# Proguard

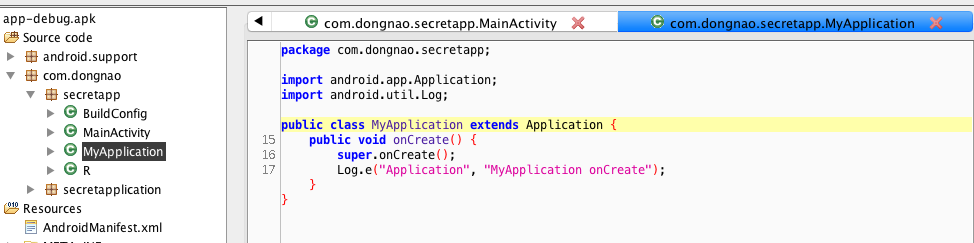
android app一般大部分的逻辑是使用java来开发，java编译出来class之后，再由dx工具编译出dex才能运行在dvm上。

所以正常来说一个app大部分的代码是在dex中，对于dex的安全问题，一直是android开发员重视的问题。

我们可能用过dex2jar、jadx、enjarify等等工具来反编译dex，我们在看别家app的代码的时候，心里肯定会想到自己写的app如果不进行一些处理，也是这么容易就能让别人看到。

我们可以试一下，使用jadx来反编译一个没有经过任何处理的app

<https://sourceforge.net/projects/jadx/files/>



可以看到我们的代码面对这些反编译工具而言没有任何的秘密。

所以在正式版中我们一般会引入proguard混淆来达到一定程度的代码保护。

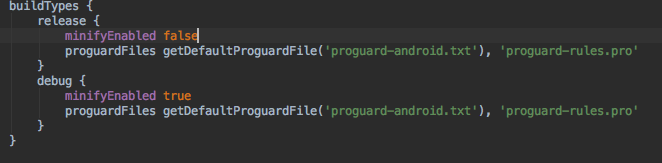
<https://developer.android.com/studio/build/shrink-code.html?hl=zh-cn>

Proguard是一个代码优化和混淆工具。

能够提供对Java类文件的压缩、优化、混淆，和预校验。压缩的步骤是检测并移除未使用的类、字段、方法和属性。优化的步骤是分析和优化方法的字节码。混淆的步骤是使用短的毫无意义的名称重命名剩余的类、字段和方法。压缩、优化、混淆使得代码更小，更高效。

所以proguard不只是用于混淆。

在Android Studio中我们能够很轻松的就启用这个工具。

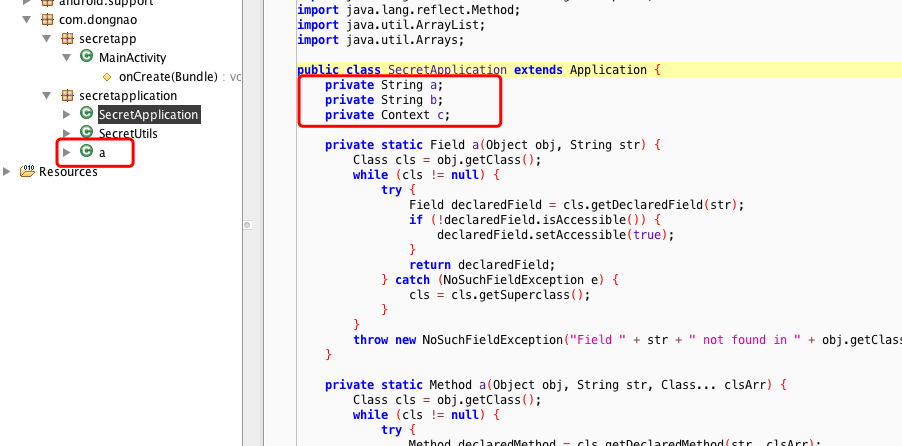


ProGuard 会移除所有（并且只会移除）未使用的代码。不过，ProGuard 难以对许多情况进行正确分析，可能会移除应用真正需要的代码。

比如需要反射、动态加载所引用的类等情况，可能因为proguard移除或者混淆了这部分没使用的类，而导致错误。所以有时需要编写混淆优化配置文件。在gradle中的proguardFiles 能够让我们传递File文件或者文件路径交给proguard来执行。

默认的getDefaultProguardFile(‘proguard-android.txt’) 会使用build/intermediates/下面的文件(可以println 查看)，如果在这个目录下没有则可以通过运行gradle任务：生成。而后面的’proguard-rules.pro’则是一个相对路径，相对于这个build.gradle同级目录下的proguard-rules.pro

让生成一个启用了优化混淆之后的新包再反编译查看有什么不同。



可以看到一个类的名字变成了无意义的a,而在一个类中里面声明的成员变量都变成了a/b/c这样的无意义的字符，同时v4和v7也被混淆了。那么为什么会这样？

查看build/intermediates/proguard-android.txt-3.0.0

所以对于混淆最重要的配置就是



可以看到keep一共有六个，并且可以分为两组，第二组是第一组加"names"。

-keep 指定类和类成员（变量和方法）不被混淆。

**-keep** class com.dongnao.proxy.guard.test.Bug

（保护了类名）

**-keep** class com.dongnao.proxy.guard.test.Bug{  
 **public static void \*();** }

（保护了 public static void的没有参数的函数）

**-keep** class com.dongnao.proxy.guard.test.Bug{  
 **\*；**

}

(保护所有)

-keepclassmembers 指定类成员不被混淆(就是-keep的缩小版，不管类名了)。

**-keepclassmembers**

**class** com.dongnao.proxy.guard.test.Bug

（都被混淆了）

-keepclasseswithmembers 指定类和类成员不被混淆，前提是指定的类成员存在。

**-keepclasseswithmembers** class com.dongnao.proxy.guard.test.Bug

(保护类名，但是没指定成员，所以函数名被混淆)

**-keepclasseswithmembers** class com.dongnao.proxy.guard.test.Bug{  
 **native <methods>;** }

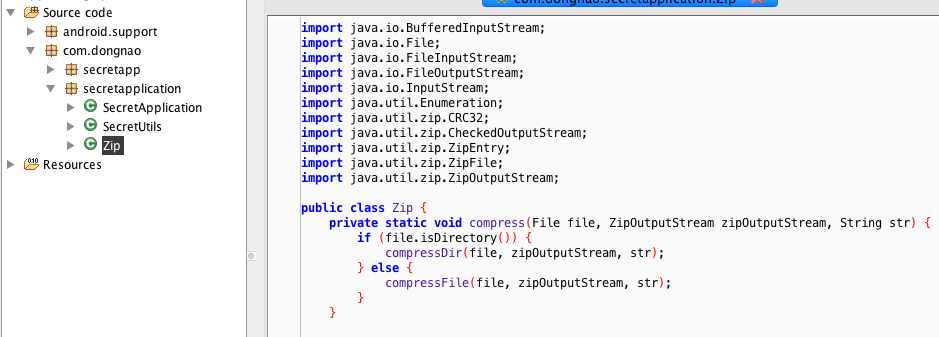
全被混淆了。注意 前提是指定的类成员存在 因为不存在native函数，所以这条语句等于无效，

添加names的三个和上面的三个对应，没有加的是From being removed or renamed，即保持不被移除和重命名；加的是From being renamed，即保持不被重命名。两者的区别就是是否保持不被移除，前者如果指定了不被混淆和移除，那么即使这个类或类成员有无用的也会被保留下来；后者如果是无用的类或类成员，即使指定了不被混淆那么也会被移除的。

我们如果需要对我们的某些代码进行保护就需要来通过keep来指定。

比如我们刚刚被混淆的Zip与support-v4

然后再通过jdx查看:



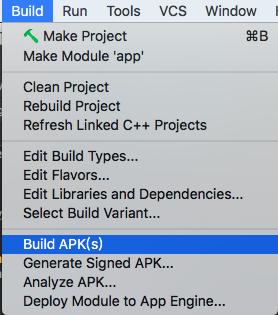
现在都没有混淆了。

我们发布的apk正常都是会经过proguard处理过后的。然后我们看到一个问题，在经过混淆之后，我们的类名变成了a/b/c这样的，那如果我们集成了类似友盟、bugly等异常上报功能，那么我们怎么知道这个异常到底出现在哪里？

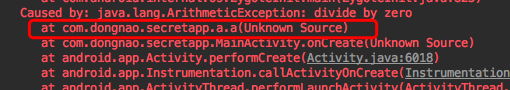
我们在Activity中主动写一个bug



然后生成一个debug的apk



运行产生这样的日志:

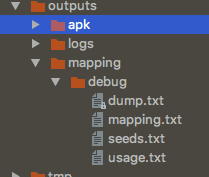


这个日志有两个问题：

1. 我们不知道是哪个类报告出的。
2. 不知道这个类是哪一行报出的。

第一个问题很显然是由于我们的代码进行了混淆。

在开启混淆后，我们生成apk会产生



dump.txt

说明 APK 中所有类文件的内部结构。

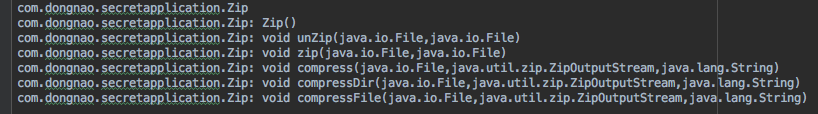
这个文件内容非常多，可读性也并不高

mapping.txt

提供原始与混淆过的类、方法和字段名称之间的转换。

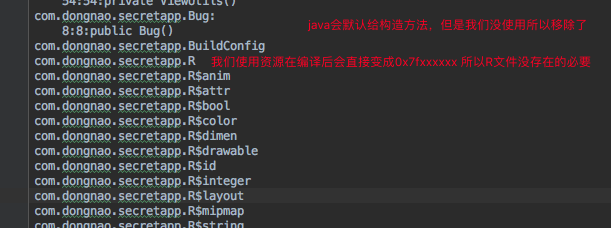
seeds.txt

列出未进行混淆的类和成员。我们保护的zip没有被混淆



usage.txt

列出从 APK 移除的代码。可以在这个文件中查看是不是有我们不想被移除的类



最重要的当然就是我们的mapping文件了。通过这个文件我们就能定位到:



出现错误的函数是Bug类中的test函数。

我们现在比较简单，所以一下子就能定位到错误函数，如果函数层级较深，光靠自己比对来查找难度就比较大了。所以sdk中提供了一个工具在tools/proguard/bin中。

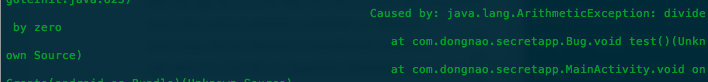
我们把logcat里面保存的错误堆栈复制到一个文件中：



然后执行retrace 脚本（在 Windows 上为 retrace.bat；在 Mac/Linux 上为 retrace.sh）

retrace.bat|retrace.sh [-verbose] mapping.txt [<stacktrace\_file>]





现在错误日志一句mapping还原了。

所以我们每次在发布版本之后都需要保留这个mapping文件。所以一般异常上报平台都会提供mapping上传然后帮助我们分析。

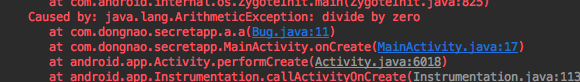
第一个问题解决的，第二个问题是行数显示的是Unknow Source



如果希望出现具体行数,我们需要在配置文件中加入

抛出异常时保留代码行号，在异常分析中可以方便定位

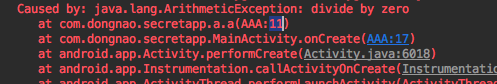
-keepattributes SourceFile,LineNumberTable



另外还可以配合

-renamesourcefileattribute AAA

使用字符串"AAA"来替代真正的类，避免泄漏更多的信息



虽然我们的代码经过了混淆，但是实际上，对于有耐心有条件的人来说，还是能够从混淆过后的代码中看到蛛丝马迹。

Jdx-gui工具可能对做到这一点的帮助不大，因为它不支持函数的跳转。现在我换一种工具来进行反编译。

dex2jar: java

<https://sourceforge.net/projects/dex2jar/>

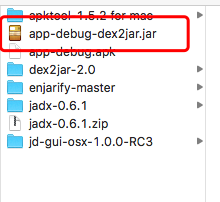
enjarify: python3

<https://github.com/google/enjarify>

两种工具都可以,使用dex2jar

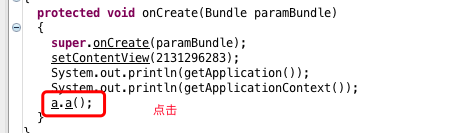


反编译后生成了一个jar

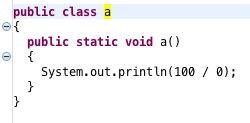


然后使用jd-gui打开：

<http://jd.benow.ca/>



直接跳到了



更加方便。

所以只要花功夫，混淆完全能够被完整的解读出来。当然并不是说混淆就没用。混淆之后解读难度更大了。

proguard移除无用代码还能缩小apk大小。