# Java虚拟机：类加载机制

**关于项目**

[**Android-**](https://github.com/lichenyu10/Android-)项目旨在通过提供一系列的工具与方法，降低阅读Android系统源码的门槛，让更多的Android工程师理解Android系统，掌握Android系统。

**文章目录**

* 一 类文件结构
* 二 类加载机制
* 三 类加载器

这篇文章我们来聊一聊关于Android虚拟机的那些事，当然这里我们并不需要去讲解关于虚拟机的底层细节，所讲的东西都是大家平常在开发中经常用的。例如类的加载机制、资源加载机制、APK打包流程、APK安装流程 以及Apk启动流程等。讲解这些知识是为了后续的文章《大型Android项目的工程化实践：插件化》、《大型Android项目的工程化实践：热更新》、《大型Android项目的工程化实践：模块化》等系列的文章做一个 原理铺垫。

好了，让我们开始吧~😁

## 一 类文件基本结构

Class文件是一组以8位字节为基础的单位的二进制流，各个数据项按严格的顺序紧密的排列在Class文件中，中间没有任何间隔。

这么说有点抽象，我们先来举一个简单的小例子。🤞

public class TestClass {

 public int sum(int a, int b) {
 return a + b;
 }
}

编译生成Class文件，然后使用hexdump命令查看Class文件里的内容。

javac TestClass.java
hexdump TestClass.class

Class文件内容如下所示：

Classfile /Users/guoxiaoxing/Github-app/android-open-source-project-analysis/demo/src/main/java/com/guoxiaoxing/android/framework/demo/native\_framwork/vm/TestClass.class
 Last modified 2018-1-23; size 333 bytes
 MD5 checksum 72ae3ff578aa0f97b9351522005ec274
 Compiled from "TestClass.java"
public class com.guoxiaoxing.android.framework.demo.native\_framwork.vm.TestClass
 minor version: 0
 major version: 52
 flags: ACC\_PUBLIC, ACC\_SUPER
Constant pool:
 #1 = Methodref #4.#15 // java/lang/Object."<init>":()V
 #2 = Fieldref #3.#16 // com/guoxiaoxing/android/framework/demo/native\_framwork/vm/TestClass.m:I
 #3 = Class #17 // com/guoxiaoxing/android/framework/demo/native\_framwork/vm/TestClass
 #4 = Class #18 // java/lang/Object
 #5 = Utf8 m
 #6 = Utf8 I
 #7 = Utf8 <init>
 #8 = Utf8 ()V
 #9 = Utf8 Code
 #10 = Utf8 LineNumberTable
 #11 = Utf8 inc
 #12 = Utf8 ()I
 #13 = Utf8 SourceFile
 #14 = Utf8 TestClass.java
 #15 = NameAndType #7:#8 // "<init>":()V
 #16 = NameAndType #5:#6 // m:I
 #17 = Utf8 com/guoxiaoxing/android/framework/demo/native\_framwork/vm/TestClass
 #18 = Utf8 java/lang/Object
{
 public com.guoxiaoxing.android.framework.demo.native\_framwork.vm.TestClass();
 descriptor: ()V
 flags: ACC\_PUBLIC
 Code:
 stack=1, locals=1, args\_size=1
 0: aload\_0
 1: invokespecial #1 // Method java/lang/Object."<init>":()V
 4: return
 LineNumberTable:
 line 10: 0

 public int inc();
 descriptor: ()I
 flags: ACC\_PUBLIC
 Code:
 stack=2, locals=1, args\_size=1
 0: aload\_0
 1: getfield #2 // Field m:I
 4: iconst\_1
 5: iadd
 6: ireturn
 LineNumberTable:
 line 15: 0
}
SourceFile: "TestClass.java"

Class文件十六机制内容如下所示：

注：笔者用的二进制查看软件是iHex，可以去AppStore下载，Windows用户可以使用WinHex。

这是一份十六进制表示的二进制流，每个位排列紧密，都有其对应的含义，具体说来，如下所示：

注：下列表中四个段分别为 类型、名称、说明、数量

* u4 magic 识别Class文件格式，具体值为0xCAFEBABE 1
* u2 minor\_version Class文件格式副版本号 1
* u2 major\_version Class文件格式主版本号 1
* u2 constant\_pool\_count 常数表项个数 1
* cp\_info constant\_pool 常数表，又称变长符号表 constant\_pool\_count-1
* u2 access\_flags Class的声明中使用的修改符掩码 1
* u2 this\_class 常数表索引，索引内保存类名或接口名 1
* u2 super\_class 常数表索引，索引内保存父类名 1
* u2 interfaces\_count 超接口个数 1
* u2 interfaces 常数表索引，各超接口名称 interfaces\_count
* u2 fields\_count 类的域个数 1
* field\_info fields 域数据，包括属性名称索引 fields\_count
* u2 methods\_count 方法个数 1
* method\_info methods 方法数据，包括方法名称索引 methods\_count
* u2 attributes\_count 类附加属性个数 1
* attribute\_info attributes 类附加属性数据，包括源文件名称等 attributs\_count

我们可以看着在上面这张表中有类似u2、attribute\_info这样的类型，事实上Class文件采用一种类似于C语言结构体的伪结构struct来存储数据，这种结构有两种数据类型：

* 无符号数：基本数据类型，例如u1代表1个字节，u2代表2个字节，u4代表2个字节，u8代表8个字节。
* 表：由多个无符号数或者其他表作为数据项而构成的复合数据结构，用于描述有层次关系的复合数据结构，一般以"\_info"结尾。

我们分别来看看上述的各个字段的具体含义已经对应数值。

注：这一块的内容可能有点枯燥，但是它是我们后续学习类加载机制，Android打包机制，以及学习插件化、热更新框架的基础，所以需要掌握。 但是也没必要都记住每个段的含义，你只需要有个整体性的认识即可，后续如果忘了具体的内容，可以再回来查阅。😁

### 1.1 魔数

具体含义

魔数：1-4字节，用来确定这个文件是否为一个能被虚拟机接受的Class文件，它的值为0xCAFEBABE。

对应数值

ca fe ba be

### 1.2 版本号

具体含义

版本号：5-6字节是次版本号，7-8字节是主版本号

对应数值

5-6字节是次版本号0x0000（即0），7-8字节是主版本号0x0034（即52）.

JDK版本号与数值的对应关系如下所示：

* JDK 1.8 = 52
* JDK 1.7 = 51
* JDK 1.6 = 50
* JDK 1.5 = 49
* JDK 1.4 = 48
* JDK 1.3 = 47
* JDK 1.2 = 46
* JDK 1.1 = 45

### 1.3 常量池计数/常量池

具体含义

常量池计数：常量池中常量的数量不是固定的，因此常量池入口处会放置一项u2类型的数据，代表常量池容器计数。注意容器计数从1开始，索引为0代表不引用任何一个 常量池的项目。

对应数值

9-10字节是常量池容器计数0x0013（即19）。说明常量池里有18个常量，从1-18.

这是我们上面用javap分析的字节码文件里的常量池里常量的个数是一直的。

举个常量池里的常量的例子🤞

它的常量值如下所示：

#17 = Utf8 com/guoxiaoxing/android/framework/demo/native\_framwork/vm/TestClass

常量池主要存放字面量与符号引用。

字面量包括：

* 文本字符串
* 声明为final的常量值等

符号引用包括：

* 类与接口的全限定名
* 字段的名称与描述符
* 方法的名称与描述符

常量池里的每个常量都用一个表来表示，表的结构如下所示：

cp\_info {
 //代表常量类型
 u1 tag;
 //代表存储的常量，不同的常量类型有不同的结构
 u1 info[];
}

目标一共有十四中常量类型，如下所示：

注：下表字段分别为 类型、标志（tag）、描述

* CONSTANT\_Utf8\_info 1 UTF8编码的Unicode字符串
* CONSTANT\_Integer\_info 3 整型字面量
* CONSTANT\_Float\_info 4 浮点型字面量
* CONSTANT\_Long\_info 5 长整型字面量
* CONSTANT\_Double\_info 6 双精度浮点型字面量
* CONSTANT\_Class\_info 7 类或接口的符号引用
* CONSTANT\_String\_info 8 字符串类型字面量
* CONSTANT\_Fieldref\_info 9 字段的符号引用
* CONSTANT\_Methodref\_info 10 类中方法的符号引用
* CONSTANT\_InterfaceMethodref\_info 11 接口中方法的符号引用
* CONSTANT\_NameAndType\_info 12 字段或方法的部分符号引用

### 1.4 访问标志

具体含义

访问标志：常量池之后就是访问标志，该标志用于识别一些类或则接口层次的访问信息。这些访问信息包括这个Class是类还是接口，是否定义Abstract类型等。

对应数值

常量池之后就是访问标志，前两个字节代表访问标志。

从上面的分析中常量池最后一个常量是#14 = Utf8 java/lang/Object，所以它后面的两个字节就代表访问标志，如下所示：

访问表示值与含义如下所示：

* ACC\_PUBLIC 0x0001 是否为public
* ACC\_FINAL 0x0010 是否为final
* ACC\_SUPER 0x0020 JDK 1.0.2以后编译出来的类该标志位都为真
* ACC\_INTERFACE 0x0200 是否为接口
* ACC\_ABSTRACT 0x0400 是否为抽象的（接口和抽象类）
* ACC\_SYNTHETIC 0x1000 表示这个代码并非由用户产生的
* ACC\_ANNOTATION 0x2000 是否为注解
* ACC\_ENUM 0x4000 是否为枚举

我们上面写了一个普通的Java类，ACC\_PUBLIC位为真，又由于JDK 1.0.2以后编译出来的类ACC\_SUPER标志位都为真，所以最终的值为：

0x0001 & 0x0020 = 0x0021

这个值就是上图中的值。

### 1.5 类索引、父类索引与接口索引

具体含义

类索引（用来确定该类的全限定名）、父类索引（用来确定该类的父类的全限定名）是一个u2类型的数据（单个类、单继承），接口索引是一个u2类型的集合（多接口实现，用来描述该类实现了哪些接口）

对应数值

类索引、父类索引与接口索引紧紧排列在访问标志之后。

类索引为0x0002，它的全限定名为com/guoxiaoxing/android/framework/demo/native\_framwork/vm/TestClass。

父类索引为0x0003，它的全限定名为java/lang/Object。

接口索引的第一项是一个u2类型的数据表示接口计数器，表示实现接口的个数。这里没有实现任何接口，所以为0x0000。

### 1.6 字段表集合

具体含义

字段表用来描述接口或者类里声明的变量、字段。包括类级变量以及实例级变量，但不包括方法内部声明的变量。

字段表结构如下所示：

field\_info {
 u2 access\_flags;//访问标志位，例如private、public等
 u2 name\_index;//字段的简单名称，例如int、long等
 u2 descriptor\_index;//方法的描述符，描述字段的数据类型，方法的参数列表和返回值
 u2 attributes\_count;
 attribute\_info attributes[attributes\_count];
}

access\_flags取值如下所示：

* ACC\_PUBLIC 0x0001 是否为 public;
* ACC\_PRIVATE 0x0002 是否为 private;
* ACC\_PROTECTED 0x0004 是否为 protected;
* ACC\_STATIC 0x0008 是否为 static;
* ACC\_FINAL 0x0010 是否为 final;
* ACC\_VOLATILE 0x0040 是否为 volatile;
* ACC\_TRANSIENT 0x0080 是否为 transient;
* ACC\_SYNTHETIC 0x1000 是否为 synthetic;
* ACC\_ENUM 0x4000 是否为enum.

descriptor\_index里描述符的含义如下所示：

* B byte
* C char
* D double
* F float
* I int
* J long
* S short
* Z boolean
* V void
* L Object, 例如 Ljava/lang/Object

对应数值

* 第一个u2类型的值为0x0001，代表当前容器计数器field\_count为1，说明这个类只有一个字段表数据。也就是我们上面定义的类成员变量private int m；
* 第二个u2类型的值为0x0002，代表access\_flags，说明这个成员变量的类型为private。
* 第三个u2类型的值为0x0005，代表name\_index为5。
* 第四个u2类型的值为0x0006，代表descriptor\_index为6。

### 1.7 方法表集合

方法便用来描述方法相关信息。

方法表的类型与字段表完全相同，如下所示：

method\_info {
 u2 access\_flags;//访问标志位，例如private、public等
 u2 name\_index;//方法名
 u2 descriptor\_index;//方法的描述符，描述字段的数据类型，方法的参数列表和返回值
 u2 attributes\_count;
 attribute\_info attributes[attributes\_count];
}

对应的值

* 第一个u2类型的值为0x0002，代表当前类有两个方法，即为构造函数和我们上面写的inc()方法。
* 第二个u2类型的值为0x0001，代表access\_flags，即方法的访问类型为public。
* 第三个u2类型的值为0x0007，代表name\_index，即为。
* 第四个u2类型的值为0x0008，代表descriptor\_index，即为()V。
* 第五个u2类型的值为0x0001，代表attributes\_count，表示该方法的属性集合有一项属性。
* 第六个u2类型的值为0x0009，代表属性名称，对应常量"code"，代表此属性是方法的字节码描述。

后续还有属性表集合等相关信息，这里就不再赘述，更多内容请参见Java虚拟机规范（Java SE 7）.pdf。

通过上面的描述，我们理解了Class存储格式的细节，那么这些是如何被加载到虚拟机中去的呢，加载到虚拟机之后又会发生什么变化呢？🤔

我们接着来看。

## 二 类的加载流程

什么是类的加载？🤔

类的加载就是虚拟机通过一个类的全限定名来获取描述此类的二进制字节流。

类加载的流程图如下所示：

**加载**

1. 通过一个类的全限定名来获取此类的二进制流。
2. 将这个字节流所代表的静态存储结构转换为方法去的运行时数据结构。
3. 在内存中生成一个代码这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口。

事实上，从哪里将一个类加载成二进制流是有很开发的，具体说来：

* 从zip包中读取，这就发展成了我们常见的JAR、AAR依赖。
* 运行时动态生成，这是我们常见的动态代理技术，在java.reflect.Proxy中就是用ProxyGenerateProxyClass来为特定接口生成代理类的二进制流。

**验证**

验证主要是验证加载进来的字节码二进制流是否符合虚拟机规范。

1. 文件格式验证：验证字节码流是否符合Class文件格式的规范，并且能够被当前版本的虚拟机处理。
2. 元数据验证：对字节码描述的语义进行分析，以保证其描述的信息符合Java语言规范的要求。
3. 字节码验证：对字节码的数据流和控制流进行分析，确定程序语义是合法的，符合逻辑的。
4. 符号引用验证：这个阶段在解析阶段中完成，虚拟机将符号引用转换为直接引用。

**准备**

准备阶段正式为类变量分为内存并设置变量的初始值，所使用的内存在方法去里被分配，这些变量指的是被static修饰的变量，而不包括实例的变量，实例的变量会伴随着对象的实例化一起在Java堆 中分配。

**解析**

解析阶段将符号引用转换为直接引用，符号引用我们前面已经说过，它以CONSTANT\_class\_info等符号来描述引用的目标，而直接引用指的是这些符号引用加载到虚拟机中以后 的内存地址。

这里的解析主要是针对我们上面提到的字段表、方法表、属性表里面的信息，具体说来，包括以下类型：

* 接口
* 字段
* 类方法
* 接口方法
* 方法类型
* 方法句柄
* 调用点限定符

**初始化**

初始化阶段开始执行类构造器()方法，该方法是由所有类变量的赋值动作和static语句块合并产生的

关于类构造器()方法，它和实例构造器()是不同的，关于这个方法我们需要注意以下几点：

* 类构造器()方法与实例构造器()方法不同，不需要显式的调用父类的构造器，虚拟机会保证父类构造器先执行。
* 类构造器()方法对于类或者接口不是必须的，如果一个类既没有赋值操作，也没有静态语句块，则不会生成该方法。
* 接口可以有变量初始化的赋值操作，因此接口也可以生成clinit>()方法、
* 虚拟机会保证一个类的()方法在多线程环境下能够被正确的加锁和同步。如果多个线程同时去初始化一个类，那么只会有一个线程执行该类的clinit>()方法 ，其他线程会被阻塞。

讲完了类的加载流程，我们接着来看看类加载器。

## 三 类加载器

### 3.1 Java虚拟机类加载机制

类的加载就是虚拟机通过一个类的全限定名来获取描述此类的二进制字节流，而完成这个加载动作的就是类加载器。

类和类加载器息息相关，判定两个类是否相等，只有在这两个类被同一个类加载器加载的情况下才有意义，否则即便是两个类来自同一个Class文件，被不同类加载器加载，它们也是不相等的。

注：这里的相等性保函Class对象的equals()方法、isAssignableFrom()方法、isInstance()方法的返回结果以及Instance关键字对对象所属关系的判定结果等。

类加载器可以分为三类：

* 启动类加载器（Bootstrap ClassLoader）：负责加载-Xbootclasspath参数所指定的路径的，并且是被虚拟机所识别的库到内存中。
* 扩展类加载器（Extension ClassLoader）：负责加载.ext.dirs系统变量所指定的路径的所有类库到内存中。
* 应用类加载器（Application ClassLoader）：负责加载用户类路径上的指定类库，如果应用程序中没有实现自己的类加载器，一般就是这个类加载器去加载应用程序中的类库。

这么多类加载器，那么当类在加载的时候会使用哪个加载器呢？🤔

这个时候就要提到类加载器的双亲委派模型，流程图如下所示：

双亲委派模型的整个工作流程非常的简单，如下所示：

如果一个类加载器收到了加载类的请求，它不会自己立即去加载类，它会先去请求父类加载器，每个层次的类加载器都是如此。层层传递，直到传递到最高层的类加载器，只有当 父类加载器反馈自己无法加载这个类，才会有当前子类加载器去加载该类。

关于双亲委派机制，在ClassLoader源码里也可以看出，如下所示：

public abstract class ClassLoader {

 protected Class<?> loadClass(String name, boolean resolve)
 throws ClassNotFoundException
 {
 //首先，检查该类是否已经被加载
 Class c = findLoadedClass(name);
 if (c == null) {
 long t0 = System.nanoTime();
 try {
 //先调用父类加载器去加载
 if (parent != null) {
 c = parent.loadClass(name, false);
 } else {
 c = findBootstrapClassOrNull(name);
 }
 } catch (ClassNotFoundException e) {
 // ClassNotFoundException thrown if class not found
 // from the non-null parent class loader
 }

 if (c == null) {
 //如果父类加载器没有加载到该类，则自己去执行加载
 long t1 = System.nanoTime();
 c = findClass(name);

 // this is the defining class loader; record the stats
 }
 }
 return c;
 }
}

为什么要这么做呢？🤔

这是为了要让越基础的类由越高层的类加载器加载，例如Object类，无论哪个类加载器去尝试加载这个类，最终都会传递给最高层的类加载器去加载，前面我们也说过，类的相等性是由 类与其类加载器共同判定的，这样Object类无论在何种类加载器环境下都是同一个类。

相反如果没有双亲委派模型，那么每个类加载器都会去加载Object，那么系统中就会出现多个不同的Object类了，如此一来系统的最基础的行为也就无法保证了。

理解了JVM上的类加载机制，我们再来看看Android虚拟机上上是如何加载类的。

### 3.2 Android虚拟机类加载机制

Java虚拟机加载的是class文件，而Android虚拟机加载的是dex文件（多个class文件合并而成），所以两者既有相似的地方，也有所不同。

Android类加载器类图如下所示：

可以看到Android类加载器的基类是BaseDexClassLoader，它有派生出两个子类加载器：

* PathClassLoader: 主要用于系统和app的类加载器,其中optimizedDirectory为null, 采用默认目录/data/dalvik-cache/
* DexClassLoader: 可以从包含classes.dex的jar或者apk中，加载类的类加载器, 可用于执行动态加载, 但必须是app私有可写目录来缓存odex文件. 能够加载系统没有安装的apk或者jar文件， 因此很多插件化方案都是采用DexClassLoader;

除了这两个子类以为，还有两个类：

* DexPathList：就跟它的名字那样，该类主要用来查找Dex、SO库的路径，并这些路径整体呈一个数组。
* DexFile：用来描述Dex文件，Dex的加载以及Class额查找都是由该类调用它的native方法完成的。

我们先来看看基类BaseDexClassLoader的构造方法

public BaseDexClassLoader(String dexPath, File optimizedDirectory,
 String librarySearchPath, ClassLoader parent) {
 super(parent);
 this.pathList = new DexPathList(this, dexPath, librarySearchPath, optimizedDirectory);
}

BaseDexClassLoader构造方法的四个参数的含义如下：

* dexPath：指的是在Androdi包含类和资源的jar/apk类型的文件集合，指的是包含dex文件。多个文件用“：”分隔开，用代码就是File.pathSeparator。
* optimizedDirectory：指的是odex优化文件存放的路径，可以为null，那么就采用默认的系统路径。
* libraryPath：指的是native库文件存放目录，也是以“：”分隔。
* parent：parent类加载器

DexClassLoader与PathClassLoader都继承于BaseDexClassLoader，这两个类只是提供了自己的构造函数，没有额外的实现，我们对比下它们的构造函数的区别。

**PathClassLoader**

public class PathClassLoader extends BaseDexClassLoader {

 public PathClassLoader(String dexPath, ClassLoader parent) {
 super(dexPath, null, null, parent);
 }

 public PathClassLoader(String dexPath, String librarySearchPath, ClassLoader parent) {
 super(dexPath, null, librarySearchPath, parent);
 }
}

**DexClassLoader**

public class DexClassLoader extends BaseDexClassLoader {

 public DexClassLoader(String dexPath, String optimizedDirectory,
 String librarySearchPath, ClassLoader parent) {
 super(dexPath, new File(optimizedDirectory), librarySearchPath, parent);
 }
}

可以发现这两个类的构造函数最大的差别就是DexClassLoader提供了optimizedDirectory，而PathClassLoader则没有，optimizedDirectory正是用来存放odex文件 的地方，以后可以利用DexClassLoader实现动态加载。

上面我们也说过，Dex的加载以及Class额查找都是由DexFile调用它的native方法完成的，我们来看看它的实现。

我们来看看Dex文件加载、类的查找加载的序列图，如下所示：

从上图Dex加载的流程可以看出，optimizedDirectory决定了调用哪一个DexFile的构造函数。

如果optimizedDirectory为空，这个时候其实是PathClassLoader，则调用：

DexFile(File file, ClassLoader loader, DexPathList.Element[] elements)
 throws IOException {
 this(file.getPath(), loader, elements);
}

如果optimizedDirectory不为空，这个时候其实是DexClassLoader，则调用：

private DexFile(String sourceName, String outputName, int flags, ClassLoader loader,
 DexPathList.Element[] elements) throws IOException {
 if (outputName != null) {
 try {
 String parent = new File(outputName).getParent();
 if (Libcore.os.getuid() != Libcore.os.stat(parent).st\_uid) {
 throw new IllegalArgumentException("Optimized data directory " + parent
 + " is not owned by the current user. Shared storage cannot protect"
 + " your application from code injection attacks.");
 }
 } catch (ErrnoException ignored) {
 // assume we'll fail with a more contextual error later
 }
 }

 mCookie = openDexFile(sourceName, outputName, flags, loader, elements);
 mFileName = sourceName;
 //System.out.println("DEX FILE cookie is " + mCookie + " sourceName=" + sourceName + " outputName=" + outputName);
}

所以你可以看到DexClassLoader在加载Dex文件的时候比PathClassLoader多了一个openDexFile()方法，该方法调用的是native方法openDexFileNative()方法。

👉 [dalvik\_system\_DexFile.cpp](https://android.googlesource.com/platform/dalvik/%2B/0dcf6bb/vm/native/dalvik_system_DexFile.cpp)

这个方法并不是真的打开Dex文件，而是将Dex文件以一种mmap的方式映射到虚拟机进程的地址空间中去，实现文件磁盘地址和进程虚拟地址空间中一段虚拟地址的一一对映关系。实现这样的映射关系后，虚拟机 进程就可以采用指针的方式读写操作这一段内存，而系统会自动回写脏页面到对应的文件磁盘上，即完成了对文件的操作而不必再调用read,write等系统调用函数。

关于mmap，它是一种很有用的文件读写方式，限于篇幅这里不再展开，更多关于mmap的内容可以参见文章：http://www.cnblogs.com/huxiao-tee/p/4660352.html

到这里，Android虚拟机的类加载机制就讲的差不多了，我们再来总结一下。

Android虚拟机有两个类加载器DexClassLoader与PathClassLoader，它们都继承于BaseDexClassLoader，它们内部都维护了一个DexPathList的对象，DexPathList主要用来存放指明包含dex文件、native库和优化odex目录。 Dex文件采用DexFile这个类来描述，Dex的加载以及类的查找都是通过DexFile调用它的native方法来完成的。