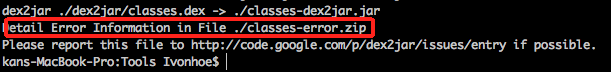
Android安全之应用防dex2jar原理及实现

<https://ivonhoe.github.io/2017/02/09/%E7%BE%8E%E5%9B%A2%E5%A6%82%E4%BD%95%E9%98%B2dex2jar/>

### 一、反编译某平台代码

最近在看某外卖平台的代码，发现某外卖平台最新版本版本无法正常的通过dex2jar工具将dex转换出Java源代码，在转换过程中会提示出错，如图：

[](https://ivonhoe.github.io/res/dex2jar_error/dex2jar_error1.png)

查看转换出的Java源代码，会发现很多类方法提示下图所示异常，很多方法中都会抛出RuntimeException：can not merge I and Z:

|  |  |
| --- | --- |
| 1 | public class AsyncTaskService extends IntentService {    protected void onHandleIntent(Intent paramIntent){  throw new RuntimeException("d2j fail translate: java.lang.RuntimeException: can not merge I and Z\n\tat...);  }    public int onStartCommand(Intent paramIntent, int paramInt1, int paramInt2){  throw new RuntimeException("d2j fail translate: java.lang.RuntimeException: can not merge I and Z\n\tat...");  }  } |

查看日志文件会发现很多类似的错误信息，可以看到方法内RuntimeException栈信息和反编译的错误信息是相同的，都提示can not merge I and Z。

[](https://ivonhoe.github.io/res/dex2jar_error/dex2jar_error2.png)

### 二、为什么？

本来以为这是dex2jar工具低版本的一个bug，但更新了dex2jar以后，依然还是会出现上述错误。  
java.lang.RuntimeException: can not merge I and Z这个异常，在[sourceforge](https://sourceforge.net/p/dex2jar/tickets/238/)上解释的比较清楚，其实是一个dex2jar工具检查出的一个参数异常”，**The problem is caused by strict type calculation, because in java syntaxt, a boolean can not assign to an inteager. so dex2jar forbid merge type Z and I.** 你用布尔类型入参调用一个参数为整型的函数，当然会检查出错，为啥这么说，我使用apktool工具，看了一下apk的smali代码。发现报错的函数的最前面都含有一段奇怪smali的代码：

|  |  |
| --- | --- |
| 1 | invoke-static {}, Lpnf/this/object/does/not/Exist;->a()Z  move-result v0  invoke-static {v0}, Lpnf/this/object/does/not/Exist;->b(I)V |

看上面的代码，pnf.this.object.does.not.Exist.a()方法返回一个boolean类型数据，放入v0寄存器，作为pnf.this.object.does.not.Exist.b(int)函数的入参。正常情况下这样的语法错误在java代码编译时就不会通过的。看到这里你会不会想，如果我不想别人直接看到我的Java代码，是不是可以通过在核心函数中插入上面这段有语法错误的代码，以达到dex2jar工具检查出错的目的呢？从而将代码被阅读的门槛从java提高到smali。

### 三、手动代码注入

为了验证上面的猜想，这里我通过反编译一个apk，手动插入有语法错误的smali代码，以验证防dex2jar的思路，具体步骤如下：

* 1.反编译一个apk。
* 2.修改smali代码，插入上面这三句有语法错误的代码。
* 3.重打包，使用dex2jar工具转换新包的dex，看是否能正常转换出Java源代码。并检查运行时是否出错。

我这里用一个Hello World应用来测试，使用apktool反编译出smali代码，并在Application的onCreate方法中插入这段有语法错误的代码。

|  |  |
| --- | --- |
| 1 | # virtual methods  .method public onCreate()V  .locals 3  invoke-super {p0}, Lcn/trinea/android/lib/h/c;->onCreate()V  invoke-virtual {p0}, Lcn/trinea/android/developertools/MyApplication;->getApplicationContext()Landroid/content/Context;  move-result-object v0  invoke-static {}, Lpnf/object/does/not/Exist;->a()Z  move-result v3  invoke-static {v3}, Lpnf/object/does/not/Exist;->b(I)V  return-void  .end method |

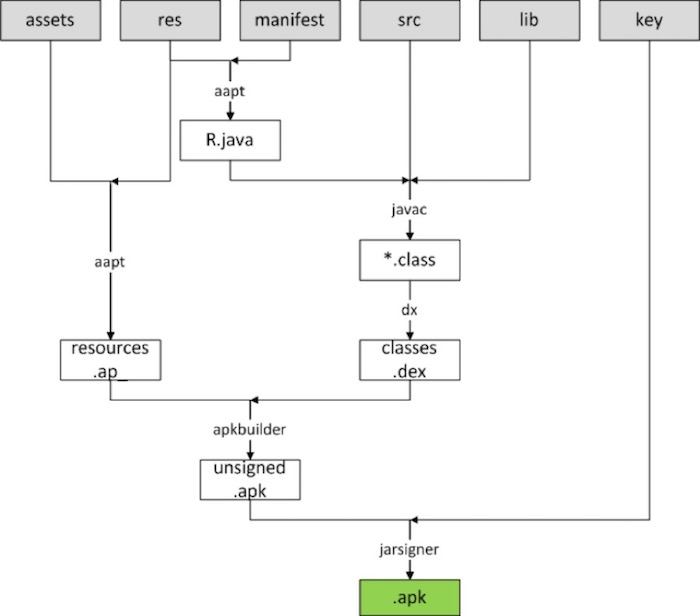
这里不要忘了，你可能需要另外编译出Exist.smali这个文件，不然运行时一定会爆出ClassNotFound异常。将下面的Exist.java**编译出Exist.smali**放入相应的包路径，重打包就可以了。Java代码如下：

|  |  |
| --- | --- |
| 1 | public class Exist {  public static boolean a() {  return false;  }  public static void b(int test) {  }  } |

最后，验证下果然重新打包后的apk，确实不能正常转换出Java源代码，这里就不贴图了，因为转换出错日志是一样的。并且运行时也不会出错。接下来会写一个Gradle编译插件，针对特定的函数，插入代码，防止dex2jar工具查看Java源代码。

### 四、实现思路

Android客户端在防止其Java代码被dex2jar转换时其实就是借助dex2jar的语法检查机制，将有语法错误的字节码插入到想要保护的Java函数中里面，以达到dex2jar转换出错的目的。接下来我就大致记录下如何开发Gradle编译插件，在编译过程中实现上述防护思路，先看下Android APK打包流程：

[](https://ivonhoe.github.io/res/dexguard/dexguard4.jpeg)

Android APK打包流程如上图所示，Java代码先通过Java Compiler生成.class文件，再通过dx工具生成dex文件，最后使用apkbuilder工具完成代码与资源文件的打包，并使用jarsigner签名，最后可能还有使用zipalign对签名后的apk做对齐处理。

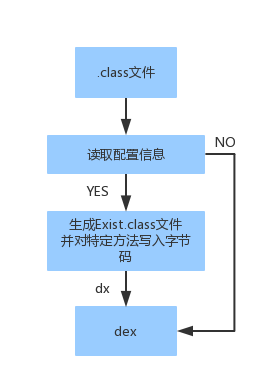
如果需要完成对特定函数的代码注入，可以在Java代码编译生成class文件后，在dex文件生成前，针对class字节码进行操作，以本例为例需要动态生成Exsit类文件的字节码。

|  |  |
| --- | --- |
| 1 | // 动态生成Exist.class  public class Exist {  public static boolean a() {  return false;  }  public static void b(int test) {  }  } |

将下列Java代码转换成字节码插入需要保护的函数中。

|  |  |
| --- | --- |
| 1 | // 插入到特定的Java函数内  Exist.b(Exist.a()); |

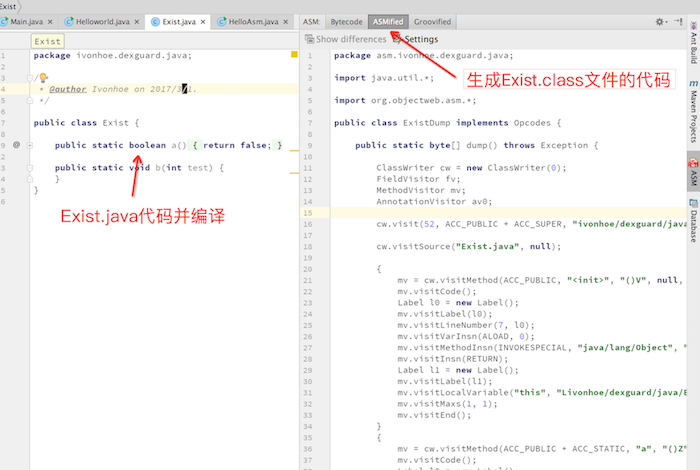
并将修改后的.class文件放入dex打包目录中，完成dex打包，具体流程如下图所示：

[](https://ivonhoe.github.io/res/dexguard/dexguard6.png)

Gradle提供了叫Transform的API，允许三方插件在class文件转换为dex文件前操作编译好的class文件，这个API的目标就是简化class文件的自定义的操作而不用对Task进行处理，并且可以更加灵活地进行操作。详细的可以参考[区长的博客](http://blog.csdn.net/sbsujjbcy/article/details/50839263)。

### 五、使用ASM操作Java字节码

ASM 是一个 Java 字节码操控框架。它能被用来动态生成类或者增强既有类的功能。ASM 可以直  
接产生二进制 class 文件，也可以在类被加载入 Java 虚拟机之前动态改变类行为。这里推荐一个IDEA插件:ASM ByteCode Outline。可以查看.class文件的字节码，并可以生成成ASM框架代码。安装ASM Bytecode Outline插件后，可以在Intellij IDEA->Code->Show Bytecode Outline查看类文件对应个字节码和ASM框架代码，利用ASM框架代码就可以生成相应的.class文件了。

[](https://ivonhoe.github.io/res/dexguard/dexguard3.png)

生成Exist字节码的具体实现，生成Exist.java的构造函数：

|  |  |
| --- | --- |
| 1 | ClassWriter cw = new ClassWriter(0);  FieldVisitor fv;  MethodVisitor mv;  AnnotationVisitor av0;  cw.visit(51, ACC\_PUBLIC + ACC\_SUPER, "ivonhoe/dexguard/java/Exist", null, "java/lang/Object", null);  cw.visitSource("Exist.java", null);  mv = cw.visitMethod(ACC\_PUBLIC, "<init>", "()V", null, null);  mv.visitCode();  Label l0 = new Label();  mv.visitLabel(l0);  mv.visitLineNumber(7, l0);  mv.visitVarInsn(ALOAD, 0);  mv.visitMethodInsn(INVOKESPECIAL, "java/lang/Object", "<init>", "()V", false);  mv.visitInsn(RETURN);  Label l1 = new Label();  mv.visitLabel(l1);  mv.visitLocalVariable("this", "Livonhoe/dexguard/java/Exist;", null, l0, l1, 0);  mv.visitMaxs(1, 1);  mv.visitEnd(); |

声明一个函数名为a，返回值为boolean类型的无参函数：

|  |  |
| --- | --- |
| 1 | mv = cw.visitMethod(ACC\_PUBLIC + ACC\_STATIC, "a", "()Z", null, null);  mv.visitCode();  l0 = new Label();  mv.visitLabel(l0);  mv.visitLineNumber(10, l0);  mv.visitInsn(ICONST\_0);  mv.visitInsn(IRETURN);  mv.visitMaxs(1, 0);  mv.visitEnd(); |

声明一个函数名为b，参数为int型，返回类型为void的函数

|  |  |
| --- | --- |
| 1 | MV = CW.VISITmETHOD(acc\_public + acc\_static, "b", "(i)v", NULL, NULL);  MV.VISITcODE();  L0 = NEW lABEL();  MV.VISITlABEL(L0);  MV.VISITlINEnUMBER(14, L0);  MV.VISITiNSN(return);  L1 = NEW lABEL();  MV.VISITlABEL(L1);  MV.VISITlOCALvARIABLE("TEST", "i", NULL, L0, L1, 0);  MV.VISITmAXS(0, 1);  MV.VISITeND(); |

在指定函数内，插入Exist.b(Exist.a());对应的字节码的具体实现，绕过Java编译器的语法检查：

|  |  |
| --- | --- |
| 1 | static class InjectClassVisitor extends ClassVisitor {  private String methodName;  InjectClassVisitor(int i, ClassVisitor classVisitor, String method) {  super(i, classVisitor)  this.methodName = method;  }  @Override  public MethodVisitor visitMethod(int access, String name, String desc,  String signature, String[] exceptions) {  MethodVisitor mv = super.visitMethod(access, name, desc, signature, exceptions);  mv = new MethodVisitor(Opcodes.ASM4, mv) {  @Override  void visitCode() {  // 在方法体开始调用时  if (name.equals(methodName)) {  mv.visitMethodInsn(INVOKESTATIC, "ivonhoe/dexguard/java/Exist", "a", "()Z", false);  mv.visitMethodInsn(INVOKESTATIC, "ivonhoe/dexguard/java/Exist", "b", "(I)V", false);  }  super.visitCode()  }  @Override  public void visitMaxs(int maxStack, int maxLocal) {  if (name.equals(methodName)) {  super.visitMaxs(maxStack + 1, maxLocal);  } else {  super.visitMaxs(maxStack, maxLocal);  }  }  }  return mv;  }  } |

### 六、总结

看到这里可能你会有一个疑惑，为什么有语法错误的代码，在运行时不会出错，个人理解不单单是因为bool类型在内存中是以0或1表示，也因为int和bool在Android虚拟机中都存储在32位寄存器中，如果使用int和long类型的参数互换，在dx阶段的编译就会报错。下面是插件源码，有兴趣的同学可以尝试一下~

#### 插件源码

详细的Gradle源码和实例可参考<https://github.com/Ivonhoe/dexguard>

#### 使用方法

* 在root project的build.gradle中添加依赖classpath 'ivonhoe.gradle.dexguard:dexguard-gradle:0.0.2-SNAPSHOT'

|  |  |
| --- | --- |
| 1 | buildscript {  repositories {  maven { url 'https://raw.githubusercontent.com/Ivonhoe/mvn-repo/master/' }  mavenCentral()  }  dependencies {  classpath 'com.android.tools.build:gradle:2.3.0'  classpath 'ivonhoe.gradle.dexguard:dexguard-gradle:0.0.2-SNAPSHOT'  }  } |

* 在app项目的build.gradle中添加插件，map.txt中配置需要保护的方法名

|  |  |
| --- | --- |
| 1 | apply plugin: 'ivonhoe.dexguard'  dexguard {  guardConfig = "${rootDir}/map.txt"  } |

### 七、参考文档

* [#238 java.lang.RuntimeException: can not merge I and Z](https://sourceforge.net/p/dex2jar/tickets/238/)
* [Smali语法](http://www.jianshu.com/p/730c6e3e21f6)
* [Android 热修复使用Gradle Plugin1.5改造Nuwa插件](http://blog.csdn.net/sbsujjbcy/article/details/50839263)
* [ASM-操作字节码初探](http://www.wangyuwei.me/2017/01/20/ASM-%E6%93%8D%E4%BD%9C%E5%AD%97%E8%8A%82%E7%A0%81%E5%88%9D%E6%8E%A2/)
* [手摸手增加字节码往方法体内插代码](https://www.diycode.cc/topics/581)

### 转载请标明出处[病已blog https://ivonhoe.github.io/](https://ivonhoe.github.io/)