Android提供了丰富的应用程序框架，它允许你建立创新的应用和游戏在Java语言环境中的移动设备。

Android应用提供多个入口点。App由多个可以单独调用的组件组成。比如：一个单独的Activity提供一个UserInterface，一个service在后台运行。在一个组件中通过intent启动另一个组件（可以是不同应用程序的组件）。这种模式对于单一的应用程序提供多个入口点，并默认允许其他app调用。

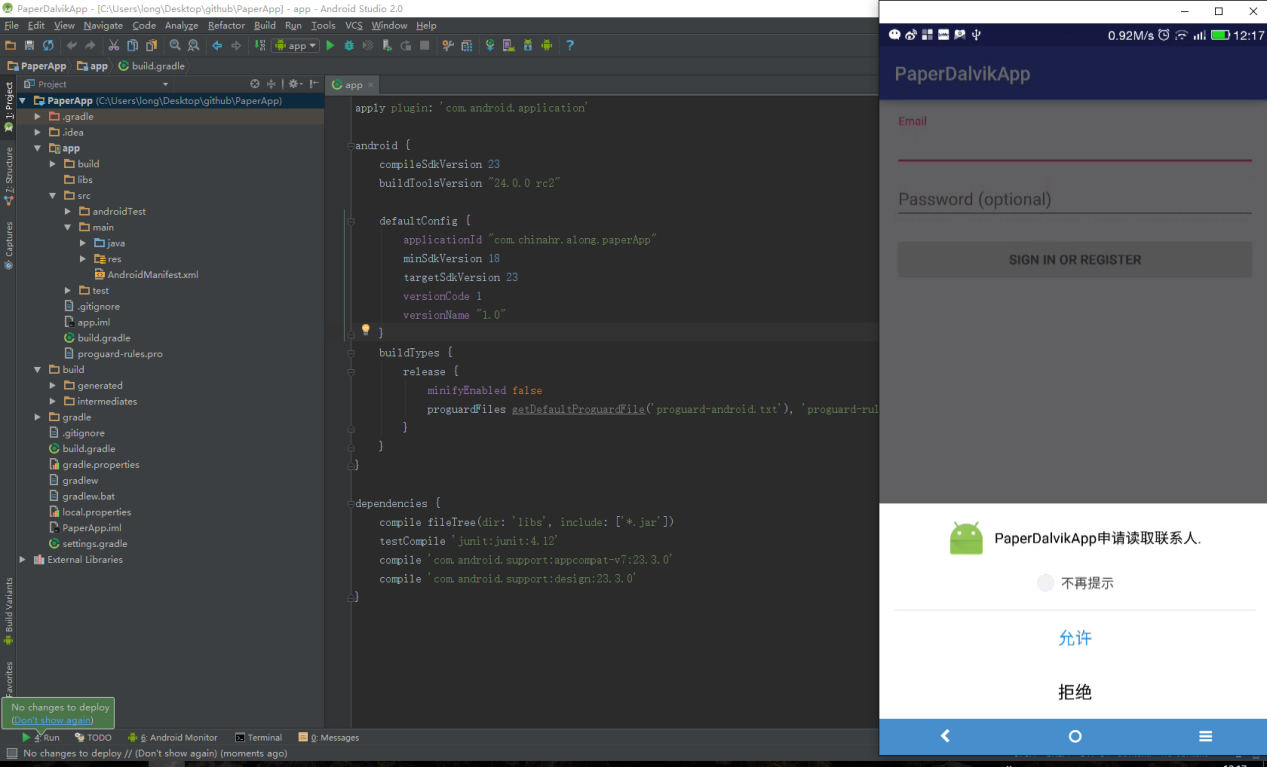
Apps可以适配不同的设备。Android提供了一个自适应的应用程​​序框架，它允许您为不同的设备配置提供独特的资源。例如，您可以创建不同的屏幕大小不同的XML布局文件和系统确定应用基于当前设备的屏幕尺寸，其布局。您可以在运行时查询设备功能的可用性，如果任何应用功能需要特定的硬件，如相机。如果需要，您还可以声明功能，您的应用程序需要这样的应用市场，如谷歌Play商店不允许在不支持该功能的设备的安装。

正常的工作流是这样的：创建，编译，测试，打包。

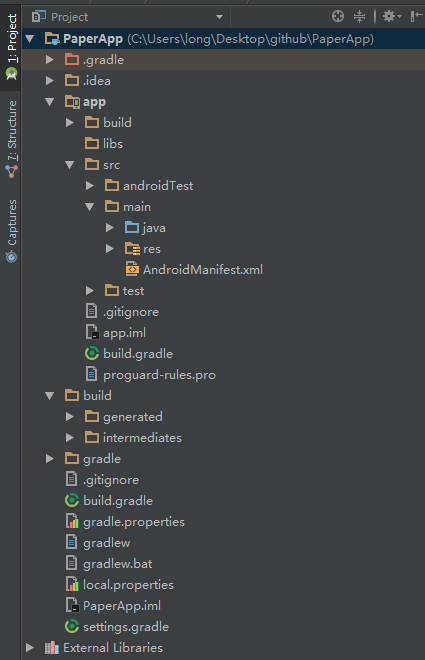
# 开始工作

## 创建应用

我们先创建一个app如下：



项目结构如下：



分为3块：

1. 编译系统：

.gradle 版本由gradle中的wrapper控制,通过gradlew执行

1. 配置文件：

Gradle.properties 与gradle相关属性的配置

Build.gradle 与gradle编译相关的配置

Local.properties 与本地环境配置相关

1. 应用模块：

build 编译后的文件存在的位置（包括最终生成的apk也在这里面）

libs 依赖的库所在的位置（jar和aar)

src 源代码所在的目录

src/main 主要代码所在位置（src/androidTest)就是测试代码所在位置了

src/main/assets android中附带的一些文件

src/main/java 最最重要的，我们的java代码所在的位置

src/main/jniLibs jni的一些动态库所在的默认位置(.so文件)

src/main/res android资源文件所在位置

src/main/AndroidManifest.xml AndroidManifest不用介绍了吧~

build.gradle 和这个项目有关的gradle配置，相当于这个项目的Makefile，一些项目的依赖就写在这里面

proguard.pro 代码混淆配置文件

## 编译，打包应用

构建过程使用许多工具，生成很多中间文件。在Android Studio中，运行gradle的编译任务会完成这一切的构建。构建过程非常灵活可配置。下图描述了参与编译的工具和流程：

标准的构建系统会合并配置在product flavors, build types, dependencies上的资源。过程如下：

1. Android Asset打包工具（AAPT）编译应用程序资源文件，如 AndroidManifest.xml文件，activity的布局XML文件。之后生成R.java，这样就可以从Java代码中引用资源。
2. aidl工具将.aidl文件转化成java的接口。
3. 编译java源码生成class文件。Dex工具将第三方library和class文件转化成dex 文件。

在标准Java环境， Java源代码被编译成Java字节码，其存储在.class文件。 .class文件是在运行时由JVM读取。在Java代码中.java生成一个.class文件。比如说，java源文件包含一个公共类，一个静态内部类和三个匿名类，编译过程将输出5个 .class文件。

在Android平台上， Java源代码仍然编译成.class文件。然后通过dx工具将.Class文件转换成.dex即Dalvik虚拟机执行文件。一个.class文件只包含一个类，一个.dex文件包含多个类。

1. Apkbuilder将编译的资源，没有编译的资源，dex文件打包成apk文件。
2. Apk文件创建成功后需要签名才能安装到设备上面。最后对齐apk，使应用程序运行时占用更少的内存。如果应用程序达到方法个数65k的限制，需要进行分包处理。
3. 编译后生成的apk输出在build文件中。

## 开启Android

当你按下电源开关后Android设备执行了以下步骤。

Android启动流程/过程

此处图片中step2中的一个单词拼写错了，Boot Loaeder应该为Boot Loader（多谢@jameslast 提醒）

第一步：启动电源以及系统启动

当电源按下，引导芯片代码开始从预定义的地方（固化在ROM）开始执行。加载引导程序到RAM，然后执行。

第二步：引导程序

引导程序是在Android操作系统开始运行前的一个小程序。引导程序是运行的第一个程序，因此它是针对特定的主板与芯片的。设备制造商要么使用很受欢迎的引导程序比如redboot、uboot、qi bootloader或者开发自己的引导程序，它不是Android操作系统的一部分。引导程序是OEM厂商或者运营商加锁和限制的地方。

引导程序分两个阶段执行。第一个阶段，检测外部的RAM以及加载对第二阶段有用的程序；第二阶段，引导程序设置网络、内存等等。这些对于运行内核是必要的，为了达到特殊的目标，引导程序可以根据配置参数或者输入数据设置内核。

Android引导程序可以在\bootable\bootloader\legacy\usbloader找到。

传统的加载器包含的个文件，需要在这里说明：

init.s初始化堆栈，清零BBS段，调用main.c的\_main()函数；

main.c初始化硬件（闹钟、主板、键盘、控制台），创建linux标签。

更多关于Android引导程序的可以在这里了解。

第三步：内核

Android内核与桌面linux内核启动的方式差不多。内核启动时，设置缓存、被保护存储器、计划列表，加载驱动。当内核完成系统设置，它首先在系统文件中寻找”init”文件，然后启动root进程或者系统的第一个进程。

第四步：init进程

init是第一个进程，我们可以说它是root进程或者说有进程的父进程。init进程有两个责任，一是挂载目录，比如/sys、/dev、/proc，二是运行init.rc脚本。

init进程可以在/system/core/init找到。

init.rc文件可以在/system/core/rootdir/init.rc找到。

readme.txt可以在/system/core/init/readme.txt找到。

对于init.rc文件，Android中有特定的格式以及规则。在Android中，我们叫做Android初始化语言。

Android初始化语言由四大类型的声明组成，即Actions（动作）、Commands（命令）、Services（服务）、以及Options（选项）。

Action（动作）：动作是以命令流程命名的，有一个触发器决定动作是否发生。

Service（服务）：服务是init进程启动的程序、当服务退出时init进程会视情况重启服务。

Options（选项）

选项是对服务的描述。它们影响init进程如何以及何时启动服务。

咱们来看看默认的init.rc文件。这里我只列出了主要的事件以及服务。

Table

Action/Service 描述

on early-init 设置init进程以及它创建的子进程的优先级，设置init进程的安全环境

on init 设置全局环境，为cpu accounting创建cgroup(资源控制)挂载点

on fs 挂载mtd分区

on post-fs 改变系统目录的访问权限

on post-fs-data 改变/data目录以及它的子目录的访问权限

on boot 基本网络的初始化，内存管理等等

service servicemanager 启动系统管理器管理所有的本地服务，比如位置、音频、Shared preference等等…

service zygote 启动zygote作为应用进程

在这个阶段你可以在设备的屏幕上看到“Android”logo了。

第五步

在Java中，我们知道不同的虚拟机实例会为不同的应用分配不同的内存。假如Android应用应该尽可能快地启动，但如果Android系统为每一个应用启动不同的Dalvik虚拟机实例，就会消耗大量的内存以及时间。因此，为了克服这个问题，Android系统创造了”Zygote”。Zygote让Dalvik虚拟机共享代码、低内存占用以及最小的启动时间成为可能。Zygote是一个虚拟器进程，正如我们在前一个步骤所说的在系统引导的时候启动。Zygote预加载以及初始化核心库类。通常，这些核心类是只读的，也是Android SDK或者核心框架的一部分。在Java虚拟机中，每一个实例都有它自己的核心库类文件和堆对象的拷贝。

Zygote加载进程

加载ZygoteInit类，源代码：/frameworks/base/core/java/com/android/internal/os/ZygoteInit.java

registerZygoteSocket()为zygote命令连接注册一个服务器套接字。

preloadClassed “preloaded-classes”是一个简单的包含一系列需要预加载类的文本文件，你可以在<Android Source>/frameworks/base找到“preloaded-classes”文件。

preloadResources() preloadResources也意味着本地主题、布局以及android.R文件中包含的所有东西都会用这个方法加载。

在这个阶段，你可以看到启动动画。

第六步：系统服务或服务

完成了上面几步之后，运行环境请求Zygote运行系统服务。系统服务同时使用native以及java编写，系统服务可以认为是一个进程。同一个系统服务在Android SDK可以以System Services形式获得。系统服务包含了所有的System Services。

Zygote创建新的进程去启动系统服务。你可以在ZygoteInit类的”startSystemServer”方法中找到源代码。

核心服务：

启动电源管理器；

创建Activity管理器；

启动电话注册；

启动包管理器；

设置Activity管理服务为系统进程；

启动上下文管理器；

启动系统Context Providers；

启动电池服务；

启动定时管理器；

启动传感服务；

启动窗口管理器；

启动蓝牙服务；

启动挂载服务。

其他服务：

启动状态栏服务；

启动硬件服务；

启动网络状态服务；

启动网络连接服务；

启动通知管理器；

启动设备存储监视服务；

启动定位管理器；

启动搜索服务；

启动剪切板服务；

启动登记服务；

启动壁纸服务；

启动音频服务；

启动耳机监听；

启动AdbSettingsObserver（处理adb命令）。

第七步：引导完成

一旦系统服务在内存中跑起来了，Android就完成了引导过程。在这个时候“ACTION\_BOOT\_COMPLETED”开机启动广播就会发出去。

## 安装应用

首先，将模拟器执行起来，直到正式进入系统。如果是手机的话，插上数据线（注意要确保手机驱动已经安装，不然会找不到改设备）

接着，打开一个cmd窗口， 输入：adb devices。用于搜索你的设备 或 模拟器，如果搜索到就可以对它进行安装。手机设备有可能搜索不到，那是你驱动没装好！

最后，输入：adb install --(apk完整路径）.例如：adb install paperApp.apk

执行完这两个命令之后，就交给系统来处理安装过程。流程如下：

（1） 拷贝apk文件到指定目录。在Android系统中，apk安装文件是会被保存起来的，默认情况下，用户安装的apk首先会被拷贝到 /data/app 目录下。

/data/app目录是用户有权限访问的目录，在安装apk的时候会自动选择该目录存放用户安装的文件，而系统出厂的apk文件则被放到了 /system 分区下,包括 /system/app，/system/vendor/app，以及 /system/priv-app 等等，该分区只有Root权限的用户才能访问，这也就是为什么在没有Root手机之前，我们无法删除系统出厂的app的原因了。

1. 解压apk，拷贝文件，创建应用的数据目录。为了加快app的启动速度，apk在安装的时候，会首先将app的可执行文件（dex）拷贝到 /data/dalvik-cache 目录，缓存起来。验证dex文件是否符合规范，并优化dex生成odex；接着会将odex文件解析成易于被dalvik加载和执行的dexFile结构体。在data/app-lib目录下存放二进制的so动态库。然后，在/data/data/目录下创建应用程序的数据目录（以应用的包名命名），存放应用的相关数据，如数据库、xml文件、cache、用户自定义的二进制so动态库等。
2. 解析apk的AndroidManifinest.xml文件

Android系统中，也有一个类似注册表的东西，用来记录当前所有安装的应用的基本信息，每次系统安装或者卸载了任何apk文件，都会更新这个文件。这个文件位于如下目录：/data/system/packages.xml。系统在安装apk的过程中，会解析apk的AndroidManifinest.xml文件，提取出这个apk的重要信息写入到packages.xml文件中，这些信息包括：权限、应用包名、APK的安装位置、版本、userID等等。

由此，我们就知道了为啥一些应用市场和软件管理类的app能够很清楚地知道当前手机所安装的所有的app，以及这些app的详细信息了。

（4） 其他操作

与windows应用安装类似，部分apk的安装也会向Launcher应用申请添加创建快捷方式。

# 运行时

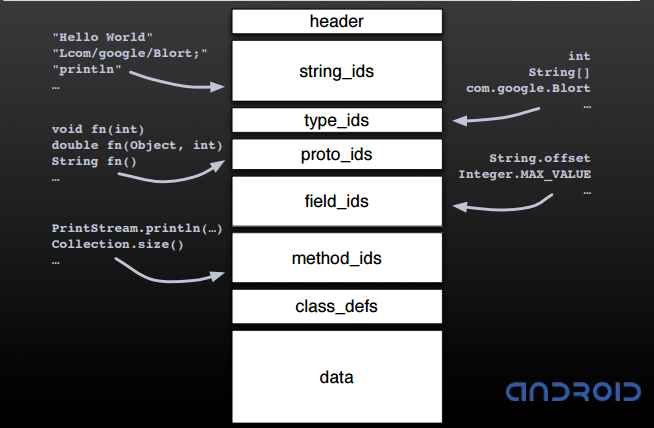
在4.4前，系统采用Dalcik虚拟机运行程序。4.4之后逐步过渡到ART运行时。

Dalvik虚拟机解释执行的dex字节码： java->java bytecode（class）->dalvik bytecode（dex） 注：相对JVM，Dalvik基于寄存器，且经过优化并允许有限的内存中同时运行多个虚拟机实例，每个Dalvik应用作为一个独立的Linxu进程执行。如果一个应用中有很多类，编译后会相应生成很多class文件，class文件之间也会有不少冗余信息，dex格式文件把所有classs文件内容整合到一个文件，这样可以减少整体文件占用和IO操作，同时也提高了类的查找速度。此外，dex格式文件增加了新的操作码支持，文件结构也相对简洁，使用等长的指令来提高解析速度。而且dex文件会尽量扩大只读结构的大小，来提高进程间数据共享的速度。

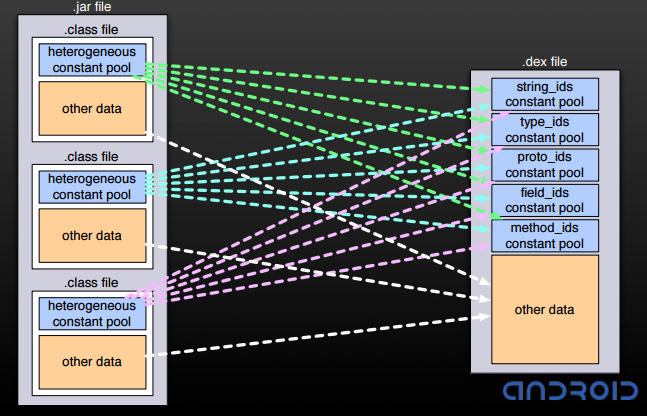
## Dalvik运行时

下面我们就看下dex的生命是如何发挥作用吧。

dex格式文件使用共享的、特定类型的常量池机制类节省内存。常量池存储类中的所有字面常量，它包括字符串常量、字段常量等值。



Java源文件生成dex文件的映射关系



|  |  |  |
| --- | --- | --- |
| 名称 | 格式 | 描述 |
| header | DexHeader | 文件头 |
| string\_ids | DexStringId | 字符串索引表，记录了各个字符所在的偏移值。 |
| type\_ids | DexTypeId | 类型资源的索引信息 |
| proto\_ids | DexProtoId | 方法原型资源的索引信息 |
| field\_ids | DexFieldId | 代码中的字段的索引信息 |
| method\_ids | DexMothodId | Dex文件中类方法数据的索引信息 |
| class\_defs | DexClassDef | 类的定义和数据索引信息 |
| datas | ubyte | 数据区，保存着各个类的数据。 |
| Link\_data | ubyte | 静态连接数据。 |

1. header是dex文件的文件头，简单的记录了dex文件的一些基本信息和大致的数据分布。header的大小固定为0x70,其中每一项信息所占用的大小也是固定的。同样在/dalvik/libdex/DexFile.h可以看到header\_item的定义：

\* Direct-mapped "header\_item" struct.

\*/

typedef 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; /\*字节序标号\*/

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;

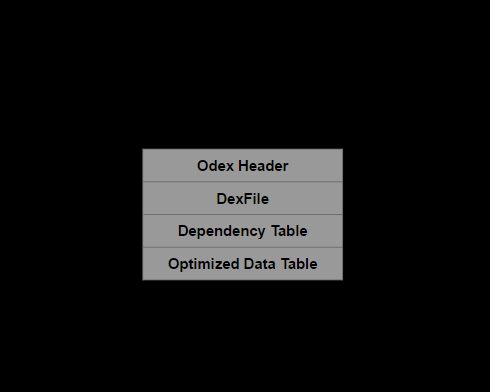
u4 classDefsOff;

u4 dataSize;

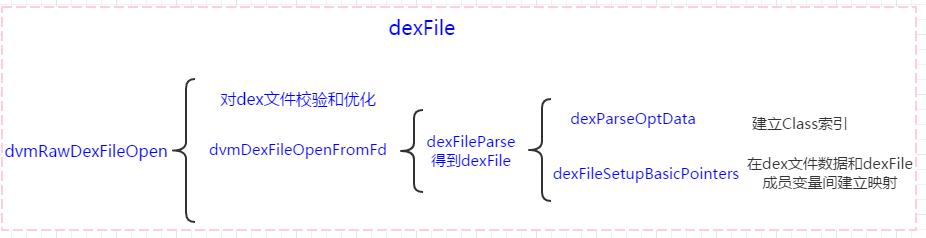
u4 dataOff;

} DexHeader;

1. string\_ids这一块区域中存储的是dex文件中字符串的资源的索引信息。即目标字符串在dex文件数据区的物理偏移地址。在DexFile.h中可以找到它的定义：该数据结构只有一个stringDataOff成员，当虚拟机需要读取该字符串是，只需要将dex文件在内存中的起始地址加上stringDataOff所表示的偏移值，即可得到该字符串在内存中的实际物理地址。在Dex文件中，每个字符串都对应了一个DexStringId数据结构，该数据结构的大小为4B,是一个确定的量。并且虚拟机可以通过头文件中的String\_ids\_size知道当前Dex文件的字符串总数，也就是string\_ids区域中DexStringId的总数，因此虚拟机通过简单的乘法即可实现对改索引资源进行正确的访问。
2. apk的安装在java层由Installer类执行，但最终通过socket由C层的installd来实现。dex的优化是由installd的do\_dexopt函数调用dexopt来完成操作。dex优化后生成odex后就要被拉去解析啦。



odex解析成dexFile结构体后，类加载器可以通过该结构体实现对Dex文件中各类数据的查找和获取(由此可知，dexFile和dex文件有对应关系)。看解析流程：

　dexFile的结构体如下：

/\*

\* Structure representing a DEX file.

\*

\* Code should regard DexFile as opaque, using the API calls provided here

\* to access specific structures.

\*/

typedef 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;

/\* mapped in "auxillary" section \*/

const DexClassLookup\* pClassLookup;

/\* points to start of DEX file data \*/

const u1\* baseAddr;

/\* track memory overhead for auxillary structures \*/

int overhead;

/\* additional app-specific data structures associated with the DEX \*/

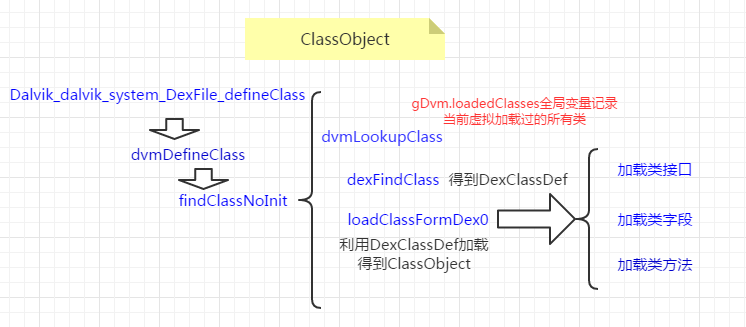
void\* auxData;

} DexFile;

　　ok，得到dexFile，至此类加载器的索引文件得到了。在dalvik代码执行中2个数据结构很重要：ClassObject、Method。类加载通过类描述符在dexFile(所以它叫dex在内存中的描述)结构体得到其在dex文件的地址，并对其解析和加载：



　　执行流程：



　　上面表述了类加载流程，其实质在是在内存中得到类的ClassObejct ！其实这也是FindClass函数(FindClass还是调用findClassNoInit)需要去做的流程。所以说类加载和findclass都只是为了在dalvik中得到该类的ClasssObject。类说完了，但dalvik是执行代码，我怎么执行你的类方法呢？记得上面说的Method结构(ClassObject结构里指向它的指针，ClassObject里当然包括方法咯)，它才是dalvik的执行单元，ClassObject只是描述，类似于odex和dexFile吧。具体可以看：[android dalvik浅析一：解释器及其执行](http://www.cnblogs.com/vendanner/p/4784381.html" \t "http://www.cnblogs.com/vendanner/p/_blank)

void dvmCallMethodV(Thread\* self, const Method\* method, Object\* obj,

bool fromJni, JValue\* pResult, va\_list args)

 　　下面我们来看下在dalvik在加载过程中使用到的数据结构(Method、ClassObject的结构体见源码Object.h)：

　　Method:　　　　java方法在dalvik中的结构体，是dalvik的执行单位；拥有指向ClassObject的指针

　　ClassObject:　　java类在dalvik中的结构体，包含多个Method结构；拥有指向DvmDex的指针；拥有指向加载该类的ClassLoader指针

　　DvmDex:　　　 拥有指向dexFile的指针

　　dexFile:　　　　dex在dalvik的描述结构

 　　一目了然，我们可以通过Method追踪到odex文件。

## ART运行时

ART虚拟机执行的本地机器码： java->java bytecode（class）->dalvik bytecode（dex）->optimized android runtime machine code（oat） 注：ART所使用的AOT（Ahead-Of-Time）编译，在应用首次安装时，字节码预编译成机器码存储在本地，也就是说在程序运行前编译。而Dalvik是典型的JIT（Just\_In\_Time），此模式下，应用每次运行的时候，字节码都需要即时编译器转换为机器码再执行，也就是在程序运行时编译。因此在App运行时，ART模式相对于Dalvik省去了解释字节码的过程，占用内存也相应减少，进而提高App的运行效率。无论是对dex字节码进行优化，还是将dex字节码翻译成本地机器码，最终得到的结果都是保存在相同名称的一个odex文件里面的，但是前者对应的是一个dey文件（表示这是一个优化过的dex），后者对应的是一个oat文件（实际上是一个自定义的elf文件，里面包含的都是本地机器指令）。

一个应用程序只有一个classes.dex文件， 这个classes.dex文件经过编译后，就得到一个OAT主执行文件。不过，应用程序也可以在运行时动态加载DEX文件。这些动态加载的DEX文件在加载的时候同样会被翻译成OAT再运行，它们相应打包在应用程序的classes.dex文件来说，就不属于主执行文件了。

OatFile类的静态成员函数Open的实现虽然只有寥寥几行代码，但是要理解它还得先理解宏ART\_USE\_PORTABLE\_COMPILER的的作用。在前面Android运行时ART简要介绍和学习计划一文中提到，ART运行时利用LLVM编译框架来将DEX字节码翻译成本地机器指令，其中要通过一个称为Backend的模块来生成本地机器指令。这些生成的机器指令就保存在ELF文件格式的OAT文件的oatexec段中。

ART运行时会为每一个类方法都生成一系列的本地机器指令。这些本地机器指令不是孤立存在的，因为它们可能需要其它的函数来完成自己的功能。例如，它们可能需要调用ART运行时的堆管理系统提供的接口来为对象分配内存空间。这样就会涉及到一个模块依赖性问题，就好像我们在编写程序时，需要依赖C库提供的接口一样。这要求Backend为类方法生成本地机器指令时，要处理调用其它模块提供的函数的问题。

ART运行时支持两种类型的Backend：Portable和Quick。Portable类型的Backend通过集成在LLVM编译框架里面的一个称为MCLinker的链接器来生成本地机器指令。关于MCLinker的更多知识，可以参考https://code.google.com/p/mclinker。简单来说，假设我们有一个模块A，它依赖于模块B、C和D，那么在为模块A生成本地机器指令时，指出它依赖于模块B、C和D就行了。在生成的OAT文件中会记录好这些依赖关系，这是ELF文件格式本来就支持的特性。这些OAT文件要通过系统的动态链接器提供的dlopen函数来加载。函数dlopen在加载OAT文件的时候，会通过重定位技术来处理好它与其它模块的依赖关系，使得它能够调用其它模块提供的接口。这个实际上就通用的编译器、静态连接器以及动态链接器合作在一起干的事情，MCLinker扮演的就是静态链接器的角色。既然是通用的技术，因为就称能产生这种OAT文件的Backend为Portable类型的。

另一方面，Quick类型的Backend生成的本地机器指令用另外一种方式来处理依赖模块之间的依赖关系。简单来说，就是ART运行时会在每一个线程的TLS（线程本地区域）提供一个函数表。有了这个函数表之后，Quick类型的Backend生成的本地机器指令就可以通过它来调用其它模块的函数。也就是说，Quick类型的Backend生成的本地机器指令要依赖于ART运行时提供的函数表。这使得Quick类型的Backend生成的OAT文件在加载时不需要再处理模式之间的依赖关系。再通俗一点说的就是Quick类型的Backend生成的OAT文件在加载时不需要重定位，因此就不需要通过系统的动态链接器提供的dlopen函数来加载。由于省去重定位这个操作，Quick类型的Backend生成的OAT文件在加载时就会更快，这也是称为Quick的缘由。

关于ART运行时类型为Portable和Quick两种类型的Backend，我们就暂时讲解到这里，后面分析ART运行时执行类方法的时候，我们再详细分析。现在我们需要知道的就是，如果在编译ART运行时时，定义了宏ART\_USE\_PORTABLE\_COMPILER，那么就表示要使用Portable类型的Backend来生成OAT文件，否则就使用Quick类型的Backend来生成OAT文件。默认情况下，使用的是Quick类型的Backend。

接下就可以很好地理解OatFile类的静态成员函数Open的实现了：

1. 如果编译时指定了ART\_USE\_PORTABLE\_COMPILER宏，并且参数executable为true，那么就通过OatFile类的静态成员函数OpenDlopen来加载指定的OAT文件。OatFile类的静态成员函数OpenDlopen直接通过动态链接器提供的dlopen函数来加载OAT文件。

2. 其余情况下，通过OatFile类的静态成员函数OpenElfFile来手动加载指定的OAT文件。这种方式是按照ELF文件格式来解析要加载的OAT文件的，并且根据解析获得的信息将OAT里面相应的段加载到内存中来。

接下来我们就分别看看OatFile类的静态成员函数OpenDlopen和OpenElfFile的实现，以便可以对OAT文件有更清楚的认识。

OatFile类的成员函数Dlopen首先是通过动态链接器提供的dlopen函数将参数elf\_filename指定的OAT文件加载到内存中来，接着同样是通过动态链接器提供的dlsym函数从加载进来的OAT文件获得两个导出符号oatdata和oatlastword的地址，分别保存在当前正在处理的OatFile对象的成员变量begin\_和end\_中。根据图1所示，符号oatdata的地址即为OAT文件里面的oatdata段加载到内存中的开始地址，而符号oatlastword的地址即为OAT文件里面的oatexec加载到内存中的结束地址。符号oatlastword本身也是属于oatexec段的，它自己占用了一个地址，也就是sizeof(uint32\_t)个字节，于是将前面得到的end\_值加上sizeof(uint32\_t)，得到的才是oatexec段的结束地址。

实际上，上面得到的begin\_值指向的是加载内存中的oatdata段的头部，即OAT头。这个OAT头描述了OAT文件所包含的DEX文件的信息，以及定义在这些DEX文件里面的类方法所对应的本地机器指令在内存的位置。另外，上面得到的end\_是用来在解析OAT头时验证数据的正确性的。此外，如果参数requested\_base的值不等于0，那么就要求oatdata段必须要加载到requested\_base指定的位置去，也就是上面得到的begin\_值与requested\_base值相等，否则的话就会出错返回。

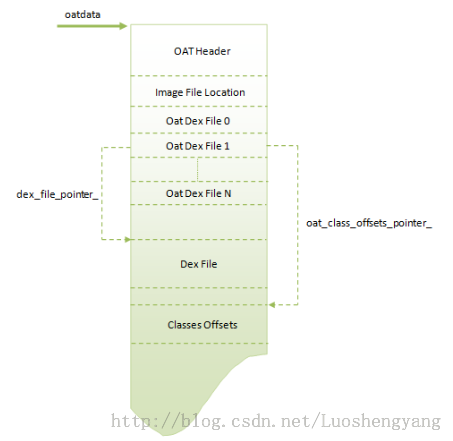
最后，OatFile类的成员函数Dlopen通过调用另外一个成员函数Setup来解析已经加载内存中的oatdata段

OatFile类的静态成员函数OpenElfFile的实现与前面分析的成员函数Dlopen是很类似的，唯一不同的是前者通过ElfFile类来手动加载参数file指定的OAT文件，实际上就是按照ELF文件格式来解析参数file指定的OAT文件，并且将文件里面的oatdata段和oatexec段加载到内存中来。我们可以将ElfFile类看作是ART运行时自己实现的OAT文件动态链接器。一旦参数file指定的OAT文件指定的文件加载完成之后，我们同样是通过两个导出符号oatdata和oatlastword来获得oatdata段和oatexec段的起止位置。同样，如果参数requested\_base的值不等于0，那么就要求oatdata段必须要加载到requested\_base指定的位置去。

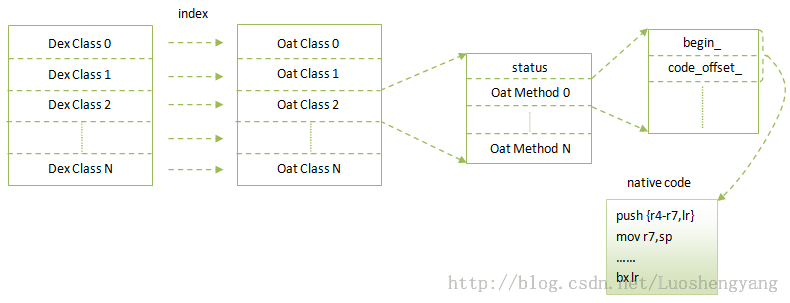
将参数file指定的OAT文件加载到内存之后，OatFile类的静态成员函数OpenElfFile最后也是调用OatFile类的成员函数Setup来解析其中的oatdata段。OatFile类的成员函数Setup定义在文件art/runtime/oat\_file.cc中，我们分三部分来阅读，以便可以更好地理解OAT文件的格式。

 通过第5个信息我们可以找到DEX文件里面的每一个类方法对应的本地机器指令。这个数组的大小等于header->class\_defs\_size\_，即DEX文件里面的每一个类在数组中都对应有一个偏移值。这里的header指向的是DEX文件头，它的class\_defs\_size\_描述了DEX文件包含的类的个数。在DEX文件中，每一个类都是有一个从0开始的编号，该编号就是用来索引到上述数组的，从而获得对应的类所有方法的本地机器指令信息。

        最后，上述得到的每一个DEX文件的信息都被封装在一个OatDexFile对象中，以便以后可以直接访问。如果我们使用OatDexFile来描述每一个DEX文件的描述信息，那么就可以通过图4看到这些描述信息在oatdata段的位置：



  通过上面对OAT文件加载过程的分析，我们就可以清楚地看到OAT文件的格式，以及如何在OAT文件中找到一个类方法的本地机器指令。我们通过图7来总结在OAT文件中找到一个类方法的本地机器指令的过程：



我们从左往右来看图7。首先是根据类签名信息从包含在OAT文件里面的DEX文件中查找目标Class的编号，然后再根据这个编号找到在OAT文件中找到对应的OatClass。接下来再根据方法签名从包含在OAT文件里面的DEX文件中查找目标方法的编号，然后再根据这个编号在前面找到的OatClass中找到对应的OatMethod。有了这个OatMethod之后，我们就根据它的成员变量begin\_和code\_offset\_找到目标类方法的本地机器指令了。其中，从DEX文件中根据签名找到类和方法的编号要求对DEX文件进行解析，这就需要利用Dalvik虚拟机的知识了。

## Classloader

在运行时将会有多个classloader被加载进来。通常情况下，类加载器分为逻辑上的树，子类加载器委托给父类加载器所有请求。只有当父类加载器不能满足要求时，子类加载器才会试图加载。

classloader是一个实现类加载器通用功能的抽象类。

1. 加载类的实例时，先查询当前classloader是否加载过？有，就直接返回；没有，则查询父类是否加载过？有，则返回；没有，就执行当前classloader的findclass方法。这样做的结果是：如果类被父类ClassLoader，那么之后整个系统的生命周期中，永远不会再加载。所以想要实现自定义加载class，需要复写ClassLoader的loadclass()。

作用：共享功能，一些Framework层级的类一旦被顶层的ClassLoader加载过就缓存在内存里面，以后任何地方用到都不需要重新加载。除此之外还有隔离功能，不同继承路线上的ClassLoader加载的类肯定不是同一个类，这样的限制避免了用户自己的代码冒充核心类库的类访问核心类库包可见成员的情况。

protected Class<?> loadClass(String className, boolean resolve) throws ClassNotFoundException {

Class<?> clazz = findLoadedClass(className);

if (clazz == null) {

ClassNotFoundException suppressed = null;

try {

clazz = parent.loadClass(className, false);

} catch (ClassNotFoundException e) {

suppressed = e;

}

if (clazz == null) {

try {

clazz = findClass(className);

} catch (ClassNotFoundException e) {

e.addSuppressed(suppressed);

throw e;

}

}

}

return clazz;

}

2.ClassLoader的内部类BootClassLoader--系统唯一明确的根ClassLoader。显然BootClassLoader实现了ClassLoader的loadClass()。

BootClassLoader将会通过Class类的native方法去加载框架需要的类。

@Override

protected Class<?> loadClass(String className, boolean resolve)

throws ClassNotFoundException {

Class<?> clazz = findLoadedClass(className);

if (clazz == null) {

clazz = findClass(className);

}

return clazz;

}

3.静态内部类SystemClassLoader,创建一个PathClassLoader,并且指定父加载器为BootClassLoader。

static private class SystemClassLoader {

public static ClassLoader loader = ClassLoader.createSystemClassLoader();

}

private static ClassLoader createSystemClassLoader() {

String classPath = System.getProperty("java.class.path", ".");

// String[] paths = classPath.split(":");

// URL[] urls = new URL[paths.length];

// for (int i = 0; i < paths.length; i++) {

// try {

// urls[i] = new URL("file://" + paths[i]);

// }

// catch (Exception ex) {

// ex.printStackTrace();

// }

// }

//

// return new java.net.URLClassLoader(urls, null);

// TODO Make this a java.net.URLClassLoader once we have those?

return new PathClassLoader(classPath, BootClassLoader.getInstance());

}

Android提供了几个具体的实现：

BaseDexClassLoader.加载dex文件的父类，提供基础功能。没有重写ClassLoader订的loadclass()，说明依然遵循双亲委托机制。

1.返回给定package的信息。

@Override

protected synchronized Package getPackage(String name) {

if (name != null && !name.isEmpty()) {

Package pack = super.getPackage(name);

if (pack == null) {

pack = definePackage(name, "Unknown", "0.0", "Unknown", "Unknown", "0.0", "Unknown", null);

}

return pack;

}

return null;

}

2.重写findClass(),通过pathList去查找类的方法。

@Override

protected Class<?> findClass(String name) throws ClassNotFoundException {

List<Throwable> suppressedExceptions = new ArrayList<Throwable>();

Class c = pathList.findClass(name, suppressedExceptions);

if (c == null) {

ClassNotFoundException cnfe = new ClassNotFoundException("Didn't find class \"" + name + "\" on path: " +

pathList);

for (Throwable t : suppressedExceptions) {

cnfe.addSuppressed(t);

}

throw cnfe;

}

return c;

}

3.遍历dex或resource的列表。通过DexFile的loadClassBinaryName()返回Class

public Class findClass(String name, List<Throwable> suppressed) {

for (Element element : dexElements) {

DexFile1 dex = element.dexFile1;

if (dex != null) {

Class clazz = dex.loadClassBinaryName(name, definingContext, suppressed);

if (clazz != null) {

return clazz;

}

}

}

if (dexElementsSuppressedExceptions != null) {

suppressed.addAll(Arrays.asList(dexElementsSuppressedExceptions));

}

return null;

}

4.将Class的name与ClassLoader进行绑定，唯一确定一个类。

public Class loadClassBinaryName(String name, ClassLoader loader, List<Throwable> suppressed) {

return defineClass(name, loader, mCookie, suppressed);

}

private static Class defineClass(String name, ClassLoader loader, long cookie, List<Throwable> suppressed) {

Class result = null;

try {

result = defineClassNative(name, loader, cookie);

} catch (NoClassDefFoundError e) {

if (suppressed != null) {

suppressed.add(e);

}

} catch (ClassNotFoundException e) {

if (suppressed != null) {

suppressed.add(e);

}

}

return result;

}

PathClassLoader,继承BaseDexClassLoader.系统默认的加载安装应用中声明的类的ClassLoader。

1.加载包含类和资源的jar包或者apk文件

public PathClassLoader(String dexPath, ClassLoader parent) {

super(dexPath, null, null, parent);

}

2.加载包含dex的JAR、ZIP、APK文件或者单独的dex文件。另外还有一个native的library文件。

public PathClassLoader(String dexPath, String libraryPath, ClassLoader parent) {

super(dexPath, null, libraryPath, parent);

}

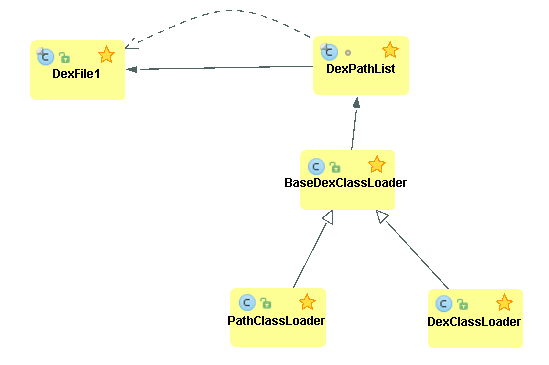
DexClassLoader,继承BaseDexClassLoader.加载不是应用中的代码。存放在应用私有文件目录下，读取jar、apk文件的列表，优化缓存到指

定文件夹下。

public DexClassLoader(String dexPath, String optimizedDirectory, String libraryPath, ClassLoader parent) {

super(dexPath, new File(optimizedDirectory), libraryPath, parent);

}



两者区别：

PathClassLoader是通过new DexFile(path)来产生DexFile对象。DexClassLoader通过DexFile通过静态方法LoadDex(path,outputpath,0)得到DexFile对象。也就是说，PathClassLoader不能主动从zip包中释放dex,只支持操作dex文件；而DexClassLoader可以支持.apk、.jar和.dex文件，并且会在指定的outpath路径释放出dex文件。

private static DexFile loadDexFile(File file, File optimizedDirectory) throws IOException {

if (optimizedDirectory == null) {

return new DexFile(file);

} else {

String optimizedPath = optimizedPathFor(file, optimizedDirectory);

return DexFile.loadDex(file.getPath(), optimizedPath, 0);

}

}

应用运行加载一个类是这样的过程：BootClassLoader加载基础类，如java.lang.String。PathClassLoader加载“/data/app/packagename-1/base.apk”里面的类，在这个过程中，通过BaseClassLoader的成员变量DexPathList去后者的成员变量Element的数组中加载类，并且是按照顺序去读取，并且只有在真正的从dex加载的时候才会指定类所对应的ClassLoader。

同理：系统加载native library时的流程相似：

@Override

public String findLibrary(String name) {

return pathList.findLibrary(name);

}

/\*\*

\* List of native library directories.

\*/

private final File[] nativeLibraryDirectories;

public String findLibrary(String libraryName) {

String fileName = System.mapLibraryName(libraryName);

for (File directory : nativeLibraryDirectories) {

String path = new File(directory, fileName).getPath();

if (IoUtils.canOpenReadOnly(path)) {

return path;

}

}

return null;

}

也就是说，如果能做到将修复的dex文件所在的Element或者nativeLibraryDirectories在bug类所在Element或者nativeLibraryDirectories之前，所有问题就迎刃而解。这里有很多并且，这些并且非一不可，成为了此次能够热修复成功的关键。

## 动态加载补丁dex

1. 修复类打包成dex文件。见前-编译打包应用。
2. 通过DexClassLoader将dex文件加载到内存中。
3. 获取当前PathClassLoader的Element数组和DexClassLoader的Element数组，将二者合并之后（后者在前）重新赋值给前者成员变量pathList的成员变量dexElements。

如此便实现了class的覆盖。

## 两个问题

1.android中许多组件类是需要在Manifest文件里面注册后才能工作的，所以即使动态加载了一个新的组件类进来，没有注册的话还是无法工作。

2.Res资源。在编译时期，资源与R.id对应好。运行时通过这些id从Resource实例中获取对应的资源。如果是运行时动态加载进来的新类，用到R.id的地方将会抛出异常。

说到底，核心要解决的问题是：如何给外部的新类提供上下文环境？