目录

前言

一个Java程序，会通过javac编译成class文件，然后通过虚拟机加载（ClassLoader ）到方法区，执行引擎会执行这些字节码，并翻译成操作系统底层相关函数。这是JVM运行java代码的整体流程。

由于Java中的ClassLoader 类加载 机制和Android中是不同的，本文将介绍Android虚拟机的类加载机制

Dex

了解JVM的老铁都知道，JVM运行的是Class字节码。
由于Class字节码文件IO操作比较多、查找类效率低、基于栈的加载模式加载速度慢以及内存占用较大的缺点,并不适用于移动端。所以DVM（Dalvik VM）设计了一种压缩文件的格式Dex（Dalvik Executable Format）
Dex文件是由多个.class文件处理压缩后的产物，Dex最终可以在Android运行时环境执行。

ART与Dalvik

1.DVM（Dalvik VM）是实现了JVM虚拟机规范的一个虚拟机，默认使用CMS垃圾回收器，与JVM不同的是DVM运行Dex文件；Android应用程序运行在虚拟机，一个进程对应一个单独的虚拟机实例，所以一个Android应用至少有对应一个单独的虚拟机实例
2.ART（Android Runtime）是在Anndroid 4.4版本引入的一个开发者选项，也是Android5.0及以上版本默认的Android运行时。ART虚拟机执行的是本地机器码（预编译得到机器码）。Android的运行时从Dalvik换成ART并不需要开发者将应用编译成机器码，APK任然是一个包含dex字节码的文件

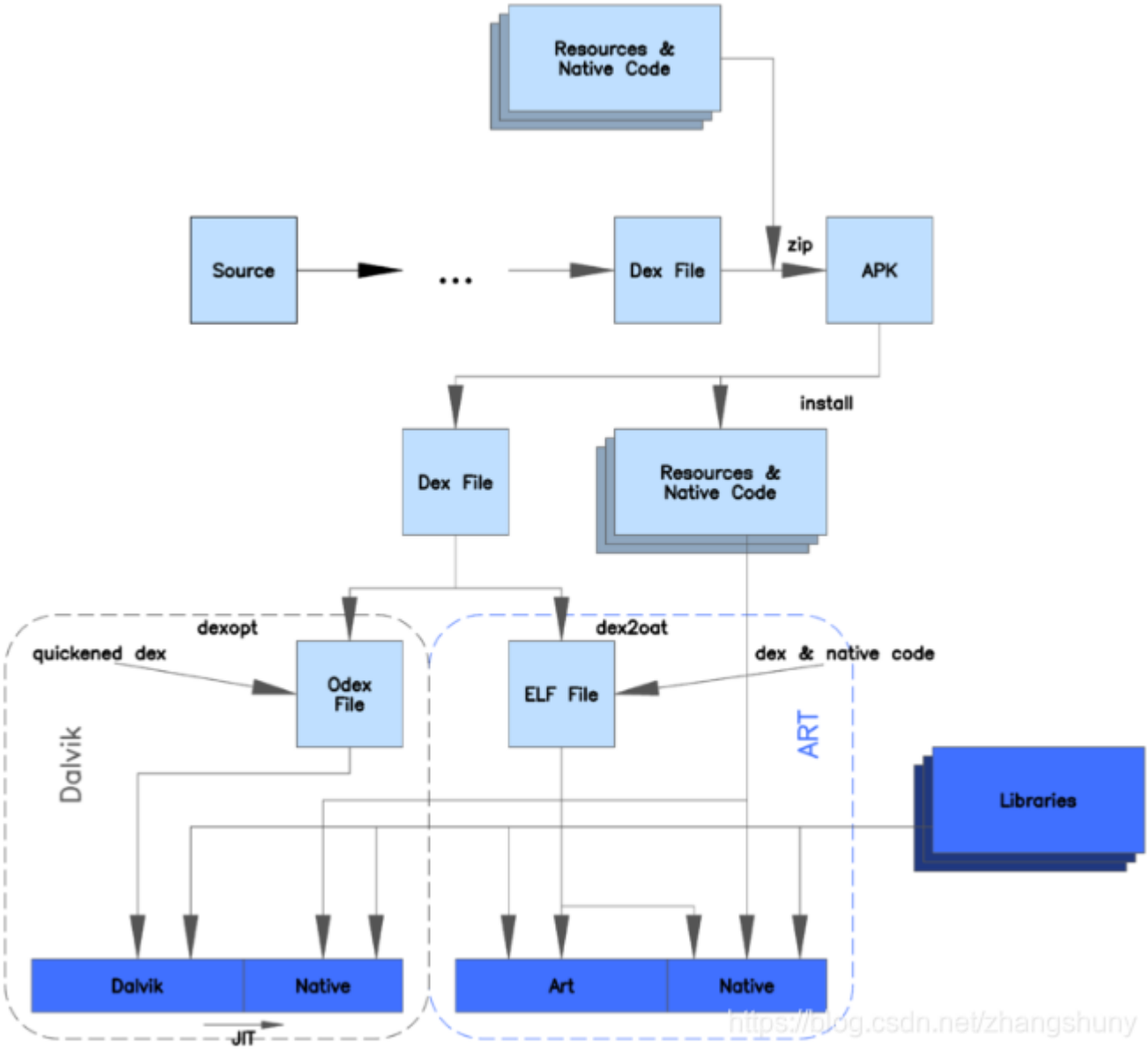
（ART和Dalvik都是运行Dex字节码的兼容运行时，因此针对Dalvik开发的应用也能运行在ART虚拟机中。）

3.ART与Dalvik的差别是二者执行的指令集不同，ART的指令集是基于寄存器的，Dalvik的指令集是基于栈的。

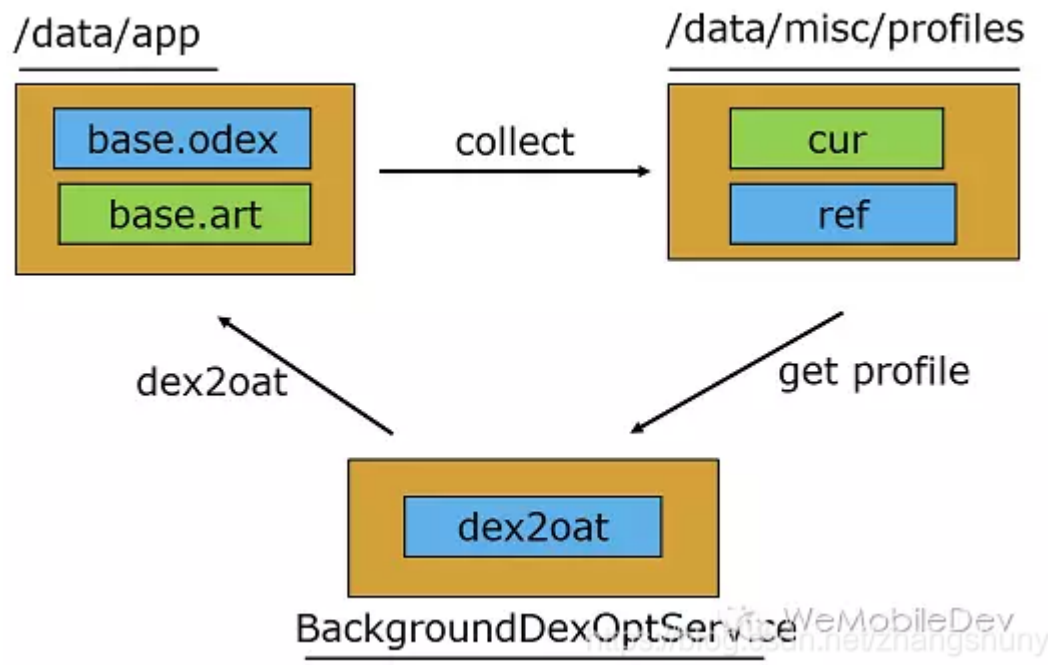
dexopt与dexaot

1.dexopt：在Dalvik加载一个dex文件的时候，会对dex文件进行验证和优化，验证和优化后变成odex（Optimized dex [/'ɒptɪmaɪzd/]）文件，odex与dex很像，使用了一些优化操作码,提高dex的执行效率；

2.dexaot:ART预先编译机制，在应用安装时对odex文件（dex优化成了odex）执行**AOT**（Ahead-Of-Time）提前编译操作，编译为OAT(OAT文件本质上是一个ELF文件，它将OAT文件格式内嵌在ELF文件里)可执行文件（机器码）



Android N(7.0)混合编译



有没有发现Android应用从7.0版本开始在手机配置差别不大的情况下安装应用变得比之前的Android版本下快了。

这就得益于Android N的混合编译

ART使用AOT（Ahead-Of-Time）编译，在应用安装在应用安装时对dex文件提前编译。使得安装变得缓慢，从Android N开始混合使用AOT编译+解释+JIT

1.初次安装应用时不进行AOT编译，运行过程中解释执行，对经常执行的方法进行JIT，经过JIT编译的方法会记录到Profile配置文件中

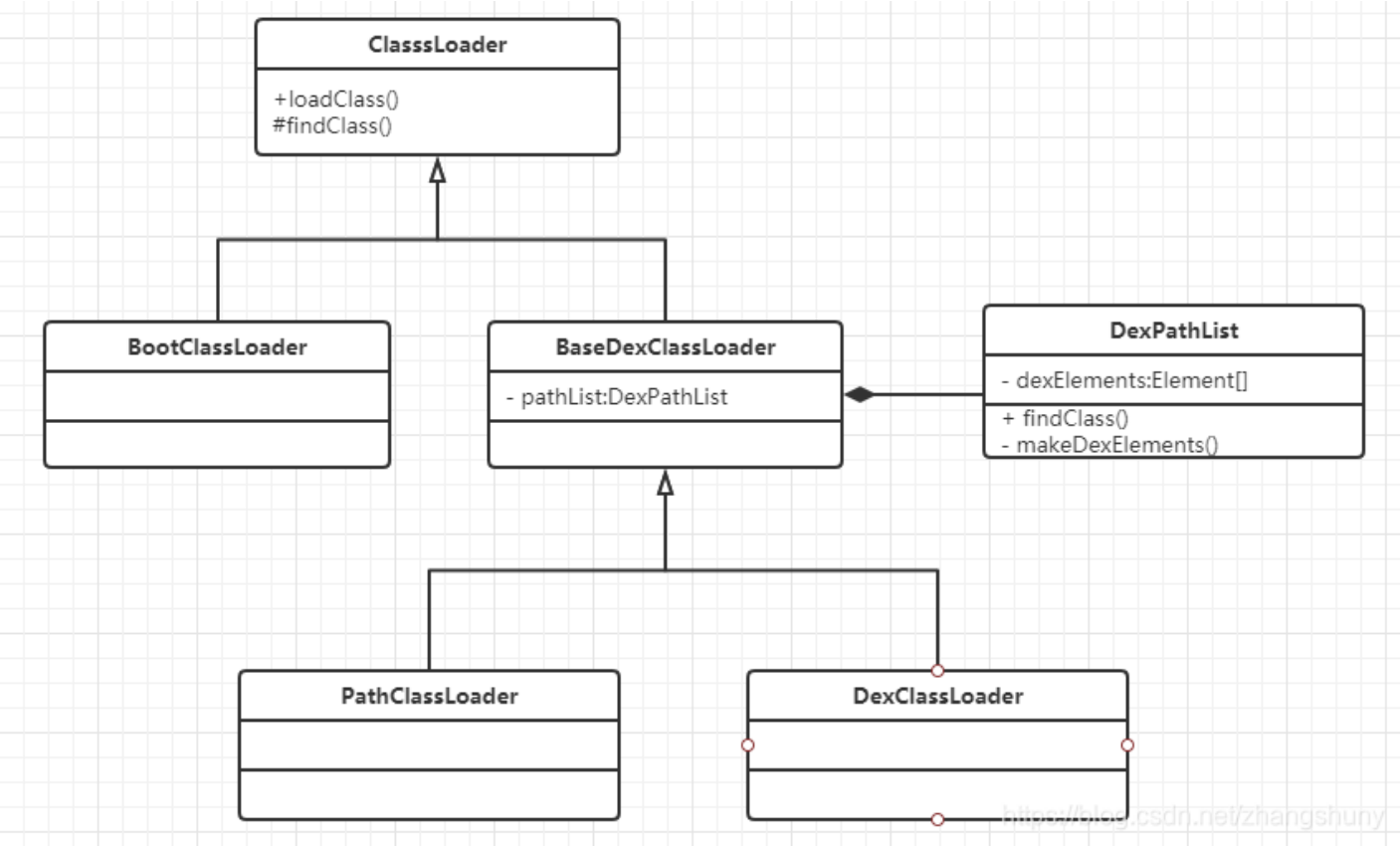
2.当处于设备闲置或充电或有四个小时间隔时，编译守护进程会运行，根据Profile文件对常用代码（“热代码”）进行AOT(All Of the Time compilation:全时段编译)编译，生成base.art文件（称为app_image 类对象映像）。这个art文件会在apk启动时会加入到PathClassloader的ClassTable中，系统在查找类的时候会先查找base.art中的。

Android N混合编译主要解决的问题：

> 1.应用安装时间过长；在N之前，应用在安装时需要对所有ClassN.dex做AOT机器码编译，类似微信这种比较大型的APP可能会耗时数分钟。但是往往我们只会使用一个应用20%的功能，剩下的80%我们付出了时间成本，却没带来太大的收益。
> 2.降低占ROM空间；同样全量编译AOT机器码，12M的dex编译结果往往可以达到50M之多。只编译用户用到或常用的20%功能，这对于存储空间不足的设备尤其重要。
> 3.提升系统与应用性能；减少了全量编译，降低了系统的耗电。在boot.art的基础上，每个应用增加了base.art, 通过预加载与缓存提升应用性能。
> 4.快速的系统升级；以往厂商ota时，需要对安装的所有应用做全量的AOT编译，这耗时非常久。事实上，同样只有20%的应用是我们经常使用的，给不常用的应用，不常用的功能付出的这些成本是不值得的。

ClassLoader介绍

Android9.0下ClassLoader的UML类图



一个Java程序，会通过javac编译成一个或多个class文件，运行的时候，需要将class文件加载到虚拟机中使用，负责加载这些文件的就是Java的类加载机制。
ClassLoader的作用就是加载class文件到方法区，提供给程序运行的时候使用。
ClassLoader是一个抽象类。具体的实现类有三个

1.BootClassLoader

BootClassLoader用于加载Android Framework层的class文件

2.PathClassLoader

PathClassLoader是Android应用程序的类加载器（加载已安装应用的dex），可以加载指定的dex和在java、zip、apk中的dex
在Dalvik虚拟机上PathClassLoader只能加载已安装的apk的dex，Android5.0开始使用的ART虚拟机，PathClassLoader也可以加载指定的dex和在java、zip、apk中的dex

3.DexClassLoader

DexClassLoader用于加载指定的dex和在java、zip、apk中的dex

双亲委托机制

双亲委托机制原理

某个类加载器在加载类的时候，首先委托给父加载器(ClassLoader parent)加载，其父加载器再委托给它的父加载器（如果有的话），以此类推。

如果父加载器可以完成加载类的任务的话，就成功返回
如果父加载器不能完成加载类的任务或者没有父加载器的话，自己去加载类。

使用双亲委托机制目的

1.安全。防止核心API库被篡改。

即使用户自定义的类与java核心api中的类名相同，因为双亲委托机制首先会使用父类加载器加载，由于PathClassLoader是Android应用程序的类加载器，应用开发者自己写的代码由PathClassLoader加载，PathClassLoader的父加载器是BootClassLoader，核心API类的Class已经被BootClassLoader加载过了，所以用户自定义的类不会加载。比如开发者自定义了一个String类，如果没有双亲委托机制，就会篡改String类

```
package java.lang;

public class String {
    //todo
}
```

为什么PathClassLoader的父加载器是BootClassLoader?
我的这篇文章有介绍
这里需要注意一下，父加载器不是父类加载器，两者不是继承和被继承的关系，而是parent是创建ClassLoader的一个参数

2.避免重复加载。当一个类被父类加载器加载过的时候，就没必要再用子类加载器加载一遍了。

ClassLoader.loadClass()关键代码

```
protected Class<?> loadClass(String name, boolean resolve)
        throws ClassNotFoundException
    {
            // First, check if the class has already been loaded 检查class是否已经
            Class<?> c = findLoadedClass(name);
            if (c == null) {
                try {
                    //父类加载器不为null就用父类加载器加载
                    if (parent != null) {
                        c = parent.loadClass(name, false);
                    } else {
                    //父类加载器为null就用BootClassLoader加载
                        c = findBootstrapClassOrNull(name);
                    }
                } catch (ClassNotFoundException e) {
                    // ClassNotFoundException thrown if class not found
                    // from the non-null parent class loader
                }
```

```
                if (c == null) {
                    // If still not found, then invoke findClass in order
                    // to find the class.
                    //如果任然没成功加载，自己加载
                    c = findClass(name);
                }
            }
            return c;
        }
```

参考

Android N混合编译与对热补丁影响解析