[简介 2](#_Toc27841)

[创建应用 2](#_Toc5622)

[编译，打包应用 4](#_Toc19643)

[开机 6](#_Toc27247)

[安装应用 8](#_Toc32698)

[运行时 8](#_Toc5968)

[加载器Classloader 13](#_Toc8735)

[动态加载补丁dex 18](#_Toc17712)

[三个关键问题 19](#_Toc10479)

[管理后台 21](#_Toc4235)

[代码安全 21](#_Toc29726)

一切开始于一个原点？

# 简介

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。

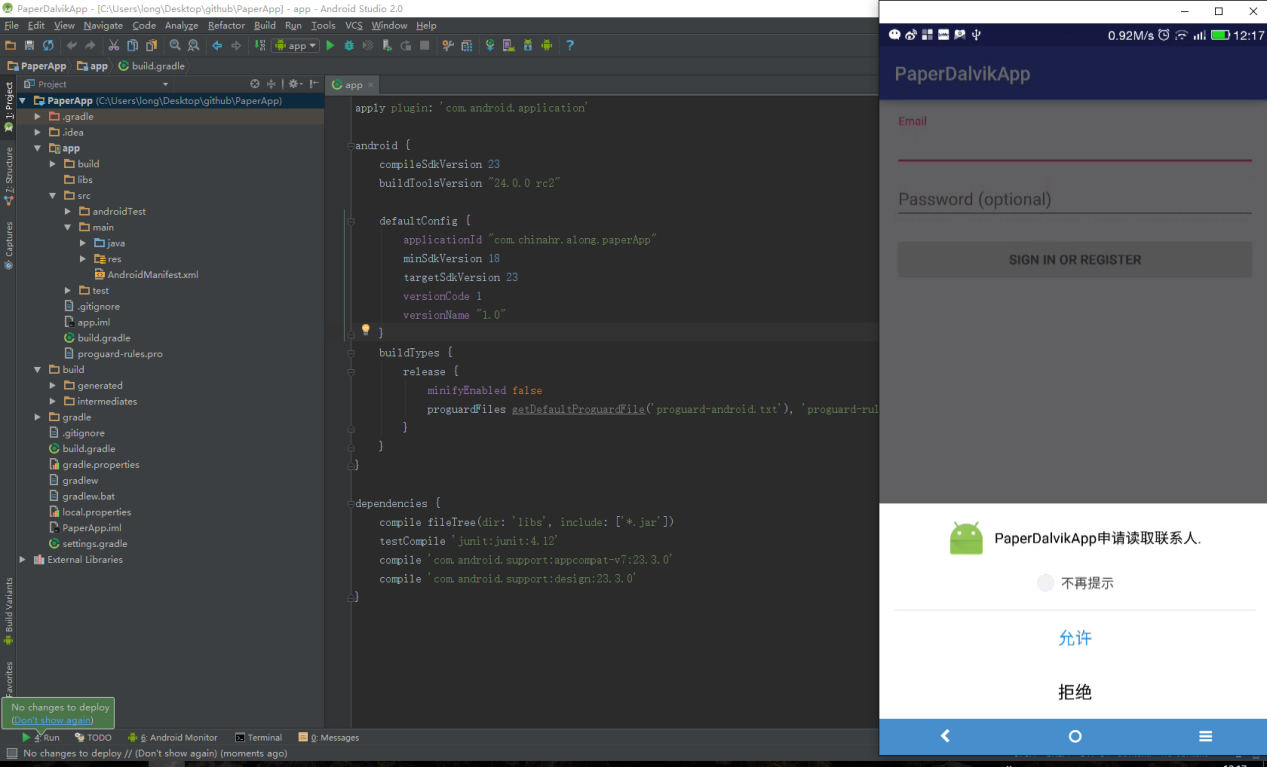
Android应用提供多个入口点。App由多个可以单独调用的组件组成。比如：一个单独的Activity提供一个UI，一个service在后台运行。在一个组件中通过intent启动另一个组件（可以是不同应用程序的组件）。这种模式对于单一的应用程序提供多个入口点，并默认允许其他app调用。

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。

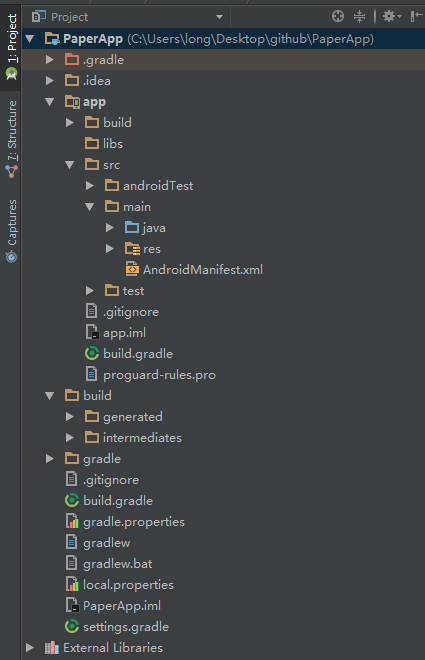
正常的工作流是这样的：创建，编译，测试，打包。

# 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cllass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。

# 开机

当你按下电源开关后Android设备执行了以下步骤：

1. 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。

1. 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。

1. 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。

1. 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。在这个阶段你可以在设备的屏幕上看到“Android”logo了。

1. 创建Zygote

Android系统在启动的时候，会创建一个Zygote进程，Zygote预加载以及初始化核心库类。通常，这些核心类一般是只读的，也是Android SDK或者核心框架的一部分。充当应用程序进程孵化器。Zygote进程在启动的过程中，又会创建一个Dalvik虚拟机。Zygote进程是通过复制自己来创建新的应用程序进程的。这意味着Zygote进程会将自己的Dalvik虚拟机复制给应用程序进程。通过这种方式就可以大大地提高应用程序的启动速度，因为这种方式避免了每一个应用程序进程在启动的时候都要去创建一个Dalvik。事实上，Zygote进程通过自我复制的方式来创建应用程序进程，省去的不仅仅是应用程序进程创建Dalvik虚拟机的时间，还能省去应用程序进程加载各种系统库和系统资源的时间，因为它们在Zygote进程中已经加载过了，并且也会连同Dalvik虚拟机一起复制到应用程序进程中去。(Zygote加载进程

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。

1. 启动系统服务或服务

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。

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）。

1. 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。

1. 安装系统应用

系统启动后，通过PackageManagerService来安装APK，系统除了会对/system/app和/data/app目录下的所有APK进行dex字节码的优化，还会对/system/framework目录下的APK或者JAR文件，以及这些APK所引用的外部JAR，进行dex字节码优化。所谓的“优化”之后会详细介绍，并且是这篇文章的重点。

Android系统在安装的过程，PackageManagerService会通过另外一个类Installer的成员函数dexopt来对APK里面的dex字节码进行优化：

public final class Installer {

......

public int dexopt(String apkPath, int uid, boolean isPublic) {

StringBuilder builder = new StringBuilder("dexopt");

builder.append(' ');

builder.append(apkPath);

builder.append(' ');

builder.append(uid);

builder.append(isPublic ? " 1" : " 0");

return execute(builder.toString());

}

......

}

# 安装应用

将一个apk发送或者下载到设备后，系统会执行一下几个操作：

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。

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、二进制的so动态库等。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。然后通过java层的DexFile去执行加载操作。

1. 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。

1. 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。

启动运行时。

Zygote进程中的Dalvik虚拟机是从AndroidRuntime::start开始创建。主要做了三件事： 创建一个JniInvocation实例，并且调用它的成员函数init来初始化JNI环境；调用AndroidRuntime类的成员函数startVm来创建一个虚拟机及其对应的JNI接口，即创建一个JavaVM接口和一个JNIEnv接口；有了上述的JavaVM接口和JNIEnv接口之后，就可以在Zygote进程中加载指定的class。

void AndroidRuntime::start(const char\* className, const char\* options)

{

......

/\* start the virtual machine \*/

JniInvocation jni\_invocation;

jni\_invocation.Init(NULL);

JNIEnv\* env;

if (startVm(&mJavaVM, &env) != 0) {

return;

}

......

/\*

\* Start VM. This thread becomes the main thread of the VM, and will

\* not return until the VM exits.

\*/

char\* slashClassName = toSlashClassName(className);

jclass startClass = env->FindClass(slashClassName);

if (startClass == NULL) {

ALOGE("JavaVM unable to locate class '%s'\n", slashClassName);

/\* keep going \*/

} else {

jmethodID startMeth = env->GetStaticMethodID(startClass, "main",

"([Ljava/lang/String;)V");

if (startMeth == NULL) {

ALOGE("JavaVM unable to find main() in '%s'\n", className);

/\* keep going \*/

} else {

env->CallStaticVoidMethod(startClass, startMeth, strArray);

#if 0

if (env->ExceptionCheck())

threadExitUncaughtException(env);

#endif

}

}

......

}

JniInvocation类的成员函数init所做的事情很简单。它首先是读取系统属性persist.sys.dalvik.vm.lib的值。前面提到，系统属性persist.sys.dalvik.vm.lib的值要么等于libdvm.so，要么等于libart.so。因此，接下来通过函数dlopen加载到进程来的要么是libdvm.so，要么是libart.so。无论加载的是哪一个so，都要求它导出JNI\_GetDefaultJavaVMInitArgs、JNI\_CreateJavaVM和JNI\_GetCreatedJavaVMs这三个接口，并且分别保存在JniInvocation类的三个成员变量JNI\_GetDefaultJavaVMInitArgs\_、JNI\_CreateJavaVM\_和JNI\_GetCreatedJavaVMs\_中。这三个接口也就是前面提到的用来抽象Java虚拟机的三个接口。

#ifdef HAVE\_ANDROID\_OS

static const char\* kLibrarySystemProperty = "persist.sys.dalvik.vm.lib";

#endif

static const char\* kLibraryFallback = "libdvm.so";

bool JniInvocation::Init(const char\* library) {

#ifdef HAVE\_ANDROID\_OS

char default\_library[PROPERTY\_VALUE\_MAX];

property\_get(kLibrarySystemProperty, default\_library, kLibraryFallback);

#else

const char\* default\_library = kLibraryFallback;

#endif

if (library == NULL) {

library = default\_library;

}

handle\_ = dlopen(library, RTLD\_NOW);

if (handle\_ == NULL) {

if (strcmp(library, kLibraryFallback) == 0) {

// Nothing else to try.

ALOGE("Failed to dlopen %s: %s", library, dlerror());

return false;

}

// Note that this is enough to get something like the zygote

// running, we can't property\_set here to fix this for the future

// because we are root and not the system user. See

// RuntimeInit.commonInit for where we fix up the property to

// avoid future fallbacks. http://b/11463182

ALOGW("Falling back from %s to %s after dlopen error: %s",

library, kLibraryFallback, dlerror());

library = kLibraryFallback;

handle\_ = dlopen(library, RTLD\_NOW);

if (handle\_ == NULL) {

ALOGE("Failed to dlopen %s: %s", library, dlerror());

return false;

}

}

if (!FindSymbol(reinterpret\_cast<void\*\*>(&JNI\_GetDefaultJavaVMInitArgs\_),

"JNI\_GetDefaultJavaVMInitArgs")) {

return false;

}

if (!FindSymbol(reinterpret\_cast<void\*\*>(&JNI\_CreateJavaVM\_),

"JNI\_CreateJavaVM")) {

return false;

}

if (!FindSymbol(reinterpret\_cast<void\*\*>(&JNI\_GetCreatedJavaVMs\_),

"JNI\_GetCreatedJavaVMs")) {

return false;

}

return true;

}

AndroidRuntime类的成员函数startVm最主要就是调用函数JNI\_CreateJavaVM来创建一个JavaVM接口及其对应的JNIEnv接口。

extern "C" jint JNI\_CreateJavaVM(JavaVM\*\* p\_vm, JNIEnv\*\* p\_env, void\* vm\_args) {

return JniInvocation::GetJniInvocation().JNI\_CreateJavaVM(p\_vm, p\_env, vm\_args);

}

JniInvocation类的静态成员函数GetJniInvocation返回的便是前面所创建的JniInvocation实例。有了这个JniInvocation实例之后，就继续调用它的成员函数JNI\_CreateJavaVM来创建一个JavaVM接口及其对应的JNIEnv接口。

jint JniInvocation::JNI\_CreateJavaVM(JavaVM\*\* p\_vm, JNIEnv\*\* p\_env, void\* vm\_args) {

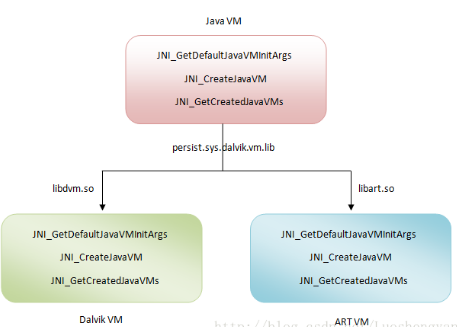
return JNI\_CreateJavaVM\_(p\_vm, p\_env, vm\_args);

}

 JniInvocation类的成员变量JNI\_CreateJavaVM\_指向的就是前面所加载的libdvm.so或者libart.so所导出的函数JNI\_CreateJavaVM，因此，JniInvocation类的成员函数JNI\_CreateJavaVM返回的JavaVM接口指向的要么是Dalvik虚拟机，要么是ART虚拟机。

通过上面的分析，我们就很容易知道，Android系统通过将ART运行时抽象成一个Java虚拟机，以及通过系统属性persist.sys.dalvik.vm.lib和一个适配层JniInvocation，就可以无缝地将Dalvik虚拟机替换为ART运行时。这个替换过程设计非常巧妙，因为涉及到的代码修改是非常少的。

以上就是ART虚拟机的启动过程，接下来分析应用程序在安装过程中将dex字节码翻译为本地机器码的过程。



从图可以知道，Dalvik虚拟机和ART虚拟机都实现了三个用来抽象Java虚拟机的接口：

1. JNI\_GetDefaultJavaVMInitArgs -- 获取虚拟机的默认初始化参数

2. JNI\_CreateJavaVM -- 在进程中创建虚拟机实例

3. JNI\_GetCreatedJavaVMs -- 获取进程中创建的虚拟机实例

在Android系统中，Davik虚拟机实现在libdvm.so中，ART虚拟机实现在libart.so中。也就是说，libdvm.so和libart.so导出了JNI\_GetDefaultJavaVMInitArgs、JNI\_CreateJavaVM和JNI\_GetCreatedJavaVMs这三个接口，供外界调用。 此外，Android系统还提供了一个系统属性persist.sys.dalvik.vm.lib，它的值要么等于libdvm.so，要么等于libart.so。当等于libdvm.so时，就表示当前用的是Dalvik虚拟机，而当等于libart.so时，就表示当前用的是ART虚拟机。

以上描述的Dalvik虚拟机和ART虚拟机的共同之处。不同的地方就在于，Dalvik虚拟机执行的是dex字节码，ART虚拟机执行的是本地机器码。这意味着Dalvik虚拟机包含有一个解释器，用来执行dex字节码，当然，Android从2.2开始，也包含有JIT（Just-In-Time），用来在运行时动态地将执行频率很高的dex字节码翻译成本地机器码，然后再执行。通过JIT，就可以有效地提高Dalvik虚拟机的执行效率。但是，将dex字节码翻译成本地机器码是发生在应用程序的运行过程中的，并且应用程序每一次重新运行的时候，都要做重做这个翻译工作的。因此，即使用采用了JIT，Dalvik虚拟机的总体性能还是不能与直接执行本地机器码的ART虚拟机相比。

那么，ART虚拟机执行的本地机器码是从哪里来的呢？Android的运行时从Dalvik虚拟机替换成ART虚拟机，并不要求开发者要将重新将自己的应用直接编译成目标机器码。也就是说，开发者开发出的应用程序经过编译和打包之后，仍然是一个包含dex字节码的APK文件。而ART虚拟机需要的是本地机器码，这就必然要有一个翻译的过程。这个翻译的过程当然不能发生应用程序运行的时候，否则的话就和Dalvik虚拟机的JIT一样了。在计算机的世界里，与JIT相对的是AOT。AOT进Ahead-Of-Time的简称，它发生在程序运行之前。静态语言（例如C/C++）开发的应用程序，编译器直接就把它们翻译成目标机器码。这种静态语言的编译方式也是AOT的一种。但是前面我们提到，ART虚拟机并不要求开发者将自己的应用直接编译成目标机器码。这样，将应用的dex字节码翻译成本地机器码的最恰当AOT时机就发生在应用安装的时候。

没有ART虚拟机之前，应用在安装的过程，其实也会执行一次“翻译”的过程。只不过这个“翻译”的过程是将dex字节码进行优化，也就是由dex文件生成odex文件。这个过程由安装服务PackageManagerService请求守护进程installd来执行的。从这个角度来说，在应用安装的过程中将dex字节码翻译成本地机器码对原来的应用安装流程基本上就不会产生什么影响。

# 加载器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;

}

1. 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;

}

1. 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;

}

1. 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;

}

1. 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) {

DexFile dex = element.dexFile;

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;

}

1. 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);

}

1. 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);

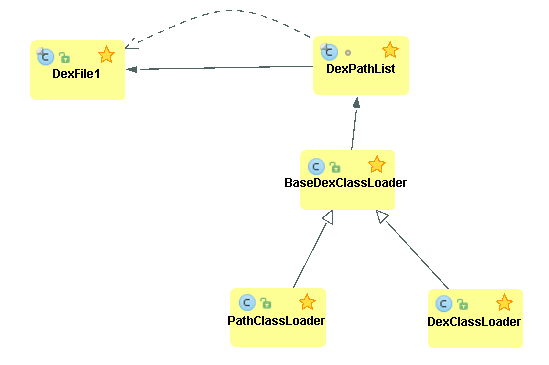
}

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。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

如何实现上述过程？

首先，有必要说明一下java中化腐朽为神奇的工具--反射。这个工具贯穿整个框架的搭建。

Java 反射机制在程序运行时，对于任意一个类或者对象，都能够知道这个类的所有属性和方法(包括私有，静态，final等任任意属性方法)，这种动态获取的信息以及动态调用对象的方法的功能称为 java 的反射机制。

以下获取属性方法或者给属性赋值，都是借助反射。读到这里，可以发现前面的ClassLoader就是反射的核心实现。

1. 自定义DexClassLoader，指定文件名与缓存odex的目录，并且可以还可以指定需要替换的nativfe Lib的目录。

DexClassLoader cl = new DexClassLoader(outputPath + File.separator + "test.jar",

outputPath, null, context.getClassLoader().getParent());

1. 获取当前系统的PathClassLoader和DexClassLoader的DexElements。

Object baseDexElements = getDexElements(getPathList(context.getClassLoader()));

Object newDexElements = getDexElements(getPathList(cl));

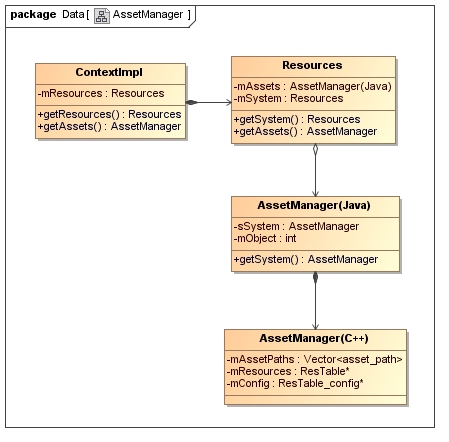
1. 合并两个数组，并赋值给PathClassLoader.

Object pathList = getPathList(context.getClassLoader());

ReflectionUtils.setField(pathList, pathList.getClass(), "dexElements", allDexElements);

# 三个关键问题

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。另外，最重要的一点：生命周期如何控制。目前还真的没什么办法能够处理这个问题，一个Activity的启动，如果不采用标准的Intent方式，没有经历过Android系统Framework层级的一系列初始化和注册过程，它的生命周期方法是不会被系统调用的（除非你能够修改Android系统的一些代码，而这已经是另一个领域的话题了，这里不展开）。那把插件APK里所有Activity都注册到主项目的Manifest里，再以标准Intent方式启动。但是事先主项目并不知道插件Activity里会新增哪些Activity，如果每次有新加的Activity都需要升级主项目的版本，那不是本末倒置了，不如把插件的逻辑直接写到主项目里来得方便。那就绕绕弯吧，生命周期不就是系统对Activity一些特定方法的调用嘛，可以在主项目里创建一个ProxyActivity，再由它去代理调用插件Activity的生命周期方法（这也是代理模式叫法的由来）。用ProxyActivity（一个标准的Activity实例）的生命周期同步控制插件Activity（普通类的实例）的生命周期，同步的方式可以有下面两种：在ProxyActivity生命周期里用反射调用插件Activity相应生命周期的方法，简单粗暴。把插件Activity的生命周期抽象成接口，在ProxyActivity的生命周期里调用。另外，多了这一层接口，也方便主项目控制插件Activity。这里补充说明下，Fragment自带生命周期，用Fragment来代替Activity开发可以省去大部分生命周期的控制工作，但是会使得界面跳转比较麻烦，而且Honeycomb以前没有Fragment，无法在API11以前的系统使用。
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。解决Res资源加载的一种方式：首先将插件中的资源解压出来，然后通过文件流去读取资源，这样做理论上是可行的，但是实际操作起来还是有很大难度 的。首先不同资源有不同的文件流格式，比如图片、XML等，其次针对不同设备加载的资源可能是不一样的，如何选择合适的资源也是一个需要解决的问题，基于这两点，这种方法也不建议使用，因为它实现起来有较大难度。为了方便地对插件进行资源管理，下面给出一种合理的方式。一个进程是可以同时加载多个应用程序的，也就是可以同时加载多个APK文件。每一个APK文件在进程中都对应有一个全局的Resourses对象以及一个全局的AssetManager对象。其中，这个全局的Resourses对象保存在一个对应的ContextImpl对象的成员变量mResources中，而这个全局的AssetManager对象保存在这个全局的Resourses对象的成员变量mAssets中。上述ContextImpl、Resourses和AssetManager的关系如图所示：

****Resources类有一个成员函数getAssets，通过它就可以获得保存在Resources类的成员变量mAssets中的AssetManager，例如，ContextImpl类的成员函数getAssets就是通过调用其成员变量mResources所指向的一个Resources对象的成员函数getAssets来获得一个可以用来访问应用程序的非编译资源文件的AssetManager。

Android应用程序除了要访问自己的资源之外，还需要访问系统的资源。系统的资源打包在/system/framework/framework-res.apk文件中，它在应用程序进程中是通过一个单独的Resources对象和一个单独的AssetManager对象来管理的。这个单独的Resources对象就保存在Resources类的静态成员变量mSystem中，我们可以通过Resources类的静态成员函数getSystem就可以获得这个Resources对象，而这个单独的AssetManager对象就保存在AssetManager类的静态成员变量sSystem中，我们可以通过AssetManager类的静态成员函数getSystem同样可以获得这个AssetManager对象。

AssetManager类除了在Java层有一个实现之外，在 C++层也有一个对应的实现，而Java层的AssetManager类的功能就是通过C++层的AssetManager类来实现的。Java层的每一个AssetManager对象都有一个类型为int的成员变量mObject，它保存的便是在C++层对应的AssetManager对象的地址，因此，通过这个成员变量就可以将Java层的AssetManager对象与C++层的AssetManager对象关联起来。

C++层的AssetManager类有三个重要的成员变量mAssetPaths、mResources和mConfig。其中，mAssetPaths保存的是资源存放目录，mResources指向的是一个资源索引表，而mConfig保存的是设备的本地配置信息，例如屏幕密度和大小、国家地区和语言等等配置信息。有了这三个成员变量之后，C++层的AssetManager类就可以访问应用程序的资源了。

，主要就是创建和初始化用来访问应用程序资源的AssetManager对象和Resources对象，其中，初始化操作包括设置AssetManager对象的资源文件路径以及设备配置信息等。有了这两个初始化的AssetManager对象和Resources对象之后，就可以查找应用程序资源了。

创建一个AssetManager，执行addAssetPath方法将资源文件路径加载进去。最后实例化一个Resource。

try {

AssetManager assetManager = AssetManager.class.newInstance();

Method addAssetPath = assetManager.getClass().getMethod("addAssetPath", String.class);

addAssetPath.invoke(assetManager, mDexPath);

mAssetManager = assetManager;

} catch (Exception e) {

e.printStackTrace();

}

Resources superRes = super.getResources();

mResources = new Resources(mAssetManager, superRes.getDisplayMetrics(),

superRes.getConfiguration());

1. 虚拟机在安装app期间为了提高性能：当一个类只引用当前dex的文件，就会打上一个CLASS\_ISPREVERIFIED标志。在运行的时候如果使用另一个dex里面的文件就会抛出异常，直接崩溃！解决这个问题的唯一方法就是防止类被打上这个标志。也就是在目标类上面引用其他dex里面的类就可以解决这个问题。方法很简单：先创建一个java文件，编译成class文件，打包成单独的dex。之后采用javassist在目标类的构造方法中调用该dex文件中的类。

# 管理后台

更多的时候希望可以在线下载补丁，并且在补丁有新版本的时候，app要从服务器下载最新的补丁替换本地已经存在的旧补丁。为此，我们应该有一个管理后台，它大概有以下功能：

1. 上传不同版本的补丁，并向APP项目提供补丁信息查询功能和下载功能。
2. 管理在线的补丁，并能向不同版本号的APP主项目提供最合适的补丁patch。
3. 万一最新的补丁出现紧急BUG，要提供旧版本回滚功能。
4. 出于安全考虑应该对APP项目的请求信息做一些安全性校验。

# 代码安全

有一个非常有借鉴意义的执行代码安全方案：将所有的dex,so,资源打包为apk，然后交给PatchManager--这种方案好像在哪里见过！就是Android系统的执行文件apk。

一切又回到了原点。细细品味，意味深长！