第四章 Android 虚拟机

4.1 Dalvik 虚拟机简介

Java 语言的一个非常重要的特点就是与平台的无关性。而使用 Java 虚拟机是实现这一特点的关键。一般的高级语言如果要在不同的平台上运行，至少需要编译成不同的目标代码。而引入 Java 语言虚拟机后,Java 语言在不同平台上运行时不需要重新编译。

Java 虚拟机屏蔽了与具体平台相关的信息，使得 Java 语言编译程序只需生成在 Java 虚拟机上运行的目标代码（字节码），就可以在多种平台上不加修改地运行。Java 虚拟机在执行字节码时，把字节码解释成具体平台上的机器指令执行。

Dalvik 的出现是为了躲避 Sun 公司 Java ME 的版权以及授权问题，由 Google 公司自己设计用于 Android 平台的 Java 虚拟机.

4.2 Dalvik 虚拟机的主要特征

- 专有的 DEX 文件格式

一个应用中会定义很多类，编译完成后即会有很多相应的 CLASS 文件，CLASS 文件间会有不少冗余的信息；而 DEX 文件格式会把所有的 CLASS 文件内容整合到一个文件中。这样，除了减少整体的文件尺寸，I/O 操作，也提高了类的查找速度。 原来每个类文件中的常量池，在 DEX 文件中由一个常量池来管理。

- 一个应用，一个虚拟机实例，一个进程

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 等等都是依赖底层操作系统实现的。所有 Android 应用的线程都对应一个 Linux 线程，虚拟机因而可以更多的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，可以最大程度的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 FORK 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，库的加载，预置类库的加载和初始化等等操作，而在系统需要一个新的虚拟机实例时，Zygote 通过复制自身，最快速的提供个系统。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。

- 基于寄存器

相对于基于堆栈的虚拟机实现，基于寄存器的虚拟机实现虽然在硬件通用性上要差一些，但是它在代码的执行效率上却更胜一筹。在基于寄存器的虚拟机里，可以更为有效的减少冗余指令的分发和减少内存的读写访问。

4.3 DEX 再优化

DEX 文件的结构是紧凑的，但是如果我们还想要求运行时的性能有进一步提高，我们就仍然需要对 DEX 文件进行进一步优化。优化主要是针对以下几个方面：

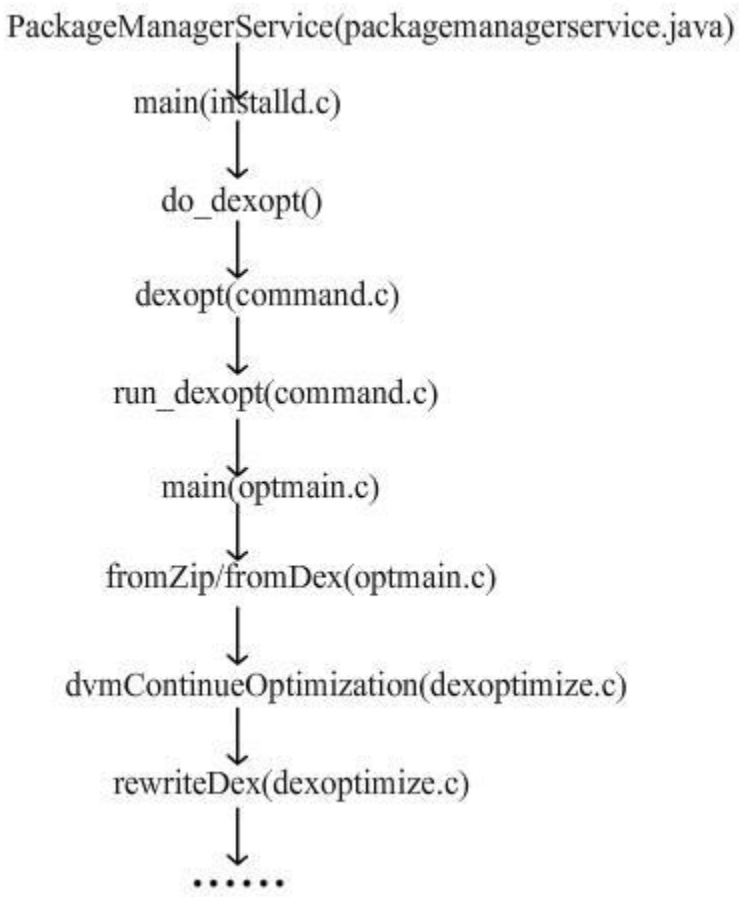
- 调整所有字段的字节序（LITTLE_ENDIAN）和对齐结构中的每一个域
- 验证 DEX 文件中的所有类
- 对一些特定的类进行优化，对方法里的操作码进行优化

优化后的文件大小会有所增加，应该是原 DEX 文件的 1-4 倍。

优化发生的时机有两个：

- 对于预置应用，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 ODEX 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。

代码调用流程：



4.4 java 程序和虚拟机

虚拟机实例的孵化器 Zygote 进程启动后，会注册 zygote socket（/dev/socket/zygote）用来侦听和处理运行字节码程序的请求。

参考 init.rc line 243：

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote
               --start-system-server
   socket zygote stream 666
   onrestart write /sys/android_power/request_state wake
   onrestart write /sys/power/state on
```

init 设置了参数“--zygote”，进入服务模式。具体实现在 Java 类 com.android.internal.os.ZygoteInit

当 zygote 进程接收到来自应用程序的请求后（当需要运行 manifest 文件中的<activity>，<service>，<receiver>和<provider>中的类时，就会通过 socket 向 zygote 发送启动命令），会 fork 出子进程，该子进程就是用来运行应用程序的虚拟机。由 zygote fork 出来的虚拟机会加载 java 程序的类和方法，并通过自身的 java 语言解释器来解释并运行 java 程序。

进程的创建是通过向 Zygote 服务器提交请求来实现的。

参考 frameworks/base/core/java/android/os/Process.java:

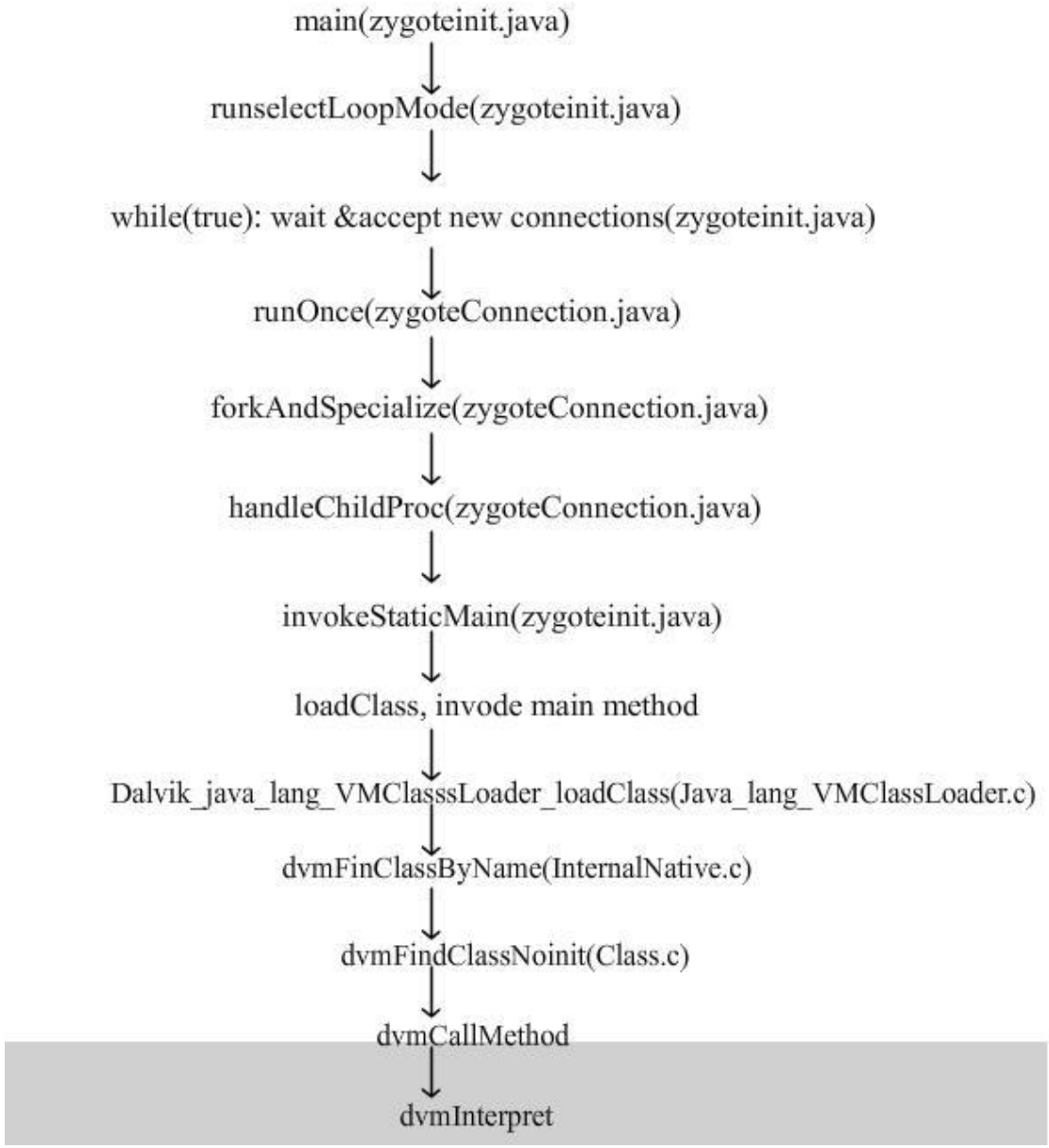
```
pid = zygoteSendArgsAndGetPid(argsForZygote);
```

zygote 收到命令后，在 runOnce()函数中 fork 一个子进程（ZygoteConnection.java）：

```
pid = Zygote.forkAndSpecialize(parsedArgs.uid, parsedArgs.gid, parsedArgs.gids,
parsedArgs.debugFlags, rlimits);
```

Dalvik 虚拟机也把传入类的 main 方法作为入口。

函数 RuntimeInit.zygoteInit()的最后，调用了 invokeStaticMain(startClass, startArgs)，把传入的命令行参数作为类名加载，执行其 main 函数。这样，用户的 java 程序最终运行起来。


main(zygoteinit.java)
runselectLoopMode(zygoteinit.java)
while(true): wait &accept new connections(zygoteinit.java)
runOnce(zygoteConnection.java)
forkAndSpecialize(zygoteConnection.java)
handleChildProc(zygoteConnection.java)
invokeStaticMain(zygoteinit.java)
loadClass, invode main method
Dalvik_java_lang_VMClasssLoader_loadClass(Java_lang_VMClassLoader.c)
dvmFinClassByName(InternalNative.c)
dvmFindClassNoinit(Class.c)
dvmCallMethod
dvmInterpret