KM上有很多关于Dex 65536这个大坑的分析与建议，这里将带来新的方案——压缩方法。

**何为压缩方法：**

把多个方法合并成一个方法，这样可以在一定程度上减少方法的数量。

**流行的做法：**

一般在Release版本中，都会集成proguard或者dexguard进行优化。

proguard会对类名、方法名、属性名等进行混淆，同时去除一些无用的方法等等，从而在一定程度减少的方法的数量。

dexguard则是proguard的收费版，其在proguard的功能基础上还支持加密等。当然加密意味着需要解密函数，因而使用dexguard会比使用proguard增加一些方法。

**新的方案：**

使用类Proguard的工具，把多个方法合并成一个方法，从而达到减少方法数量的目的。

**猜想方案：**

方案一：仿照proguard，在javac编译成class文件后，使用dx工具前，对class文件进行修改。

方案二：直接对编译完成后的dex文件进行修改。

**方案对比：**

|  |  |  |
| --- | --- | --- |
|  | 优势 | 劣势 |
| 方案一 | 1、Java经过多年发展，修改字节码的库已经非常成熟。  2、bytecode是基于堆栈的，在增加减少变量以及指令上非常方便。  3、如果直接基于proguard后的mapping文件来处理，可以免去非常繁琐的配置。 | 只能针对源码编译过程中使用，作用范围有限 |
| 方案二 | 应用范围广 | 1、修改Dex的库并不成熟。  2、dalvik opcodes基于寄存器，本地变量的数量增加会影响到参数的寄存器位置，以致于增加或减少变量以及指令的成本非常高。  3、直接修改Dex，有可能会修改到插件化所提供的接口，因而必须由开发者进行非常多的配置。 |

由于方法数的限制一般都是在开发后编译出现的，因此方案一的劣势基本可以忽略。综上所述，**选择方案一更优**。

**待处理方法选择：**

1、只要被Proguard混淆过的方法，都可能可以进行合并，而未混淆的代码，则由于可能有各种各样的潜规则（如JNI或其它地方反射调用该方法、作为插件化框架所提供的接口等），而需要通过配置Keep住。

2、没继承关系的方法，皆可进行处理。

3、有继承关系的方法，虽有部分能处理，但经测试量较少，且处理成本非常大，因而不建议处理。

**合并原理（基于bytecode）：**

1、实例方法-->静态方法：

|  |  |
| --- | --- |
| **class** Book {  **int** n;  **int** add(**int** b) {  **return** n + b;  }  } | **class** Book{  **int** n;  **static** **int** add(Book a, **int** b) {  **return** a.n+b;  }  } |
| **class** com/tencent/Book {    I n    add(I)I      ALOAD 0      GETFIELD com/tencent/Book.n : I      ILOAD 1      IADD      IRETURN  } | **class** com/tencent/Book {    I n  **static** add(**Lcom/tencent/Book;**I)I      ALOAD 0      GETFIELD com/tencent/Book.n : I      ILOAD 1      IADD      IRETURN  } |

2、多个方法-->一个方法（通用方案）:

|  |  |
| --- | --- |
| static long add(int a, long b) {  return a+b;  }  **static**add(IL)L      ILOAD 0      I2L      LLOAD 1      LADD      LRETURN  ---------------------------------------------  static long print(String str, long k) {  return str.length() - k;  }  **static**print(Ljava/lang/String;L)L      ALOAD 0      INVOKEVIRTUAL java/lang/String.length ()I      I2L      LLOAD 1      LSUB      LRETURN | static long merge(int which, int a, long b, String str) {       switch(which) {      case 0:            return a+b;       case 1:            return str.length() - k;       default:            throw new RuntimeException();       }  }  **static**merge(IIJLjava/lang/String;)J  L0      ILOAD 0      TABLESWITCH        0: L1        1: L2        default: L3  L1  **ILOAD 1**  **LLOAD 2**  **LSTORE 1**  **ISTORE 0**      ILOAD 0      I2L      LLOAD 1      LADD      LRETURN  L2  **ALOAD 4**  **LLOAD 2**  **LSTORE 1**  **ASTORE 0**      ALOAD 0      INVOKEVIRTUAL java/lang/String.length ()I      I2L      LLOAD 1      LSUB      LRETURN  L3      NEW java/lang/RuntimeException      DUP      INVOKESPECIAL java/lang/RuntimeException.<init> ()V      ATHROW |

3、多个方法-->一个方法（start with方案）:

|  |  |
| --- | --- |
| static long add(int a, long b) {  return a+b;  }  **static**add(IL)L      ILOAD 0      I2L      LLOAD 1      LADD      LRETURN  ---------------------------------------  static long print(int m, long k, String str) {  return str.length() - k + m;  }  **static**print(Ljava/lang/String;L)L      ALOAD 3      INVOKEVIRTUAL java/lang/String.length ()I      I2L      LLOAD 1      LSUB      ILOAD 0      I2L      LADD      LRETURN | static long merge(int a, long b, String str, int which) {       switch(which) {      case 0:            return a+b;       case 1:            return str.length() - b + a;       default:            throw new RuntimeException();       }  }  **static**merge(IJLjava/lang/String;)J  L0      ILOAD 4      TABLESWITCH        0: L1        1: L2        default: L3  L1      ILOAD 0      I2L      LLOAD 1      LADD      LRETURN  L2      ALOAD 3      INVOKEVIRTUAL java/lang/String.length ()I      I2L      LLOAD 1      LSUB      ILOAD 0      I2L      LADD      LRETURN  L3      NEW java/lang/RuntimeException      DUP      INVOKESPECIAL java/lang/RuntimeException.<init> ()V      ATHROW |

4、可内联函数插入到函数调用的位置：

|  |  |
| --- | --- |
| 内联函数：  static long add(int a, long b) {  return a+b;  }  **static** add(II)I      ILOAD 0      ILOAD 1      IADD      IRETURN  ---------------------------------------  调用函数：  static int count(int a, int b, int c) {      return add(add(a,b),add(b,c));  }  **static** count(III)I      ILOAD 0      ILOAD 1      INVOKESTATIC Main.add (II)I      ILOAD 1      ILOAD 2      INVOKESTATIC Main.add (II)I      INVOKESTATIC Main.add (II)I      IRETURN | static int count(int a, int b, int c) {      return (a+b) + (b+c)  }  **static** count(III)I      ILOAD 0      ILOAD 1  **ISTORE 3**  **ISTORE 4**  ILOAD **4**  ILOAD **3**      IADD      ILOAD 1      ILOAD 2  **ISTORE 3**  **ISTORE 4**  ILOAD **4**  ILOAD **3**      IADD  **ISTORE 3**  **ISTORE 4**  ILOAD **4**  ILOAD **3**      IADD      IRETURN |

**合并方案说明：**

1、通用方案会增加非常多的指令，以致于dex体积增加，而除了65K限制外，还有个8M的限制，因此通用方案可能会导致提前达到8M的限制，不建议使用。

2、内联函数方案在处理插件化框架中，修复成本非常高，且经测试，可内联的函数比例非常低，因而目前不使用。

3、上面方案中，都是针对没继承关系的来处理，并没列出有继承关系的处理方案，当然有兴趣的可找我拿或自行研究下。这里不列出是因为此方案能处理的方法比例非常低，且处理成本非常大，因而目前不使用。

**测试情况：**

以下是管家某个版本的测试结果：

|  |  |
| --- | --- |
| 总方法数 | 46485 |
| 总混淆方法数 | 17698 |
| 总处理方法数 | 10970 |
| 减少方法数 | **9553** |
| 原APK大小 | 9357811 |
| 现APK大小 | 9413331 |
| 原Dex大小 | 7792156 |
| 现Dex大小 | **7761204** |

注：合并后dex可能会减少，但APK却是增大了，原因在于压缩比率不一样。

此工具目前主要针对于无继承关系，且被proguard混淆过的来处理。

**特性说明：**

1、对受限的孤立方法进行类内合并，对其它孤立方法进行类外合并。（注：public、protected、private都会进行统一合并）。

2、合并方案只使用start with方案，以防止体积的增大。

3、针对有些项目使用了自己改写的proguard，因此这里采用了注入proguard的方式，不会影响到proguard的正常使用。

4、如果需要插件化编译时，则需另外增加一个简单配置，否则直接使用原proguard的配置即可。

5、提供相应的堆栈还原工具，与proguard的retrace工具类似。

**工具使用：**

1、快速使用：

a、打开网站：http://hanjiewu.cs0309.3g.qq.com/webapp\_scan/security\_lab/jar/

b、选择“代码混淆”，上传编译过程中，proguard所产生的jar和mapping.txt。

c、等待“下载混淆JAR”按钮出现，即可下载一个zip文件，其中内含的xxx.jar为压缩后的jar，mapping\_op.txt为混淆工具产生的mapping文件，用于堆栈还原。

2、集成到编译过程：

请联系RTX：noverguo或vincexie