同的对齐要求；如有必要，则在每个项之前插入填充字节，以实现所需的对齐效果。
link_data	ubyte[]	静态链接文件中使用的数据。本文档尚未指定本区段中数据的格式。此区段在未链接文件中为空，而运行时实现可能会在适当的情况下使用这些数据。

图9 dex文件结构详解

总结：

数据名称	解释
header	dex文件头部，记录整个dex文件的相关属性
string_ids	字符串数据索引，记录了每个字符串在数据区的偏移量
type_ids	类似数据索引，记录了每个类型的字符串索引
proto_ids	原型数据索引，记录了方法声明的字符串，返回类型字符串，参数列表
field_ids	字段数据索引，记录了所属类，类型以及方法名
method_ids	类方法索引，记录方法所属类名，方法声明以及方法名等信息
class_defs	类定义数据索引，记录指定类各类信息，包括接口，超类，类数据偏移量
data	数据区，保存了各个类的真是数据
link_data	连接数据区

DEX 文件中会出现的数据类型

类型	含义
u1	等同于uint8_t，表示 1 字节的无符号 数
u2	等同于 uint16_t，表示 2 字节的无符号数
u4	等同于 uint32_t，表示 4 字节的无符号数
u8	等同于 uint64_t，表示 8 字节的无符号数
sleb128	有符号 LEB128，可变长度 1~5 字节
uleb128	无符号 LEB128，可变长度 1~5 字节
uleb128p1	无符号 LEB128 值加1，可变长 1~5 字节

/dalvik/libdex/DexFile.h中定义如下

```
typedef uint8_t             u1;
typedef uint16_t            u2;
typedef uint32_t            u4;
typedef uint64_t            u8;
typedef int8_t              s1;
typedef int16_t             s2;
typedef int32_t             s4;
typedef int64_t             s8;
```

LEB128
LEB128（“Little-Endian Base 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 位均为有效负荷，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负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。
大图这里

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	bit_7

图10 双字节 LEB128 值的按位图

变量 uleb128p1 用于表示一个有符号值，其表示法是编码为 uleb128 的值加 1。这使得编码 -1（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明确说明的情况下非常实用：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
7f	-1	127	126
80	7f	-128	16256

dex文件头
Dex文件头主要包括校验和以及其他结构的偏移地址和长度信息。
源码位于 /dalvik/libdex/DexFile.h:DexHeader

```
struct DexHeader {
    u1  magic[8];           /* includes version number */
    u4  checksum;           /* adler32 checksum */
    u1  signature[kSHA1DigestLen]; /* SHA-1 hash */
    u4  fileSize;           /* length of entire file */
    u4  headerSize;         /* offset to start of next section */
    u4  endianTag;
    u4  linkSize;
    u4  linkOff;
    u4  mapOff;
    u4  stringIdsSize;
    u4  stringIdsOff;
    u4  typeIdsSize;
    u4  typeIdsOff;
    u4  protoIdsSize;
    u4  protoIdsOff;
    u4  fieldIdsSize;
    u4  fieldIdsOff;
    u4  methodIdsSize;
    u4  methodIdsOff;
    u4  classDefsSize;
    u4  classDefsOff;
    u4  dataSize;
    u4  dataOff;
};
```

具体详解如下图5所示
大图这里

字段名称	偏移值	长度	描述
magic	0x0	8	'Magic'值，即魔数字段，格式如"dex/n035/0"，其中的035表示结构的版本。
checksum	0x8	4	校验码。
signature	0xC	20	SHA-1签名。
file_size	0x20	4	Dex文件的总长度。
header_size	0x24	4	文件头长度，009版本=0x5C,035版本=0x70。
endian_tag	0x28	4	标识字节顺序的常量,根据这个常量可以判断文件是否交换了字节顺序,缺省情况下=0x78563412。
link_size	0x2C	4	连接段的大小，如果为0就表示是静态连接。
link_off	0x30	4	连接段的开始位置，从本文件头开始算起。如果连接段的大小为0，这里也是0。
map_off	0x34	4	map数据基地址。
string_ids_size	0x38	4	字符串列表的字符串个数。
string_ids_off	0x3C	4	字符串列表表基地址。
type_ids_size	0x40	4	类型列表里类型个数。
type_ids_off	0x44	4	类型列表基地址。
proto_ids_size	0x48	4	原型列表里原型个数。
proto_ids_off	0x4C	4	原型列表基地址。
field_ids_size	0x50	4	字段列表里字段个数。
field_ids_off	0x54	4	字段列表基地址。
method_ids_size	0x58	4	方法列表里方法个数。
method_ids_off	0x5C	4	方法列表基地址。
class_defs_size	0x60	4	类定义类表中类的个数。
class_defs_off	0x64	4	类定义列表基地址。
data_size	0x68	4	数据段的大小，必须以4字节对齐。
data_off	0x6C	4	数据段基地址

图11 dex文件头信息

各个字段详解摘要
mapOff 字段
指定 DexMapList 结构距离 Dex 头的偏移
DexMapList 结构体：

```
struct     DexMapList
{
    u4     size;                                    // DexMapItem 的个数
    DexMapItem     list[1];                // DexMapItem 结构
};
```

- size：表示接下来有多少个 DexMapItem
- list：是一个 DexMapItem 结构体数组
  DexMapItem 结构体：

```
struct     DexMapItem
{
    u2     type;                     // kDexType 开头的类型
    u2     unused;                 // 未使用，用于对齐
    u4     size;                      // 指定类型的个数
    u4     offset;                  // 指定类型数据的文件偏移
}
```

type：一个枚举常量

```
```

```
enum
{
    kDexTypeHeaderItem = 0x0000,    // 对应 DexHeader
    kDexTypeStringIdItem = 0x0001,  // 对应 stringIdsSize 与 stringIdsOff 字段

    kDexTypeTypeIdItem = 0x0002,    // 对应 typeIdsSize 与 typeIdsOff 字段
    kDexTypeProtoIdItem = 0x0003,   // 对应 protoIdsSize 与 protoIdsOff 字段
    kDexTypeFieldIdItem = 0x0004,   // 对应 fieldIdsSize 与 fieldIdsOff 字段
    kDexTypeMethodIdItem = 0x0005,  // 对应 methodIdsSize 与 methodIdsOff 字段

    kDexTypeClassDefItem = 0x0006,  // 对应 classDefsSize 与 classDefsOff 字段

    kDexTypeMapList = 0x1000,
    kDexTypeTypeList = 0x1001,
    kDexTypeAnnotationSetRefList = 0x1002,
    kDexTypeAnnotationSetItem = 0x1003,
    kDexTypeClassDataItem = 0x2000,
    kDexTypeCodeItem = 0x2001,
    kDexTypeStringDataItem = 0x2002,
    kDexTypeDebugInfoItem = 0x2003,
    kDexTypeAnnotationItem = 0x2004,
    kDexTypeEncodeArrayItem = 0x2005,
    kDexTypeAnnotationsDirectoryItem = 0x2006
};
```

- size：指定类型的个数
- offset：指定类型数据的偏移

DexStringId 结构体（stringIdsSize 与 stringIdsOff 字段）

```
typedef struct _DexStringId
{
    u4  stringDataOff;                  // 指向 MUTF-8 字符串的偏移
}DexStringId, *PDexStringId;
```

MUTF-8 编码：

1. 使用 1~3 字节编码长度
2. 大于 16 位的 Unicode 编码 U+10000~U+10FFFF 使用 3 字节来编码
3. U+0000 采用 2 字节编码
4. 采用空字符 null 作为结尾
5. 第一个字节存放字节个数（不包含自己）

DexTypeId 结构体（typeIdsSize 与 typeIdsOff 字段）
是一个类型结构体

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

- descriptorIdx：指向 DexStringId 列表的索引，它对应的字符串代表了具体类的类型

DexProtoId 结构体（protoIdsSize 与 protoIdsOff 字段）
是一个方法声明结构体，方法声明 = 返回类型 + 参数列表

```
typedef struct _DexProtoId
{
    u4  shortyIdx;          // 方法声明字符串，指向 DexStringId 列表的索引
    u4  returnTypeIdx;      // 方法返回类型字符串，指向 DexStringId 列表的索引
    u4  parametersOff;      // 方法的参数列表，指向 DexTypeList 结构体的偏移
}DexProtoId, *PDexProtoId;
```

- shortyIdx：方法声明字符串，方法声明 = 返回类型 + 参数列表
- returnTypeIdx：方法返回类型字符串
- parametersOff：指向一个 DexTypeList 结构体，存放了方法的参数列表

DexTypeList 结构体：

```
typedef struct _DexTypeList
{
    u4  size;               // 接下来 DexTypeItem 的个数
    DexTypeItem* list;      // DexTypeItem 结构
}DexTypeList, *PDexTypeList;
```

- size：接下来 DexTypeItem 的个数
- list：是一个 DexTypeItem 结构体数组

DexTypeItem 结构体：

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

typeIdx：DexTypeId 列表的索引

DexFieldId 结构体（fieldIdsSize 与 fieldIdsOff 字段）
指明了字段所有的类、字段的类型以及字段名

```
typedef struct _DexFieldId
{
    u2  classIdx;           // 类的类型，指向 DexTypeId 列表的索引
    u2  typeIdx;            // 字段的类型，指向 DexTypeId 列表的索引
    u4  nameIdx;            // 字段名，指向 DexStringId 列表的索引
}DexFieldId, *PDexFieldId;
```

- classIdx：类的类型
- typeIdx：字段的类型
- nameIdx：字段名

DexMethodId 结构体（methodIdsSize 与 methodIdsOff 字段）
方法结构体

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

- classIdx：类的类型
- protoIdx：声明的类型
- nameIdx：方法名

DexClassDef 结构体（classDefsSize 和 classDefsOff 字段）
类结构体

```
typedef struct _DexClassDef
{
    u4     classIdx;                // 类的类型，指向 DexTypeId 列表的索引
    u4     accessFlags;             // 访问标志
    u4     superclassIdx;           // 父类类型，指向 DexTypeId 列表的索引
    u4     interfacesOff;           // 接口，指向 DexTypeList 的偏移，否则为0
    u4     sourceFileIdx;           // 源文件名，指向 DexStringId 列表的索引
    u4     annotationsOff;          // 注解，指向 DexAnnotationsDirectoryItem 结
构，或者为 0
    u4     classDataOff;            // 指向 DexClassData 结构的偏移，类的数据部分
    u4     staticValuesOff;         // 指向 DexEncodedArray 结构的偏移，记录了类中的
静态数据，主要是静态方法
}DexClassDef, *PDexClassDef;
```

- classIdx：类的类型，指向 DexTypeId 列表的索引
- accessFlags：访问标志，它是以ACC_开头的枚举值
- superclassIdx：父类类型，指向 DexTypeId 列表的索引
- interfacesOff：接口，指向 DexTypeList 的偏移，如果没有，则为 0
- sourceFileIdx：源文件名，指向 DexStringId 列表的索引
- annotationsOff：注解，指向 DexAnnotationsDirectoryItem 结构，或者为 0
- classDataOff：指向 DexClassData 结构的偏移，类的数据部分
- staticValuesOff：指向 DexEncodeArray 结构的偏移，记录了类中的静态数据，没有则为 0

DexClassData 结构体：

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

- header：DexClassDataHeader 结构体，指定字段与方法的个数
- staticFields：静态字段，DexField 结构体数组
- instanceFields：实例字段，DexField 结构体数组
- directMethods：直接方法，DexMthod 结构体数组

- virtualMethods：虚方法，DexMethod 结构体数组

DexClassDataHeader 结构体：

```
typedef struct _DexClassDataHeader
{
    uleb128 staticFieldsSize;                   // 静态字段个数
    uleb128 instanceFieldsSize;                 // 实例字段个数
    uleb128 directMethodsSize;                  // 直接方法个数
    uleb128 virtualMethodsSize;                 // 虚方法个数
}DexClassDataHeader, *PDexClassDataHeader;
```

- staticFieldsSize：静态字段个数
- instanceFieldsSize：实例字段个数
- directMethodsSize：直接方法个数
- virtualMethodsSize：虚方法个数

DexField 结构体：

```
typedef struct _DexField
{
    uleb128 fieldIdx;           // 指向 DexFieldId 的索引
    uleb128 accessFlags;        // 访问标志
}DexField, *PDexField;
```

- fieldIdx：字段描述，指向 DexFieldId 的索引
- accessFlags：访问标志

DexMethod 结构体：

```
typedef struct _DexMethod
{
    uleb128 methodIdx;          // 指向 DexMethodId 的索引
    uleb128 accessFlags;        // 访问标志
    uleb128 codeOff;            // 指向 DexCode 结构的偏移
}DexMethod, *PDexMethod;
```

- methodIdx：方法描述，指向 DexMethodId 的索引
- accessFlags：访问标志
- codeOff：指向 DexCode 结构的偏移

DexCode 结构体：

```
typedef struct _DexCode
{
    u2  registersSize;          // 使用的寄存器个数
    u2  insSize;                // 参数个数
    u2  outsSize;               // 调用其他方法时使用的寄存器个数
    u2  triesSize;              // Try/Catch 个数
    u4  debbugInfoOff;          // 指向调试信息的偏移
    u4  insnsSize;              // 指令集个数，以 2 字节为单位
    u2* insns;                  // 指令集
}DexCode, *PDexCode;
```

还有一些不太常见的结构体，要用的时候再去看看就行了。Dex 文件的整体结构就这样，就是一个多层索引的结构。

string_ids（字符串索引）
这一区域存储的是Dex文件字符串资源的索引信息，该索引信息是目标字符串在Dex文件数据区所在的真实物理偏移量。

源码位于 /dalvik/libdex/DexFile.h:DexStringId

```
struct DexStringId {

    u4 stringDataOff;      /* file offset to string_data_item */
};
```

stringDataOff记录了目标字符串在Dex文件中的实际偏移量，虚拟机想读取该字符串时，只需将Dex文件在内存中的起始地址加上stringDataOff所指的偏移量，就是该字符串在内存中的实际物理地址。
在Dex文件中，每个每个字符串对应一个DexStringId,大小4B。另外虚拟机通过DexHeader中的String_ids_size获得当前Dex文件中的字符串的总数,通过乘法就可对该索引资源进行访问。

DexLink

```

struct DexLink {
    u1  bleargh;
};
```

DexFile 在内存中的映射

在Android系统中， java 源文件会被编译为“ .jar ” 格式的dex类型文件， 在代码中称为dexfile 。在加载 Class 之前， 必先读取相应的jar文件。通常我们使用read（）函数来读取文件中的内容。但在Dalvik中使用mmap（） 函数。和read（）不同， mmap（）函数会将dex文件映射到内存中，这样通过普通的内存读取操作即可访问dexfile中的内容。

Dexfile的文件格式如图12 所示， 主要有三部分组成：头部，索引，数据。通过头部可知索引的位置和数同，可知数据区的起始位置。其中classDefsOff 指定了ClassDef 在文件的起始位置， dataOff 指定了数据在文件的起始位置， ClassDef 即可理解为Class 的索引。通过读取ClassDef 可获知Class 的基本信息，其中classDataOff 指定了Class 数据在数据区的位置。

大图这里

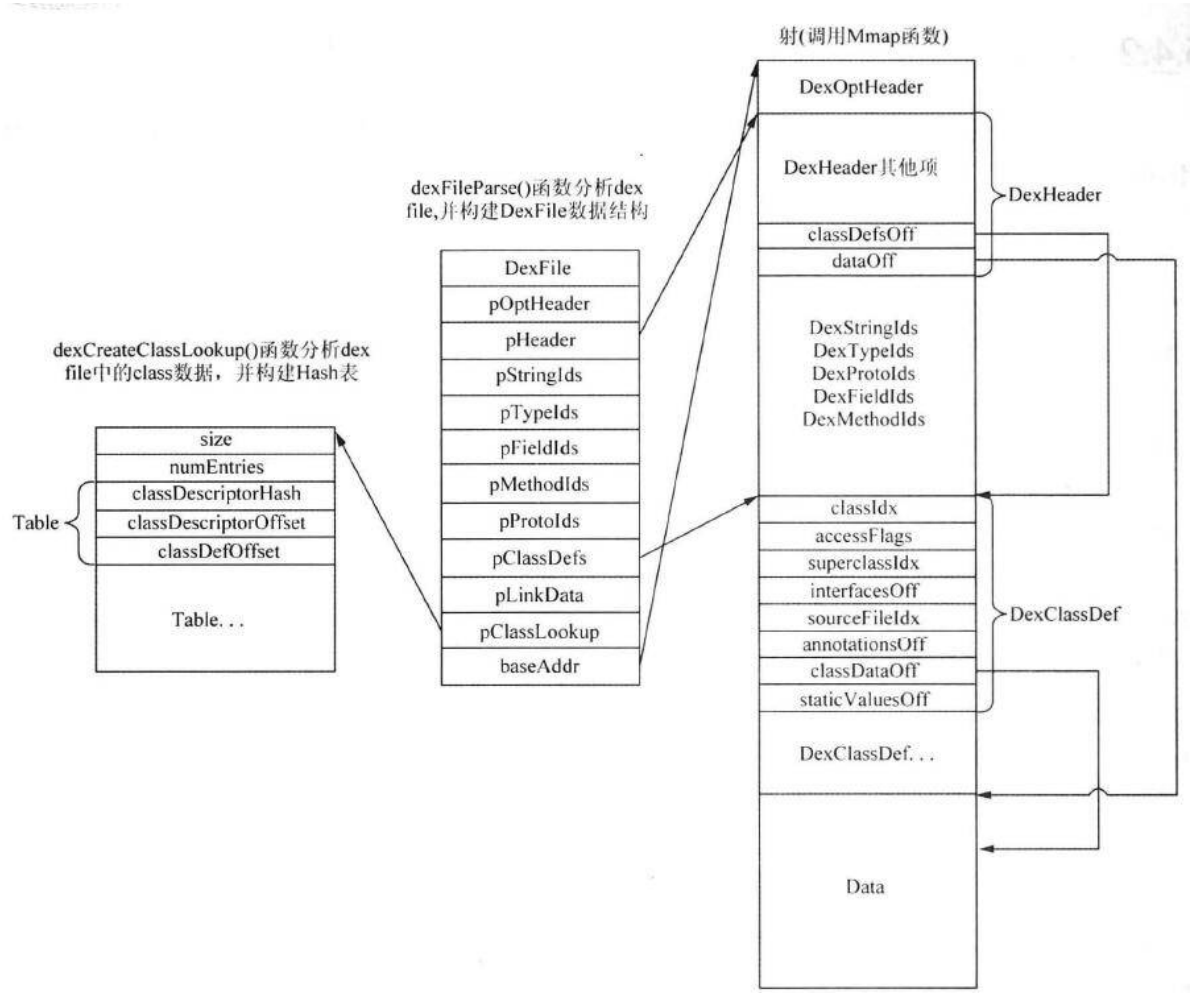


图12 DexFile的文件格式

在将dexfile文件映射到内存后，会调用dexFileParse（）函数对其分析，分析的结果存放于名为DexFile的数据结构中。DexFile 中的baseAddr指向映射区的起始位置， pClassDefs 指向ClassDefs（即class索引）的起始位置。由于在查找class 时，都是使用class的名字进行查找的，所以为了加快查找速度， 创建了一个hash表。在hash表中对class 名字进行hash，并生成index。这些操作都是在对文件解析时所完成的，这样虽然在加载过程中比较耗时，但是在运行过程中可节省大量查找时间。

解析完后， 接下来开始加载class文件。在此需要将加载类用ClassObject来保存，所以在此需要先分析和ClassObject 相关的几个数据结构。

首先在文件Object.h 中可以看到如下对结构体Object 的定义。（android2.3.7源码）

```
typedef struct Object {
    /* ptr to class object */
    ClassObject*    clazz;

    /*
     * A word containing either a "thin" lock or a "fat" monitor.  See
     * the comments in Sync.c for a description of its layout.
     */
    u4              lock;
} Object;
```

通过结构体Object定义了基本类的实现，这里有如下两个变量。

- lock： 对应Obejct 对象中的锁实现，即notify wait 的处理。

- clazz： 是结构体指针，姑且不看结构体内容，这里用了指针的定义。

下面会有更多的结构体定义：

```
struct DataObject {
    Object          obj;                /* MUST be first item */
    /* variable #of u4 slots; u8 uses 2 slots */
    u4              instanceData[1];
};
struct StringObject {
    Object          obj;                /* MUST be first item */
    /* variable #of u4 slots; u8 uses 2 slots */
    u4              instanceData[1];
};
```

我们看到最熟悉的一个词StringObject ，把这个结构体展开后是下面的样子。

```

struct StringObject {
    /* ptr t o class object */
    ClassObject*  clazz ;
    /* variable #of u4 slots; u8 uses 2 slots */
    u4   lock;
    u4              instanceData[1];
};
```

由此不难发现， 任何对象的内存结构体中第一行都是Object结构体，而这个结构体第一个总是一个ClassObejct，第二个总是lock 。按照C++中的技巧，这些结构体可以当成Object结构体使用，因此所有的类在内存中都具有"对象"的功能，即可以找到一个类（ClassObject），可以有一个锁(lock） 。

StringObject是对String类进行管理的数据对象，ArrayObejct是数据相关的管理。

ClassObject-Class 在加载后的表现形式

在解析完文件后， 接下来需要加载Class 的具体内容。在Dalvik中， 由数据结构ClassObject负责存放加载的信息。如图13所示，加载过程会在内存中alloc几个区域，分别存放directMethods 、virtualMethods 、sfields 、ifields 。这些信息是从dex 文件的数据区中读取的，首先会读取Class 的详细信息，从中获得directMethod 、virtua!Method 、sfield 、ifield 等的信息，然后再读取。在此需要注意， 在C lassObj ect 结构中有个名为super 的成员，通过super成员可以指向它的超类。
大图这里

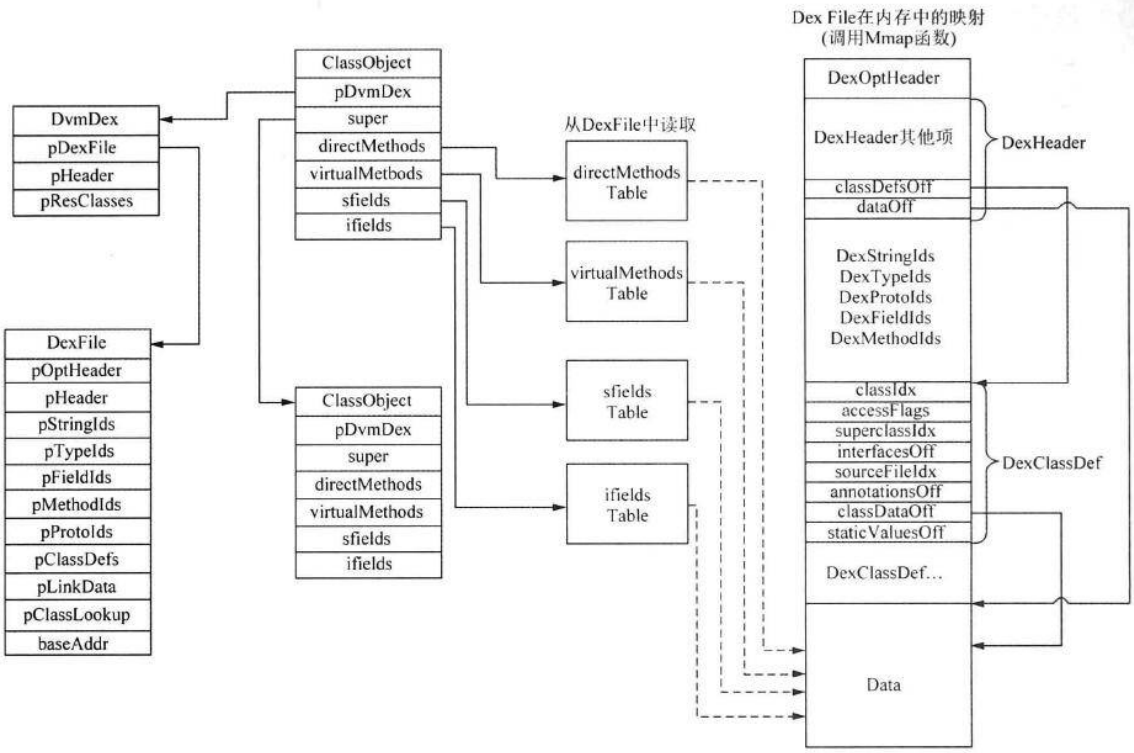


图13 加载过程

Android dex 文件优化

对Android dex 文件进行优化来说， 需要注意的一点是dex文件的结构是紧凑的，但是我们还是要想方设法地进行提高程序的运行速度，我们就仍然需要对dex文件进行进一步优化。

调整所有字段的字节序（ LITTLE_ENDIAN），和对齐结构中的每一个域来验证dex文件中的所有类，并对一些特定的类进行优化或对方法里的操作码进行优化。优化后的文件大小会有所增加， 大约是原Android dex文件的1～4 倍。

优化时机
优化发生的时机有两个：

- 对于预置应用来说，可以在系统编译后，生成优化文件，以ODEX 结尾。这样在发布时除APK文件（不包含dex）以外，还有一个相应的Android dex 文件。
- 对于非预置应用， 包含在APK文件里的dex 文件会在运行时被优化，优化后的文件将被保存在缓存中。

如下图14所示代码调用流程

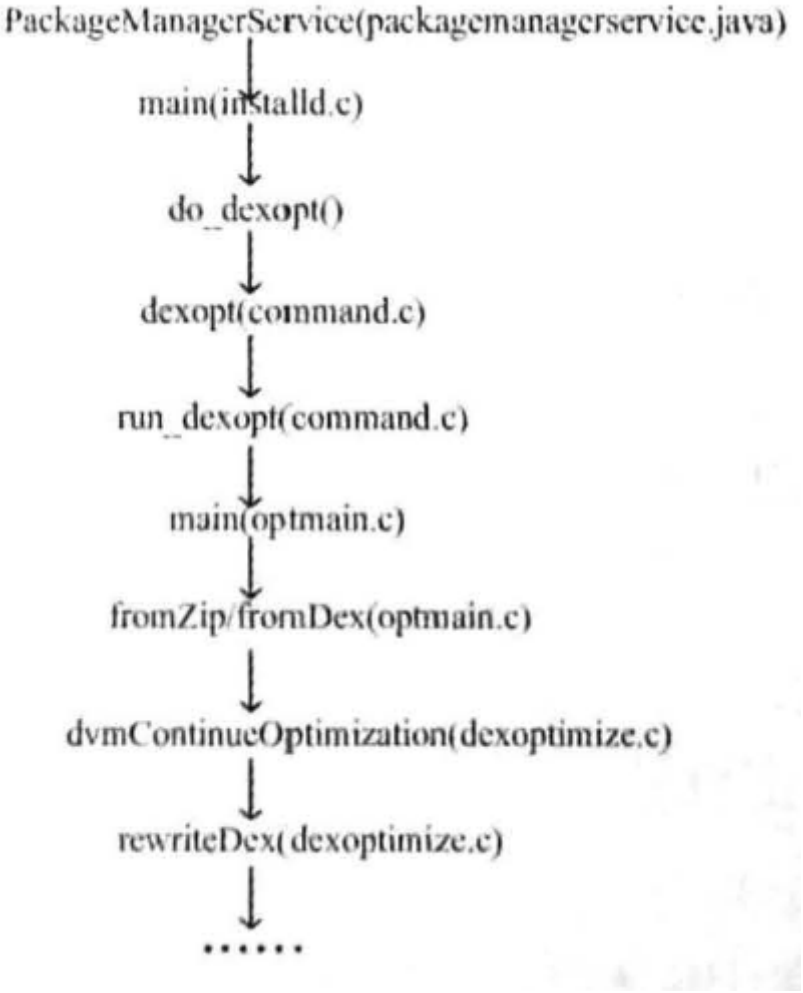


图14 代码调用流程

每一个Android应用都运行在一个Dalvik虚拟机实例里，而每一个虚拟机实例都是一个独立的进程空间。虚拟机的线程机制，内存分配和管理， Mutex等都是依赖底层操作系统而实现的。

所有Android应用的线程都对应一个Linux线程（可参考----理解Android线程创建流程），虚拟机因而可以更多地依赖操作系统的线程调度和管理机制。不同的应用在不同的进程空间里运行，加之对不同来源的应用都使用不同的Linux用户来运行，可以最大限度地保护应用的安全和独立运行。

Zygote是一个虚拟机进程，同时也是一个虚拟机实例的孵化器，每当系统要求执行一个Android应用程序，Zygote就会孵化出一个子进程来执行该应用程序。这样做的好处显而易见：Zygote进程是在系统启动时产生的，它会完成虚拟机的初始化，库的载，预置类库的加载和初始化等操作，而在系统需要一个新的虚拟机实例时，Zygote通过复制自身，最快速地提供一个虚拟机实例。另外，对于一些只读的系统库，所有虚拟机实例都和Zygote 共享一块内存区域，大大节省了内存开销。

Android 应用所使用的编程语言是Java语言，和Java SE 一样，编译时使用Oracle JDK 将Java源程序编程成标准的Java 字节码文件（. class 文件）。而后通过工具软件DX 把所有的字节码文件转成Android dex 文件（classes . dex） 。最后使用Android 打包工具（aapt）将dex 文件、资源文件以及AndroidManifest.xml 文件（二进制格式）组合成一个应用程序包（APK） 。应用程序包可以被发布到手机上运行。

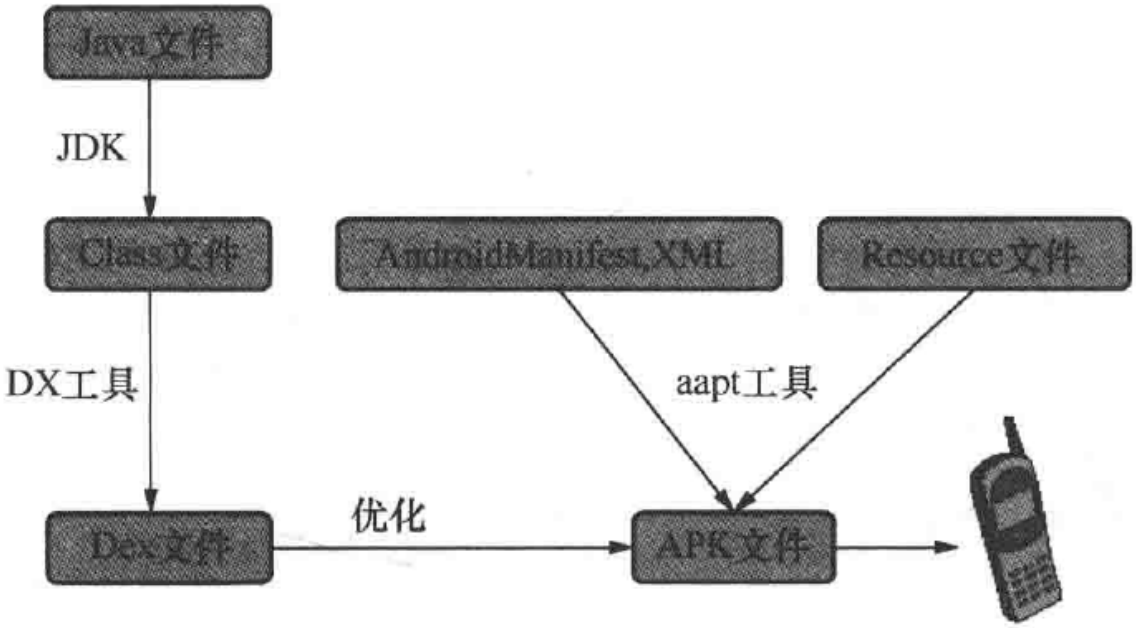


图15 Android应用编译及运行流程

odex 介绍

odex 是Optimized dex 的简写，也就是优化后的dex 文件。为什么要优化呢？主要还是为了提高Dalvik 虚拟机的运行速度。但是odex 不是简单的、通用的优化，而是在其优化过程中，依赖系统已经编译好的其他模块，简单点说：

- 从Class 文件到dex 文件是针对Android 平台的一种优化，是一种通用的优化。优化过程中，唯一的输入是Class 文件。
- odex 文件就是dex 文件具体在某个系统（不同手机，不同手机的OS，不同版本的OS 等）上的优化。odex 文件的优化依赖系统上的几个核心模块（ 由BOOTCLASSPATH 环境变量给出， 一般是/system/framework/下的jar 包，尤其是core.jar）。odex 的优化就好像是把中那些本来需要在执行过程中做的类校验、调用其他类函数时的解析等工作给提前处理了。

通过利用dexopt得到test.odex，接着利用dexdump得到其内容，最后可以利用Beyond Compare比较这两个文件的差异。
如下图所示

图16 test.dex 和test.odex 差异

图16中，绿色框中是test.dex的内容，红色框中是test.odex的内容，这也是两个文件的差异内容：

- test.dex中，TestMain类仅仅是PUBLIC的，但test.odex则增加了VERIFIED和OPTIMIZED两项。VERIFIED是表示该类被校验过了，至于校验什么东西，以后再说。
- 然后就是一些方法的不同了。优化后的odex文件，一些字节码指令变成了xxx-quick。比如图中最后一句代码对于的字节码中，未优化前invoke-virtual指令表示从method table指定项（图中是0002）里找到目标函数，而优化后的odex使用了invoke-virtual-quick表示从vtable中找到目标函数（图中是000b）。

vtable是虚表的意思，一般在OOP实现中用得很多。vtable一定比methodtable快么？那倒是有可能。我个人猜测：

- method表应该是每个dex文件独有的，即它是基于dex文件的。
- 根据odex文件的生成方法（后面会讲），我觉得vtable恐怕是把dex文件及依赖的类（比如Java基础类，如Object类等）放一起进行了处理，最终得到一张大的vtable。这个odex文件依赖的一些函数都放在vtable中。运行时直接调用指定位置的函数就好，不需要再解析了。以上仅是我的猜测。

> 注意：
> odex文件由dexopt生成，这个工具在SDK里没有，只能由源码生成。odex文件的生成有三种方式：
>
> - preopt：即OEM厂商（比如手机厂商），在制作镜像的时候，就把那些需要放到镜像文件里的jar包，APK等预先生成对应的odex文件，然后再把classes.dex文件从jar包和APK中去掉以节省文件体积。
> - installd：当一个apk安装的时候，PackageManagerService会调用installd的服务，将apk中的class.dex进行处理。当然，这种情况下，APK中的class.dex不会被剔除。
> - dalvik VM：preopt是厂商的行为，可做可不做。如果没有做的话，dalvik VM在加载一个dex文件的时候，会先生成odex。所以，dalvik VM实际上用得是odex文件。以后我们研究dalvik VM的时候会看到这部分内容。
>
> 实际上dex转odex是利用了dalvik vm，里边也会运行dalvik vm的相关方法。

总结：

1. 以标准角度来看，Class文件是由Java VM规范定义的，所以通用性更广。dex或者是odex只不过是规范在Android平台上的一种具体实现罢了，而且dex/odex在很多地方也需要遵守规范。因为dex文件的来源其实还是Class文件。
2. 对于初学者而言，我建议了解Class文件的结构为主。另外，关于dex/odex的文件结构，除非有明确需求（比如要自己修改字节码等），否则以了解原理就可以。而且，将来我们看到dalvik vm的实际代码后，你会发现dex的文件内容还是会转换成代码里的那些你很熟悉的类型，数据结构。比如dex存储字符串是一种优化后的方法，但是到vm代码中，还不是只能用字符串来表示吗？
3. 另外，你还会发现，Class、dex还是odex文件都存储了很多源码中的信息，比如类名、函数名、参数信息、成员变量信息等，而且直接用得是字符串。这和Native的二进制比起来，就容易看懂多了。