Smali 学习笔记

目录

文档修改记录

修改记录	提交人	版本
文档创建	吴志栩	V1.0
修订，完善	邹军华	V2.0

1 Dalvik 与 Smali

1.1 Dalvik 虚拟机概述

Google 于 2007 年底正式发布了 Android SDK，Dalvik 虚拟机也第一次进入了我们的视野。它的作者是丹·伯恩斯坦（Dan Bornstein），名字来源于他的祖先曾经居住过的名叫 Dalvik 的小渔村。Dalvik 虚拟机作为 Android 平台的核心组件，拥有如下几个特点：

1) 体积小，占用内存空间小；
2) 专用的 DEX 可执行文件格式，体积更小，执行速度更快；
3) 常量池采用 32 位索引值，寻址类方法名，字段名，常量更快；
4) 基于寄存器架构，并拥有一套完整的指令系统；
5) 提供了对象生命周期管理、堆栈管理、线程管理、安全和异常管理以及垃圾回收等重要功能；
6) 所有的 Android 程序都运行在 Android 系统进程里，每个进程对应着一个 Dalvik 虚拟机实例。

1.2 Dalvik 虚拟机与 Java 虚拟机的区别

Dalvik 虚拟机与传统的 Java 虚拟机有着许多不同点，两者并不兼容，它们显著的不同点主要表现在以下几个方面：

1) Java 虚拟机运行的是 Java 字节码，Dalvik 虚拟机运行的是 Dalvik 字节码；
2) Dalvik 可执行文件体积更小；
3) Java 虚拟机基于栈架构，Dalvik 虚拟机基于寄存器架构。

1.3 Smali 概述

我们都知道，Dalvik 虚拟机（Dalvik VM）是 Google 专门为 Android 平台设计的一套虚拟机。区别于标准 Java 虚拟机 JVM 的 class 文件格式，Dalvik VM 拥有专属的 DEX 可执行文件格式和指令集代码。smali 和 baksmali 则是针对 DEX 执行文件格式的汇编器和反汇编器，反汇编后 DEX 文件会产生.smali 后缀的代码文件，smali 代码拥有特定的格式与语法，smali 语言是对 Dalvik 虚拟机字节码的一种解释。

Smali 语言起初是由一个名叫 JesusFreke 的 hacker 对 Dalvik 字节码的翻译，并非一种官方标准语言，因为 Dalvik 虚拟机名字来源于冰岛一个小渔村的名字，JesusFreke 便把 smali 和 baksmali 取自了冰岛语中的“汇编器”和“反编器”。目前 Smali 是在 Google Code 上的一个开源项目。

虽然主流的 DEX 可执行文件反汇编工具不少，如 Dedexer、IDA Pro 和 dex2jar+jd-gui，但 Smali 提供反汇编功能的同时，也提供了打包反汇编代码重新生成 dex 的功能，因此 Smali 被广泛地用于 APP 广告注入、汉化和破解，ROM 定制等方面。

2 反编译、编译、打包

2.1 使用 smali 和 baksmali

smali 和 baksmali 这两个工具汇编和反汇编 DEX 文件的使用非常简单，我们使用 baksmali.jar 反汇编 HelloWorld.dex 只需输入以下命令：

```
java -jar baksmali.jar -o HelloWorldOut HelloWorld.dex
```

命令执行成功后会在 HelloWorldOut 目录下生成相应 smali 文件，我们在修改完 smali 代码后，使用 smali.jar 重新汇编成 HelloWorld.dex，输入命令：

```
java -jar smali.jar -o HelloWorld.dex HelloWoldOut
```

我们只需把生成的 HelloWorld.dex 通过 adb 命令 push 到手机上，并使用 dalvikvm 命令便可以运行这个 DEX 文件，执行的命令如下：

```
adb push HelloWorld.dex /data/local/
adb shell dalvikvm -cp /data/local/HelloWorld.dex HelloWorld
```

使用 smali 和 baksmali 工具只能用来汇编和反汇编 Android 的可执行文件，因此使用这两个工具是无法完成 Android 安装包（APK）的反编译和打包的功能，好在一个名为 android-apktool 的 Google 开源工具可以完成这个事情。

2.2 使用 apktool

apktool 工具是在 smali 工具的基础上进行封装和改进的，除了对 DEX 文件的汇编和反汇编功能外，还可以对 APK 中已编译成二进制的资源文件进行反编译和重新编译。同时也支持给 smali 代码添加调试信息以支持断点调试。因此我们直接使用 apktool 工具来反编译 Android apps，而不是 smali 和 baksmali 工具。

2.2.1 APK 组成

在介绍 apktool 使用之前，我们先来看看一个 apk 包的组成。apk 文件其实是 zip 压缩包格式，我们使用工具进行解压后可以其目录组成，以 HelloWorld.apk 为例：

```
|-- HelloWorld/
    |-- AndroidManifest.xml     被编译成二进制的配置文件
    |-- class.dex               Android 可执行文件
    |-- resources.arsc          被编译成二进制的主资源文件
    |-- assets/                 不需编译的原始资源文件目录
    |-- res/                    资源文件目录
    |-- lib/                    库文件目录
    |-- META-INF/               APK 的签名信息
```

2.2.2 使用方法

使用 apktool 反编译 HelloWorld.apk 文件到 out 目录的方法：

```
java -jar apktool.jar d HelloWorld.apk out/
```

apktool d 命令执行成功会在 out 目录下产生如下所示的一级目录结构：

```
|-- out/
    |-- AndroidManifest.xml     配置文件
    |-- apktool.yml             反编译生成的文件，供 apktool 使用
    |-- assets/                 不需反编译的资源文件目录
    |-- lib/                    不需反编译的库文件目录
    |-- res/                    反编译后的资源文件目录
    |-- smali/                  反编译生成的 smali 源码文件目录
```

在浏览各个子目录的结构后，我们可以发现其结构原始 APP 工程目录结构基本一致，smali 目录结构对应着原始的 java 源码 src 目录，而 META-INF 目录已经不见了，因为反编译会丢失签名信息。反编译后会多生成 apktool.yml 文件，这个文件记录着 apktool 版本和 Apk 文件名和是否 framework 文件等基本信息，在 apktool 重新编译时会使用到。
在修改完相应的 smali 代码和资源文件后，我们可以使用以下命令进行编译打包：

```
java -jar apktool.jar b out/ HelloWorld.apk
```

有时候我们在反编译系统 app 的时候会出错，提示无法找到资源，必须先加载合适的 framework 资源文件。这是因为 apktool 编译和反编译过程会依赖到 framework 中的代码和资源，而这个 app 中使用了 framework-res.apk 之外扩展的 framework 资源文件。因此，我们通常反编译前会把 system/framework/目录下所有资源文件 apk 都先加载在 apktool 工具中。使用的命令如下：

```
java -jar apktool.jar if framework-res.apk
```

2.2.3 反编译、编译过程

我们执行 apktool d 命令可以看整个反编译过程的输出信息：

```
I: Baksmaling...
I: Loading resource table...
I: Loaded.
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file: /home/wzx/apktool/framework/1.apk
I: Loaded.
I: Decoding file-resources...
I: Decoding values*/* XMLs...
I: Done.
I: Copying assets and libs...
```

整个反编译过程可以分为四个阶段：

1) DEX 可执行文件 baksmali 反汇编过程；
2) 二进制配置文件 AndroidManifest.xml 反编译；
3) 根据 framework 资源列表反编译二进制资源文件；
4) 直接复制不需要反编译的资源和库文件。

为最大程度地还原被编译成二进制的资源文件，apktool 需要一个可读性较高的资源的名称，而不是二进制索引，这些名称则是在 framework 的资源文件中被定义。apktool 默认内嵌了 Android 标准 framework 资源，因此大部分 APP 反编译不需要指定资源文件，而通常厂商都会扩展自己的资源文件，并在系统 APP 中引用，这就是我们前面提到的反编译前需要加载/system/framework 下所有资源文件的原因。

apktool b 命令编译打包过程输出的信息：

```
I: Checking whether sources has changed...
I: Smaling...
I: Checking whether resources has changed...
I: Building resources...
I: Copying libs...
I: Building apk file...
```

编译打包过程也可以分为四个阶段：

1) 汇编 smali 源码目录生成 DEX 可执行文件；
2) 使用 aapt 把资源文件编译成二进制格式文件；
3) 复制不需编译的库文件到目标目录；
4) 使用 aapt 把目标文件重新打包成 apk 文件。

2.3 deodex

odex（Optimized DEX）是一种经过优化的 dex 文件格式，大部分厂商发布的手机上都会做 odex 处理。odex 处理过的 image 包，其系统 jar/apk 包的可执行文件是存在同目录下的同名.odex 文件，jar/apk 包中是不存在 dex 文件的。

对于 odex 文件，我们只需知道两点：

1) odex 的文件结构是 dex 文件的一个超集，会在 dex 文件基础上附加一些数据；
2) odex 文件对 image 包中的/system/framework（可具体到目录中 core.jar、ext.jar、framework.jar、services.jar 和 android.policy.jar 这 5 个 jar 包）具有强依赖性，odex 文件在不同的 framework 上是无法被 dalvik 虚拟机加载的。

使用 apktool 是无法直接对 odex 文件进行反编译，我们必须先使用 baksmali 进行 deodex 处理。

我们以系统 APP 计算器 Calculator.apk 为例，进行 deodex 的完整过程如下：

1) odex 处理过的 apk 包中不包含 dex 文件，其对应的 odex 文件放在与 apk 文件同一目录下，我们先把 apk 文件中其他的资源等文件先反编译出来。

```
java -jar apktool.jar d Calculator.apk ./Calculator
```

2) 通过 aapt 命令获取 apk 使用的 SDK 的版本号 targetSdkVersion 为 15。

```
aapt d badging Calculator.apk | grep targetSdkVersion
```

3) 使用 baksmali 对 Calculator.odex 去 odex 并反编译成 smali 代码文件，-a 指明 sdk 版本，-x 指定需 deodex 的 odex 文件，-d 指明依赖的 framework 目录，-o 指定去输出目录。

```
java -jar baksmali.jar -a 15 -x Calculator.odex -d framework \
    -o Calculator/smali
```

4) 在修改完相应的 smali 文件和资源文件后，我们可以重新编译打包成 apk 文件。

```
java -jar apktool.jar d Calculator Calculator.apk
```

3 Smali 语法规范与格式

Smali 是对 Dalvik 虚拟机字节码的一种解释，虽然不是官方标准语言，但所有语句都遵循一套语法规范。要了解 smali 语法规范，可以先从了解 Dalvik 虚拟机字节码的指令格式开始。

3.1 Dalvik 虚拟机字节码指令格式

在 Android 4.0 源码 Dalvik/docs 目录下提供了一份文档 instruction-formats.html，里面详细列举了 Dalvik 虚拟机字节码指令的所有格式，翻译为中文后如下：

Opcode 操作码(hex)	Opcode name 操作码名称	Explanation 说明	Example 示例
00	nop	无操作	0000 - nop
01	move vx, vy	移动 vy 的内容到 vx。两个寄存器都必须在最初的 256 寄存器范围以内。	0110 - move v0, v1 移动 v1 寄存器中的内容到 v0。
02	move/from16 vx, vy	移动 vy 的内容到 vx。vy 可能在 64K 寄存器范围以内，而 vx 则是在最初的 256 寄存器范围以内。	0200 1900 - move/from16 v0, v25 移动 v25 寄存器中的内容到 v0。
03	move/16	未知[注4]	
04	move-wide	未知[注4]	
05	move-wide/from16 vx, v	移动一个 long/double 值，从 vy 到 vx。vy 可能	0516 0000 - move-wide/from16 v22, v0

	y	在 64K 寄存器范围以内，而 vx 则是在最初的 256 寄存器范围以内。	移动 v0,v1 寄存器中的内容到 v22,v23。
06	move-wide/16	未知[注4]	
07	move-object vx, vy	移动对象引用，从 vy 到 vx。	0781 - move-object v1, v8 移动 v8 寄存器中的对象引用到 v1。
08	move-object/from16 vx, vy	移动对象引用，从 vy 到 vx。vy 可以处理 64K 寄存器地址，vx 可以处理 256 寄存器地址。	0801 1500 - move-object/from16 v1, v21 移动 v21 寄存器中的对象引用到 v1。
09	move-object/16	未知[注4]	
0A	move-result vx	移动上一次方法调用的返回值到 vx。	0A00 - move-result v0 移动上一次方法调用的返回值到 v0。
0B	move-result-wide vx	移动上一次方法调用的 long/double 型返回值到 vx,vx+1。	0B02 - move-result-wide v2 移动上一次方法调用的 long/double 型返回值到 v2,v3。
0C	move-result-object vx	移动上一次方法调用的对象引用返回值到 vx。	0C00 - move-result-object v0 移动上一次方法调用的对象引用返回值到 v0。
0D	move-exception vx	当方法调用抛出异常时移动异常对象引用到 vx。	0D19 - move-exception v25 当方法调用抛出异常时移动异常对象引用到 v25。
0E	return-void	返回空值。	0E00 - return-void 返回值为 void，即无返回值，并非返回 null。
0F	return vx	返回在 vx 寄存器的值。	0F00 - return v0 返回 v0 寄存器中的值。
10	return-wide vx	返回在 vx,vx+1 寄存器的 double/long 值。	1000 - return-wide v0 返回 v0,v1 寄存器中的 double/long 值。
11	return-object vx	返回在 vx 寄存器的对象引用。	1100 - return-object v0 返回 v0 寄存器中的对象引用。
12	const/4 vx, lit4	存入 4 位常量到 vx。	1221 - const/4 v1, #int 2 存入 int 型常量 2 到 v1。目的寄存器在第二个字节的低 4 位，常量 2 在更高的 4 位。
13	const/16 vx, lit16	存入 16 位常量到 vx。	1300 0A00 - const/16 v0, #int 10 存入 int 型常量 10 到 v0。

14	const vx, lit32	存入 int 型常量到 vx。	1400 4E61 BC00 - const v0, #12345678 // #00BC614E 存入常量 12345678 到 v0。
15	const/high16 v0, lit16	存入 16 位常量到最高位寄存器，用于初始化 float 值。	1500 2041 - const/high16 v0, #float 10.0 // #41200000 存入 float 常量 10.0 到 v0。该指令最高支持 16 位浮点数。
16	const-wide/16 vx, lit16	存入 int 常量到 vx,vx+1 寄存器，扩展 int 型常量为 long 常量。	1600 0A00 - const-wide/16 v0, #long 10 存入 long 常量 10 到 v0,v1 寄存器。
17	const-wide/32 vx, lit32	存入 32 位常量到 vx,vx+1 寄存器，扩展 int 型常量到 long 常量。	1702 4e61 bc00 - const-wide/32 v2, #long 12345678 // #00bc614e 存入 long 常量 12345678 到 v2,v3 寄存器。
18	const-wide vx, lit64	存入 64 位常量到 vx,vx+1 寄存器。	1802 874b 6b5d 54dc 2b00- const-wide v2, #long 1234567890123 4567 // #002bdc545d6b4b87 存入 long 常量 12345678901234567 到 v2,v3 寄存器。
19	const-wide/high16 vx, lit16	存入 16 位常量到最高 16 位的 vx,vx+1 寄存器，用于初始化 double 值。	1900 2440 - const-wide/high16 v0, #double 10.0 // #402400000 存入 double 常量 10.0 到 v0,v1。
1A	const-string vx, *字符串 ID*	存入字符串常量引用到 vx，通过*字符串 ID* 或*字符串*。	1A08 0000 - const-string v8, "" // string@0000 存入 string@0000（字符串表#0 条目）的引用到 v8。
1B	const-string-jumbo	未知[注4]	
1C	const-class vx, *类型 ID*	存入类对象常量到 vx，通过*类型 ID* 或*类型*（如 Object.class）。	1C00 0100 - const-class v0, Test3 // type@0001 存入 Test3.class（类型 ID 表#1 条目）的引用到 v0。
1D	monitor-enter vx	获得 vx 寄存器中的对象引用的监视器。	1D03 - monitor-enter v3 获得 v3 寄存器中的对象引用的监视器。
1E	monitor-exit	释放 vx 寄存器中的对象引用的监视器。	1E03 - monitor-exit v3 释放 v3 寄存器中的对象引用的监视器。
1F	check-cast vx, *类型 ID*	检查 vx 寄存器中的对象引用是否可以转换成*类型 ID* 对应类型的实例。如不可转	1F04 0100 - check-cast v4, Test3 // type@0001 检查 v4 寄存器中的对象引用是否可以

		换，抛出 ClassCastException 异常，否则继续执行。	转换成 Test3（类型 ID 表#1 条目）的实例。
20	instance-of vx, vy, *类型 ID*	检查 vy 寄存器中的对象引用是否是*类型 ID* 对应类型的实例，如果是，vx 存入非 0 值，否则 vx 存入 0。	2040 0100 - instance-of v0, v4, Test3 // type@0001 检查 v4 寄存器中的对象引用是否是 Test3（类型 ID 表#1 条目）的实例。如果是，v0 存入非 0 值，否则 v0 存入 0。
21	array-length vx, vy	计算 vy 寄存器中数组引用的元素长度并将长度存入 vx。	2111 - array-length v0, v1 计算 v1 寄存器中数组引用的元素长度并将长度存入 v0。
22	new-instance vx, *类型 ID*	根据*类型 ID* 或*类型*新建一个对象实例，并将新建的对象的引用存入 vx。	2200 1500 - new-instance v0, java.io.FileInputStream // type@0015 实例化 java.io.FileInputStream（类型 ID 表#15H 条目）类型，并将其对象引用存入 v0。
23	new-array vx, vy, *类型 ID*	根据*类型 ID* 或*类型*新建一个数组，vy 存入数组的长度，vx 存入数组的引用。	2312 2500 - new-array v2, v1, char[] // type@0025 新建一个 char（类型 ID 表#25H 条目）数组，v1 存入数组的长度，v2 存入数组的引用。
24	filled-new-array {*参数*}, *类型 ID*	根据*类型 ID* 或*类型*新建一个数组并通过*参数*填充[注5]。新的数组引用可以得到一个 move-result-object 指令，前提是执行过 filled-new-array 指令。	2420 530D 0000 - filled-new-array {v0,v0},[I // type@0D53 新建一个 int（类型 ID 表#D53H 条目）数组，长度将为 2 并且 2 个元素将填充到 v0 寄存器。
25	filled-new-array-range {vx..vy}, *类型 ID*	根据*类型 ID* 或*类型*新建一个数组并以寄存器范围为参数填充。新的数组引用可以得到一个 move-result-object 指令，前提是执行过 filled-new-array 指令。	2503 0600 1300 - filled-new-array/range {v19..v21}, [B // type@0006 新建一个 byte（类型 ID 表#6 条目）数组，长度将为 3 并且 3 个元素将填充到 v19,v20,v21 寄存器[注4]。
26	fill-array-data vx, *偏移量*	用 vx 的静态数据填充数组引用。静态数据的位址是当前指令位置加*偏移量*的和。	2606 2500 0000 - fill-array-data v6, 00e6 // +0025 用当前指令位置+25H 的静态数据填充 v6 寄存器的数组引用。偏移量是 32 位的数字，静态数据的存储格式如下： 0003 // 表类型：静态数组数据 0400 // 每个元素的字节数（这个例

			子是 4 字节的 int 型） 0300 0000 // 元素个数 0100 0000 // 元素 #0：int 1 0200 0000 // 元素 #1：int 2 0300 0000 // 元素 #2：int 3
27	throw vx	抛出异常对象，异常对象的引用在 vx 寄存器。	2700 - throw v0 抛出异常对象，异常对象的引用在 v0 寄存器。
28	goto *目标*	通过短偏移量[注 2] 无条件跳转到*目标*。	28F0 - goto 0005 // -0010 跳转到当前位置-16（hex 10）的位置，0005 是目标指令标签。
29	goto/16 *目标*	通过 16 位偏移量[注2]无条件跳转到*目标*。	2900 0FFE - goto/16 002f // -01f1 跳转到当前位置-1F1H 的位置，002f 是目标指令标签。
2A	goto/32 *目标*	通过 32 位偏移量[注2]无条件跳转到*目标*。	
2B	packed-switch vx, *索引表偏移量*	实现一个 switch 语句，case 常量是连续的。这个指令使用*索引表*，vx 是在表中找到具体 case 的指令偏移量的索引，如果无法在表中找到 vx 对应的索引将继续执行下一个指令（即 default case）。	2B02 0C00 0000 - packed-switch v2, 000c // +000c 根据 v2 寄存器中的值执行 packed switch，索引表的位置是当前指令位置+0CH，表如下所示： 0001 // 表类型：packed switch 表 0300 // 元素个数 0000 0000 // 基础元素 0500 0000 0: 00000005 // case 0: +00000005 0700 0000 1: 00000007 // case 1: +00000007 0900 0000 2: 00000009 // case 2: +00000009
2C	sparse-switch vx, *查询表偏移量*	实现一个 switch 语句，case 常量是非连续的。这个指令使用*查询表*，用于表示 case 常量和每个 case 常量的偏移量。如果 vx 无法在表中匹配将继续执行下一个指令（即 default case）。	2C02 0c00 0000 - sparse-switch v2, 000c // +000c 根据 v2 寄存器中的值执行 sparse switch ，查询表的位置是当前指令位置+0CH，表如下所示： 0002 // 表类型：sparse switch 表 0300 // 元素个数 9cff ffff // 第一个 case 常量: -100 fa00 0000 // 第二个 case 常量: 250 e803 0000 // 第三个 case 常量: 1

			000 0500 0000 // 第一个 case 常量的偏移量: +5 0700 0000 // 第二个 case 常量的偏移量: +7 0900 0000 // 第三个 case 常量的偏移量: +9
2D	cmpl-float vx, vy, vz	比较 vy 和 vz 的 float 值并在 vx 存入 int 型返回值[注3]。	2D00 0607 - cmpl-float v0, v6, v7 比较 v6 和 v7 的 float 值并在 v0 存入 int 型返回值。非数值默认为小于。如果参数为非数值将返回-1。
2E	cmpg-float vx, vy, vz	比较 vy 和 vz 的 float 值并在 vx 存入 int 型返回值[注3]。	2E00 0607 - cmpg-float v0, v6, v7 比较 v6 和 v7 的 float 值并在 v0 存入 int 型返回值。非数值默认为大于。如果参数为非数值将返回 1。
2F	cmpl-double vx, vy, vz	比较 vy 和 vz[注2] 的 double 值并在 vx 存入 int 型返回值[注3]。	2F19 0608 - cmpl-double v25, v6, v8 比较 v6,v7 和 v8,v9 的 double 值并在 v25 存入 int 型返回值。非数值默认为小于。如果参数为非数值将返回-1。
30	cmpg-double vx, vy, vz	比较 vy 和 vz[注2] 的 double 值并在 vx 存入 int 型返回值[注3]。	3000 080A - cmpg-double v0, v8, v10 比较 v8,v9 和 v10,v11 的 double 值并在 v0 存入 int 型返回值。非数值默认为大于。如果参数为非数值将返回 1。
31	cmp-long vx, vy, vz	比较 vy 和 vz 的 long 值并在 vx 存入 int 型返回值[注3]。	3100 0204 - cmp-long v0, v2, v4 比较 v2 和 v4 的 long 值并在 v0 存入 int 型返回值。
32	if-eq vx,vy, *目标*	如果 vx == vy[注2]，跳转到 *目标*。vx 和 vy 是 int 型值。	32b3 6600 - if-eq v3, v11, 0080 // +0066 如果 v3 == v11，跳转到当前位置+66H。0080 是目标指令标签。
33	if-ne vx,vy, *目标*	如果 vx != vy[注2]，跳转到 *目标*。vx 和 vy 是 int 型值。	33A3 1000 - if-ne v3, v10, 002c // +0010 如果 v3 != v10，跳转到当前位置+10H。002c 是目标指令标签。
34	if-lt vx,v	如果 vx < vy[注2]，跳转到	3432 CBFF - if-lt v2, v3, 0023

	y, *目标*	*目标*。vx 和 vy 是 int 型值。	// -0035 如果 v2 < v3，跳转到当前位置-35H。0023 是目标指令标签。
35	if-ge vx, vy, *目标*	如果 vx >= vy [注2]，跳转到*目标*。vx 和 vy 是 int 型值。	3510 1B00 - if-ge v0, v1, 002b // +001b 如果 v0 >= v1，跳转到当前位置+1BH。002b 是目标指令标签。
36	if-gt vx,vy, *目标*	如果 vx > vy [注2]，跳转到*目标*。vx 和 vy 是 int 型值。	3610 1B00 - if-ge v0, v1, 002b // +001b 如果 v0 > v1，跳转到当前位置+1BH。002b 是目标指令标签。
37	if-le vx,vy, *目标*	如果 vx <= vy [注2]，跳转到*目标*。vx 和 vy 是 int 型值。	3756 0B00 - if-le v6, v5, 0144 // +000b 如果 v6 <= v5，跳转到当前位置+0BH。0144 是目标指令标签。
38	if-eqz vx, *目标*	如果 vx == 0 [注2]，跳转到*目标*。vx 是 int 型值。	3802 1900 - if-eqz v2, 0038 // +0019 如果 v2 == 0，跳转到当前位置+19H。0038 是目标指令标签。
39	if-nez vx, *目标*	如果 vx != 0 [注2]，跳转到*目标*。	3902 1200 - if-nez v2, 0014 // +0012 如果 v2 != 0，跳转到当前位置+18(hex 12)。0014 是目标指令标签。
3A	if-ltz vx, *目标*	如果 vx < 0 [注2]，跳转到*目标*。	3A00 1600 - if-ltz v0, 002d // +0016 如果 v0 < 0，跳转到当前位置+16H。002d 是目标指令标签。
3B	if-gez vx, *目标*	如果 vx >= 0 [注2]，跳转到*目标*。	3B00 1600 - if-gez v0, 002d // +0016 如果 v0 >= 0，跳转到当前位置+16H。002d 是目标指令标签。
3C	if-gtz vx, *目标*	如果 vx > 0 [注2]，跳转到*目标*。	3C00 1D00 - if-gtz v0, 004a // +001d 如果 v0 > 0，跳转到当前位置+1DH。004a 是目标指令标签。
3D	if-lez vx, *目标*	如果 vx <= 0 [注2]，跳转到*目标*。	3D00 1D00 - if-lez v0, 004a // +001d 如果 v0 <= 0，跳转到当前位置+1DH。004a 是目标指令标签。
3E	unused_3E	未使用	
3F	unused_3F	未使用	

40	unused_40	未使用	
41	unused_41	未使用	
42	unused_42	未使用	
43	unused_43	未使用	
44	aget vx, vy, vz	从 int 数组获取一个 int 型值到 vx，对象数组的引用位于 vy，需获取的元素的索引位于 vz。	4407 0306 - aget v7, v3, v6 从数组获取一个 int 型值到 v7，对象数组的引用位于 v3，需获取的元素的索引位于 v6。
45	aget-wide vx, vy, vz	从 long/double 数组获取一个 long/double 值到 vx,vx+1，数组的引用位于 vy，需获取的元素的索引位于 vz。	4505 0104 - aget-wide v5, v1, v4 从 long/double 数组获取一个 long/double 值到 v5,vx6，数组的引用位于 v1，需获取的元素的索引位于 v4。
46	aget-object vx, vy, vz	从对象引用数组获取一个对象引用到 vx，对象数组的引用位于 vy，需获取的元素的索引位于 vz。	4602 0200 - aget-object v2, v2, v0 从对象引用数组获取一个对象引用到 v2，对象数组的引用位于 v2，需获取的元素的索引位于 v0。
47	aget-boolean vx, vy, vz	从 boolean 数组获取一个 boolean 值到 vx，数组的引用位于 vy，需获取的元素的索引位于 vz。	4700 0001 - aget-boolean v0, v0, v1 从 boolean 数组获取一个 boolean 值到 v0，数组的引用位于 v0，需获取的元素的索引位于 v1。
48	aget-byte vx, vy, vz	从 byte 数组获取一个 byte 值到 vx，数组的引用位于 vy，需获取的元素的索引位于 vz。	4800 0001 - aget-byte v0, v0, v1 从 byte 数组获取一个 byte 值到 v0，数组的引用位于 v0，需获取的元素的索引位于 v1。
49	aget-char vx, vy, vz	从 char 数组获取一个 char 值到 vx，数组的引用位于 vy，需获取的元素的索引位于 vz。	4905 0003 - aget-char v5, v0, v3 从 char 数组获取一个 char 值到 v5，数组的引用位于 v0，需获取的元素的索引位于 v3。
4A	aget-short vx, vy, vz	从 short 数组获取一个 short 值到 vx，数组的引用位于 vy，需获取的元素的索引位于 vz。	4A00 0001 - aget-short v0, v0, v1 从 short 数组获取一个 short 值到 v0，数组的引用位于 v0，需获取的元素的索引位于 v1。
4B	aput vx, vy, vz	将 vx 的 int 值作为元素存入 int 数组，数组的引用位于 vy，元素的索引位于 vz。	4B00 0305 - aput v0, v3, v5 将 v0 的 int 值作为元素存入 int 数组，数组的引用位于 v3，元素的索引位于 v5。

4C	aput-wide vx, vy, vz	将 vx,vx+1 的 double/long 值作为元素存入 double/long 数组，数组的引用位于 vy，元素的索引位于 vz。	4C05 0104 - aput-wide v5, v1, v4 将 v5,v6 的 double/long 值作为元素存入 double/long 数组，数组的引用位于 v1，元素的索引位于 v4。
4D	aput-object vx, vy, vz	将 vx 的对象引用作为元素存入对象引用数组，数组的引用位于 vy，元素的索引位于 vz。	4D02 0100 - aput-object v2, v1, v0 将 v2 的对象引用作为元素存入对象引用数组，数组的引用位于 v1，元素的索引位于 v0。
4E	aput-boolean vx, vy, vz	将 vx 的 boolean 值作为元素存入 boolean 数组，数组的引用位于 vy，元素的索引位于 vz。	4E01 0002 - aput-boolean v1, v0, v2 将 v1 的 boolean 值作为元素存入 boolean 数组，数组的引用位于 v0，元素的索引位于 v2。
4F	aput-byte vx, vy, vz	将 vx 的 byte 值作为元素存入 byte 数组，数组的引用位于 vy，元素的索引位于 vz。	4F02 0001 - aput-byte v2, v0, v1 将 v2 的 byte 值作为元素存入 byte 数组，数组的引用位于 v0，元素的索引位于 v1。
50	aput-char vx, vy, vz	将 vx 的 char 值作为元素存入 char 数组，数组的引用位于 vy，元素的索引位于 vz。	5003 0001 - aput-char v3, v0, v1 将 v3 的 char 值作为元素存入 char 数组，数组的引用位于 v0，元素的索引位于 v1。
51	aput-short vx, vy, vz	将 vx 的 short 值作为元素存入 short 数组，数组的引用位于 vy，元素的索引位于 vz。	5102 0001 - aput-short v2, v0, v1 将 v2 的 short 值作为元素存入 short 数组，数组的引用位于 v0，元素的索引位于 v1。
52	iget vx, vy, *字段 ID*	根据*字段 ID* 读取实例的 int 型字段到 vx，vy 寄存器中是该实例的引用。	5210 0300 - iget v0, v1, Test2.i6:I // field@0003 读取 int 型字段 i6（字段表#3 条目）到 v0，v1 寄存器中是 Test2 实例的引用。
53	iget-wide vx, vy, *字段 ID*	根据*字段 ID* 读取实例的 double/long 型字段到 vx, vx+1 [注1], vy 寄存器中是该实例的引用。	5320 0400 - iget-wide v0, v2, Test2.l0:J // field@0004 读取 long 型字段 l0（字段表#4 条目）到 v0,v1，v2 寄存器中是 Test2 实例的引用。
54	iget-object vx, vy, *字*	根据*字段 ID* 读取一个实例的对象引用字段到 vx，vy	iget-object v1, v2, LineReader.fis:Ljava/io/FileInputStrea

	段 ID	寄存器中是该实例的引用。	m; // field@0002
			读取 FileInputStream 对象引用字段 fis（字段表#2 条目）到 v1，v2 寄存器中是 LineReader 实例的引用。

55	iget-boolean vx, vy, *字段 ID*	根据*字段 ID* 读取实例的 boolean 型字段到 vx，vy 寄存器中是该实例的引用。	55FC 0000 - iget-boolean v12, v15, Test2.b0:Z // field@0000 读取 boolean 型字段 b0（字段表#0 条目）到 v12，v15 寄存器中是 Test2 实例的引用。
56	iget-byte vx, vy, *字段 ID*	根据*字段 ID* 读取实例的 byte 型字段到 vx，vy 寄存器中是该实例的引用。	5632 0100 - iget-byte v2, v3, Test3.bi1:B // field@0001 读取 byte 型字段 bi1（字段表#1 条目）到 v2，v3 寄存器中是 Test2 实例的引用。
57	iget-char vx, vy, *字段 ID*	根据*字段 ID* 读取实例的 char 型字段到 vx，vy 寄存器中是该实例的引用。	5720 0300 - iget-char v0, v2, Test3.ci1:C // field@0003 读取 char 型字段 bi1（字段表#3 条目）到 v0，v2 寄存器中是 Test2 实例的引用。
58	iget-short vx, vy, *字段 ID*	根据*字段 ID* 读取实例的 short 型字段到 vx，vy 寄存器中是该实例的引用。	5830 0800 - iget-short v0, v3, Test3.si1:S // field@0008 读取 short 型字段 si1（字段表#8 条目）到 v0，v3 寄存器中是 Test2 实例的引用。
59	iput vx, vy, *字段 ID*	根据*字段 ID* 将 vx 寄存器的值存入实例的 int 型字段，vy 寄存器中是该实例的引用。	5920 0200 - iput v0, v2, Test2.i6:I // field@0002 将 v0 寄存器的值存入实例的 int 型字段 i6（字段表#2 条目），v2 寄存器中是 Test2 实例的引用。
5A	iput-wide vx, vy, *字段 ID*	根据*字段 ID* 将 vx,vx+1 寄存器的值存入实例的 double/long 型字段，vy 寄存器中是该实例的引用。	5A20 0000 - iput-wide v0, v2, Test2.d0:D // field@0000 将 v0,v1 寄存器的值存入实例的 double 型字段 d0（字段表#0 条目），v2 寄存器中是 Test2 实例的引用。

5B	iput-object vx, vy, *字段 ID*	根据*字段 ID* 将 vx 寄存器的值存入实例的对象引用字段，vy 寄存器中是该实例的引用。	5B20 0000 - iput-object v0, v2, LineReader.bis:Ljava/io/BufferedInputStream; // field@0000
			将 v0 寄存器的值存入实例的对象引用字段 bis（字段表#0 条目），v2 寄存器中是 BufferedInputStream 实例的引用。

5C	iput-boolean vx, vy, *字段 ID*	根据*字段 ID* 将 vx 寄存器的值存入实例的 boolean 型字段，vy 寄存器中是该实例的引用。	5C30 0000 - iput-boolean v0, v3, Test2.b0:Z // field@0000 将 v0 寄存器的值存入实例的 boolean 型字段 b0（字段表#0 条目），v3 寄存器中是 Test2 实例的引用。
5D	iput-byte vx, vy, *字段 ID*	根据*字段 ID* 将 vx 寄存器的值存入实例的 byte 型字段，vy 寄存器中是该实例的引用。	5D20 0100 - iput-byte v0, v2, Test3.bi1:B // field@0001 将 v0 寄存器的值存入实例的 byte 型字段 bi1（字段表#1 条目），v2 寄存器中是 Test2 实例的引用。
5E	iput-char vx, vy, *字段 ID*	根据*字段 ID* 将 vx 寄存器的值存入实例的 char 型字段，vy 寄存器中是该实例的引用。	5E20 0300 - iput-char v0, v2, Test3.ci1:C // field@0003 将 v0 寄存器的值存入实例的 char 型字段 ci1（字段表#3 条目），v2 寄存器中是 Test2 实例的引用。
5F	iput-short vx, vy, *字段 ID*	根据*字段 ID* 将 vx 寄存器的值存入实例的 short 型字段，vy 寄存器中是该实例的引用。	5F21 0800 - iput-short v1, v2, Test3.si1:S // field@0008 将 v0 寄存器的值存入实例的 short 型字段 si1（字段表#8 条目），v2 寄存器中是 Test2 实例的引用。
60	sget vx, *字段 ID*	根据*字段 ID* 读取静态 int 型字段到 vx。	6000 0700 - sget v0, Test3.is1:I // field@0007 读取 Test3 的静态 int 型字段 is1（字段表#7 条目）到 v0。
61	sget-wide vx, *字段 ID*	根据*字段 ID* 读取静态 double/long 型字段到 vx,vx+1。	6100 0500 - sget-wide v0, Test2.l1:J // field@0005 读取 Test2 的静态 long 型字段 l1（字段表#5 条目）到 v0,v1。

62	sget-object vx, *字段 ID*	根据*字段 ID* 读取静态对象引用字段到 vx。	6201 0C00 - sget-object v1, Test3.os1:Ljava/lang/Object; // field@000c
			读取 Object 的静态对象引用字段 os1（字段表#CH 条目）到 v1。

63	sget-boolean vx, *字段 ID*	根据*字段 ID* 读取静态 boolean 型字段到 vx。	6300 0C00 - sget-boolean v0, Test2.sb:Z // field@000c 读取 Test2 的静态 boolean 型字段 sb（字段表#CH 条目）到 v0。
64	sget-byte vx, *字段 ID*	根据*字段 ID* 读取静态 byte 型字段到 vx。	6400 0200 - sget-byte v0, Test3.bs1:B // field@0002 读取 Test3 的静态 byte 型字段 bs1（字段表#2 条目）到 v0。
65	sget-char v	根据*字段 ID* 读取静态 cha	6500 0700 - sget-char v0, Test

	x, *字段 ID*	r 型字段到 vx。	3.cs1:C // field@0007 读取 Test3 的静态 char 型字段 cs1（字段表#7 条目）到 v0。
66	sget-short vx, *字段 ID*	根据*字段 ID* 读取静态 short 型字段到 vx。	6600 0B00 - sget-short v0, Test3.ss1:S // field@000b 读取 Test3 的静态 short 型字段 ss1（字段表#CH 条目）到 v0。
67	sput vx, *字段 ID*	根据*字段 ID* 将 vx 寄存器中的值赋值到 int 型静态字段。	6700 0100 - sput v0, Test2.i5:I // field@0001 将 v0 寄存器中的值赋值到 Test2 的 int 型静态字段 i5（字段表#1 条目）。
68	sput-wide vx, *字段 ID*	根据*字段 ID* 将 vx,vx+1 寄存器中的值赋值到 double/long 型静态字段。	6800 0500 - sput-wide v0, Test2.l1:J // field@0005 将 v0,v1 寄存器中的值赋值到 Test2 的 long 型静态字段 l1（字段表#5 条目）。

69	sput-object vx, *字段 ID*	根据*字段 ID* 将 vx 寄存器中的对象引用赋值到对象引用静态字段。	6900 0c00 - sput-object v0, Test3.os1:Ljava/lang/Object; // field@000c
			将 v0 寄存器中的对象引用赋值到 Test3 的对象引用静态字段 os1（字段表#CH 条目）。

6A	sput-boolean vx, *字段 ID*	根据*字段 ID* 将 vx 寄存器中的值赋值到 boolean 型静态字段。	6A00 0300 - sput-boolean v0, Test3.bls1:Z // field@0003 将 v0 寄存器中的值赋值到 Test3 的 boolean 型静态字段 bls1（字段表#3 条目）。
6B	sput-byte vx, *字段 ID*	根据*字段 ID* 将 vx 寄存器中的值赋值到 byte 型静态字段。	6B00 0200 - sput-byte v0, Test3.bs1:B // field@0002 将 v0 寄存器中的值赋值到 Test3 的 byte 型静态字段 bs1（字段表#2 条目）。
6C	sput-char vx, *字段 ID*	根据*字段 ID* 将 vx 寄存器中的值赋值到 char 型静态字段。	6C01 0700 - sput-char v1, Test3.cs1:C // field@0007 将 v1 寄存器中的值赋值到 Test3 的 char 型静态字段 cs1（字段表#7 条目）。
6D	sput-short vx, *字段 ID*	根据*字段 ID* 将 vx 寄存器中的值赋值到 short 型静态字段。	6D00 0B00 - sput-short v0, Test3.ss1:S // field@000b 将 v0 寄存器中的值赋值到 Test3 的 short 型静态字段 ss1（字段表#BH 条目）。
6E	invoke-virtual {*参数*},	调用带*参数*的虚拟方法。	6E53 0600 0421 - invoke-virtual { v4, v0, v1, v2, v3}, Test

	方法名		2.method5:(IIII)V // method@0006 调用 Test2 的 method5（方法表#6 条目）方法，该指令共有 5 个参数（操作码第二个字节的 4 个最高有效位 5）[注5]。参数 v4 是"this"实例，v0, v1, v2, v3 是 method5 方法的参数，(IIII)V 的 4 个 I 分表表示 4 个 int 型参数，V 表示返回值为 void。
6F	invoke-super {*参数*}, *方法名*	调用带*参数*的直接父类的虚拟方法。	6F10 A601 0100 invoke-super {v1},java.io.FilterOutputStream.close:()V // method@01a6 调用java.io.FilterOutputStream 的 close（方法表#1A6 条目）方法，参数 v1 是"this"实例。()V 表示 close 方法没有参数，V 表示返回值为 void。
70	invoke-direct {*参数*}, *方法名*	不解析直接调用带*参数*的方法。	7010 0800 0100 - invoke-direct {v1}, java.lang.Object.<init>:()V // method@0008 调用 java.lang.Object 的<init>（方法表#8 条目）方法，参数 v1 是"this"实例[注5]。()V 表示<init>方法没有参数，V 表示返回值为 void。
71	invoke-static {*参数*}, *方法名*	调用带*参数*的静态方法。	7110 3400 0400 - invoke-static {v4}, java.lang.Integer.parseInt:(Ljava/lang/String;)I // method@0034 调用java.lang.Integer 的parseInt（方法表#34 条目）静态方法，该指令只有 1 个参数 v4[注5]，(Ljava/lang/String;)I 中的 Ljava/lang/String;表示 parseInt 方法需要 String 类型的参数，I 表示返回值为 int 型。
72	invoke-interface {*参数*}, *方法名*	调用带*参数*的接口方法。	7240 2102 3154 invoke-interface {v1, v3, v4, v5}, mwfw.IReceivingProtocolAdapter.receivePackage:(ILjava/lang/String;Ljava/io/InputStream;)Z // method@0221 调用mwfw.IReceivingProtocolAdapter 接口的 receivePackage 方法（方法表#221 条目），该指令共有 4

			个参数[注5]，参数 v1 是"this"实例，v3,v4,v5 是 receivePackage 方法的参数，(ILjava/lang/String;Ljava/io/InputStream;)Z 中的 I 表示 int 型参数，Ljava/lang/String;表示 String 类型参数，Ljava/io/InputStream;表示 InputStream 类型参数，Z 表示返回值为 boolean 型。
73	unused_73	未使用	
74	invoke-virtual/range {vx..vy}, *方法名*	调用以寄存器范围为参数的虚拟方法。该指令第一个寄存器和寄存器的数量将传递给方法。	7403 0600 1300 - invoke-virtual {v19..v21}, Test2.method5:(IIII)V // method@0006 调用 Test2 的 method5（方法表#6 条目）方法，该指令共有 3 个参数。参数 v19 是"this"实例，v20,v21 是 method5 方法的参数，(IIII)V 的 4 个 I 分表表示 4 个 int 型参数，V 表示返回值为 void。
75	invoke-super/range {vx..vy}, *方法名*	调用以寄存器范围为参数的直接父类的虚拟方法。该指令第一个寄存器和寄存器的数量将会传递给方法。	7501 A601 0100 invoke-super {v1},java.io.FilterOutputStream.close:()V // method@01a6 调用 java.io.FilterOutputStream 的 close（方法表#1A6 条目）方法，参数 v1 是"this"实例。()V 表示 close 方法没有参数，V 表示返回值为 void。
76	invoke-direct/range {vx..vy}, *方法名*	不解析直接调用以寄存器范围为参数的方法。该指令第一个寄存器和寄存器的数量将会传递给方法。	7603 3A00 1300 - invoke-direct/range {v19..21},java.lang.Object.<init>:()V // method@003a 调用 java.lang.Object 的<init>（方法表#3A 条目）方法，参数 v19 是"this"实例（操作码第五、第六字节表示范围从 v19 开始，第二个字节为 03 表示传入了 3 个参数），()V 表示<init>方法没有参数，V 表示返回值为 void。
77	invoke-static/range {vx..vy}, *方法名*	调用以寄存器范围为参数的静态方法。该指令第一个寄存器和寄存器的数量将会传递给方法。	7703 3A00 1300 - invoke-static/range {v19..21},java.lang.Integer.parseInt:(Ljava/lang/String;)I // method@0034 调用 java.lang.Integer 的 parseInt（方法表#34 条目）静态方法，参数 v19 是"this"实例（操作码第五、

			第六字节表示范围从 v19 开始，第二个字节为 03 表示传入了 3 个参数），(Ljava/lang/String;)I 中的 Ljava/lang/String;表示 parseInt 方法需要 String 类型的参数，I 表示返回值为 int 型。
78	invoke-interface-range {vx..vy}, *方法名*	调用以寄存器范围为参数的接口方法。该指令第一个寄存器和寄存器的数量将会传递给方法。	7840 2102 0100 invoke-interface {v1..v4}, mwfw.IReceivingProtocolAdapter.receivePackage:(ILjava/lang/String;Ljava/io/InputStream;)Z // method@0221 调用 mwfw.IReceivingProtocolAdapter 接口的 receivePackage 方法（方法表#221 条目），该指令共有 4 个参数[注5]，参数 v1 是"this"实例，v2,v3,v4 是 receivePackage 方法的参数，(ILjava/lang/String;Ljava/io/InputStream;)Z 中的 I 表示 int 型参数，Ljava/lang/String;表示 String 类型参数，Ljava/io/InputStream;表示 InputStream 类型参数，Z 表示返回值为 boolean 型。
79	unused_79	未使用	
7A	unused_7A	未使用	
7B	neg-int vx, vy	计算 vx = -vy 并将结果存入 vx。	7B01 - neg-int v1,v0 计算-v0 并将结果存入 v1。
7C	not-int vx, vy	未知[注4]	
7D	neg-long vx, vy	计算 vx,vx+1 = -(vy,vy+1) 并将结果存入 vx,vx+1。	7D02 - neg-long v2,v0 计算-(v0,v1) 并将结果存入(v2,v3)。
7E	not-long vx, vy	未知[注4]	
7F	neg-float vx, vy	计算 vx = -vy 并将结果存入 vx。	7F01 - neg-float v1,v0 计算-v0 并将结果存入 v1。
80	neg-double vx, vy	计算 vx,vx+1=-(vy,vy+1) 并将结果存入 vx,vx+1。	8002 - neg-double v2,v0 计算-(v0,v1) 并将结果存入(v2,v3)。
81	int-to-long vx, vy	转换 vy 寄存器中的 int 型值为 long 型值存入 vx,vx+1。	8106 - int-to-long v6, v0 转换 v0 寄存器中的 int 型值为 long 型值存入 v6,v7。
82	int-to-floa	转换 vy 寄存器中的 int 型	8206 - int-to-float v6, v0

	t vx, vy	值为 float 型值存入 vx。	转换 v0 寄存器中的 int 型值为 float 型值存入 v6。
83	int-to-double vx, vy	转换 vy 寄存器中的 int 型值为 double 型值存入 vx, vx+1。	8306 - int-to-double v6, v0 转换 v0 寄存器中的 int 型值为 double 型值存入 v6,v7。
84	long-to-int vx, vy	转换 vy,vy+1 寄存器中的 long 型值为 int 型值存入 vx。	8424 - long-to-int v4, v2 转换 v2,v3 寄存器中的 long 型值为 int 型值存入 v4。
85	long-to-float vx, vy	转换 vy,vy+1 寄存器中的 long 型值为 float 型值存入 vx。	8510 - long-to-float v0, v1 转换 v1,v2 寄存器中的 long 型值为 float 型值存入 v0。
86	long-to-double vx, vy	转换 vy,vy+1 寄存器中的 long 型值为 double 型值存入 vx,vx+1。	8610 - long-to-double v0, v1 转换 v1,vy2 寄存器中的 long 型值为 double 型值存入 v0,v1。
87	float-to-int vx, vy	转换 vy 寄存器中的 float 型值为 int 型值存入 vx。	8730 - float-to-int v0, v3 转换 v3 寄存器中的 float 型值为 int 型值存入 v0。
88	float-to-long vx, vy	转换 vy 寄存器中的 float 型值为 long 型值存入 vx, vx+1。	8830 - float-to-long v0, v3 转换 v3 寄存器中的 float 型值为 long 型值存入 v0,v1。
89	float-to-double vx, vy	转换 vy 寄存器中的 float 型值为 double 型值存入 vx,vx+1。	8930 - float-to-double v0, v3 转换 v3 寄存器中的 float 型值为 double 型值存入 v0,v1。
8A	double-to-int vx, vy	转换 vy,vy+1 寄存器中的 double 型值为 int 型值存入 vx。	8A40 - double-to-int v0, v4 转换 v4,v5 寄存器中的 double 型值为 int 型值存入 v0。
8B	double-to-long vx, vy	转换 vy,vy+1 寄存器中的 double 型值为 long 型值存入 vx,vx+1。	8B40 - double-to-long v0, v4 转换 v4,v5 寄存器中的 double 型值为 long 型值存入 v0,v1。
8C	double-to-float vx, vy	转换 vy,vy+1 寄存器中的 double 型值为 float 型值存入 vx。	8C40 - double-to-float v0, v4 转换 v4,v5 寄存器中的 double 型值为 float 型值存入 v0。
8D	int-to-byte vx, vy	转换 vy 寄存器中的 int 型值为 byte 型值存入 vx。	8D00 - int-to-byte v0, v0 转换 v0 寄存器中的 int 型值为 byte 型值存入 v0。
8E	int-to-char vx, vy	转换 vy 寄存器中的 int 型值为 char 型值存入 vx。	8E33 - int-to-char v3, v3 转换 v3 寄存器中的 int 型值为 char 型值存入 v3。
8F	int-to-short vx, vy	转换 vy 寄存器中的 int 型值为 short 型值存入 vx。	8F00 - int-to-short v3, v0 转换 v0 寄存器中的 int 型值为 short 型值存入 v0。

90	add-int vx, vy, vz	计算 vy + vz 并将结果存入 vx。	9000 0203 - add-int v0, v2, v3 计算 v2 + v3 并将结果存入 v0[注4]。
91	sub-int vx, vy, vz	计算 vy - vz 并将结果存入 vx。	9100 0203 - sub-int v0, v2, v3 计算 v2 – v3 并将结果存入 v0。
92	mul-int vx, vy, vz	计算 vy * vz 并将结果存入 vx。	9200 0203 - mul-int v0,v2,v3 计算 v2 * w3 并将结果存入 v0。
93	div-int vx, vy, vz	计算 vy / vz 并将结果存入 vx。	9303 0001 - div-int v3, v0, v1 计算 v0 / v1 并将结果存入 v3。
94	rem-int vx, vy, vz	计算 vy % vz 并将结果存入 vx。	9400 0203 - rem-int v0, v2, v3 计算 v3 % v2 并将结果存入 v0。
95	and-int vx, vy, vz	计算vy 与 vz并将结果存入 vx。	9503 0001 - and-int v3, v0, v1 计算 v0 与 v1 并将结果存入 v3。
96	or-int vx, vy, vz	计算vy 或 vz并将结果存入 vx。	9603 0001 - or-int v3, v0, v1 计算 v0 或 v1 并将结果存入 v3。
97	xor-int vx, vy, vz	计算vy 异或 vz并将结果存入 vx。	9703 0001 - xor-int v3, v0, v1 计算 v0 异或 v1 并将结果存入 v3。
98	shl-int vx, vy, vz	左移 vy，vz 指定移动的位置，结果存入 vx。	9802 0001 - shl-int v2, v0, v1 以 v1 指定的位置左移 v0，结果存入 v2。
99	shr-int vx, vy, vz	右移 vy，vz 指定移动的位置，结果存入 vx。	9902 0001 - shr-int v2, v0, v1 以 v1 指定的位置右移 v0，结果存入 v2。
9A	ushr-int vx, vy, vz	无符号右移 vy，vz 指定移动的位置，结果存入 vx。	9A02 0001 - ushr-int v2, v0, v1 以 v1 指定的位置无符号右移 v0，结果存入 v2。
9B	add-long vx, vy, vz	计算 vy,vy+1 + vz,vz+1 并将结果存入 vx,vx+1[注1]。	9B00 0305 - add-long v0, v3, v5 计算 v3,v4 + v5,v6 并将结果存入 v0,v1。
9C	sub-long vx, vy, vz	计算 vy,vy+1 - vz,vz+1 并将结果存入 vx,vx+1[注1]。	9C00 0305 - sub-long v0, v3, v5 计算 v3,v4 - v5,v6 并将结果存入 v0,v1。
9D	mul-long vx, vy, vz	计算 vy,vy+1 * vz,vz+1 并将结果存入 vx,vx+1[注1]。	9D00 0305 - mul-long v0, v3, v5 计算 v3,v4 * v5,v6 并将结果存入 v0,v1。
9E	div-long vx, vy, vz	计算 vy,vy+1 / vz,vz+1 并将结果存入 vx,vx+1[注1]。	9E06 0002 - div-long v6, v0, v2 计算 v0,v1 / v2,v3 并将结果存入 v

			6,v7。
9F	rem-long vx, vy, vz	计算 vy,vy+1 % vz,vz+1 并将结果存入 vx,vx+1 [注1]。	9F06 0002 - rem-long v6, v0, v2 计算 v0,v1 % v2,v3 并将结果存入 v6,v7。
A0	and-long vx, vy, vz	计算 vy,vy+1 与 vz,vz+1 并将结果存入 vx,vx+1 [注1]。	A006 0002 - and-long v6, v0, v2 计算 v0,v1 与 v2,v3 并将结果存入 v6,v7。
A1	or-long vx, vy, vz	计算 vy,vy+1 或 vz,vz+1 并将结果存入 vx,vx+1 [注1]。	A106 0002 - or-long v6, v0, v2 计算 v0,v1 或 v2,v3 并将结果存入 v6,v7。
A2	xor-long vx, vy, vz	计算 vy,vy+1 异或 vz,vz+1 并将结果存入 vx,vx+1 [注1]。	A206 0002 - xor-long v6, v0, v2 计算 v0,v1 异或 v2,v3 并将结果存入 v6,v7。
A3	shl-long vx, vy, vz	左移 vy,vy+1，vz 指定移动的位置，结果存入 vx,vx+1 [注1]。	A302 0004 - shl-long v2, v0, v4 以 v4 指定的位置左移 v0,v1，结果存入 v2,v3。
A4	shr-long vx, vy, vz	右移 vy,vy+1，vz 指定移动的位置，结果存入 vx,vx+1 [注1]。	A402 0004 - shr-long v2, v0, v4 以 v4 指定的位置右移 v0,v1，结果存入 v2,v3。
A5	ushr-long vx, vy, vz	无符号右移 vy,vy+1，vz 指定移动的位置，结果存入 vx,vx+1 [注1]。	A502 0004 - ushr-long v2, v0, v4 以 v4 指定的位置无符号右移 v0,v1，结果存入 v2,v3。
A6	add-float vx, vy, vz	计算 vy + vz 并将结果存入 vx。	A600 0203 - add-float v0, v2, v3 计算 v2 + v3 并将结果存入 v0。
A7	sub-float vx, vy, vz	计算 vy - vz 并将结果存入 vx。	A700 0203 - sub-float v0, v2, v3 计算 v2 - v3 并将结果存入 v0。
A8	mul-float vx, vy, vz	计算 vy * vz 并将结果存入 vx。	A803 0001 - mul-float v3, v0, v1 计算 v0 * v1 并将结果存入 v3。
A9	div-float vx, vy, vz	计算 vy / vz 并将结果存入 vx。	A903 0001 - div-float v3, v0, v1 计算 v0 / v1 并将结果存入 v3。
AA	rem-float v	计算 vy % vz 并将结果存	AA03 0001 - rem-float v3, v0,

	x, vy, vz	入 vx。	v1 计算 v0 % v1 并将结果存入 v3。
AB	add-double vx, vy, vz	计算 vy,vy+1 + vz,vz+1 并将结果存入 vx,vx+1 [注1]。	AB00 0305 - add-double v0, v3, v5 计算 v3,v4 + v5,v6 并将结果存入 v0,v1。
AC	sub-double vx, vy, vz	计算 vy,vy+1 - vz,vz+1 并将结果存入 vx,vx+1 [注1]。	AC00 0305 - sub-double v0, v3, v5 计算 v3,v4 - v5,v6 并将结果存入 v0,v1。
AD	mul-double vx, vy, vz	计算 vy,vy+1 * vz,vz+1 并将结果存入 vx,vx+1 [注1]。	AD06 0002 - mul-double v6, v0, v2 计算 v0,v1 * v2,v3 并将结果存入 v6,v7。
AE	div-double vx, vy, vz	计算 vy,vy+1 / vz,vz+1 并将结果存入 vx,vx+1 [注1]。	AE06 0002 - div-double v6, v0, v2 计算 v0,v1 / v2,v3 并将结果存入 v6,v7。
AF	rem-double vx, vy, vz	计算 vy,vy+1 % vz,vz+1 并将结果存入 vx,vx+1 [注1]。	AF06 0002 - rem-double v6, v0, v2 计算 v0,v1 % v2,v3 并将结果存入 v6,v7。
B0	add-int/2addr vx, vy	计算 vx + vy 并将结果存入 vx。	B010 - add-int/2addr v0,v1 计算 v0 + v1 并将结果存入 v0。
B1	sub-int/2addr vx, vy	计算 vx - vy 并将结果存入 vx。	B140 - sub-int/2addr v0, v4 计算 v0 – v4 并将结果存入 v0。
B2	mul-int/2addr vx, vy	计算 vx * vy 并将结果存入 vx。	B210 - mul-int/2addr v0, v1 计算 v0 * v1 并将结果存入 v0。
B3	div-int/2addr vx, vy	计算 vx / vy 并将结果存入 vx。	B310 - div-int/2addr v0, v1 计算 v0 / v1 并将结果存入 v0。
B4	rem-int/2addr vx, vy	计算 vx % vy 并将结果存入 vx。	B410 - rem-int/2addr v0, v1 计算 v0 % v1 并将结果存入 v0。
B5	and-int/2addr vx, vy	计算 vx 与 vy 并将结果存入 vx。	B510 - and-int/2addr v0, v1 计算 v0 与 v1 并将结果存入 v0。
B6	or-int/2addr vx, vy	计算 vx 或 vy 并将结果存入 vx。	B610 - or-int/2addr v0, v1 计算 v0 或 v1 并将结果存入 v0。
B7	xor-int/2addr vx, vy	计算 vx 异或 vy 并将结果存入 vx。	B710 - xor-int/2addr v0, v1 计算 v0 异或 v1 并将结果存入 v0。
B8	shl-int/2addr vx, vy	左移 vx，vy 指定移动的位置，并将结果存入 vx。	B810 - shl-int/2addr v0, v1 以 v1 指定的位置左移 v0，结果存入 v0。

B9	shr-int/2addr vx, vy	右移 vx，vy 指定移动的位置，并将结果存入 vx。	B910 - shr-int/2addr v0, v1 以 v1 指定的位置右移 v0，结果存入 v0。
BA	ushr-int/2addr vx, vy	无符号右移 vx，vy 指定移动的位置，并将结果存入 vx。	BA10 - ushr-int/2addr v0, v1 以 v1 指定的位置无符号右移 v0，结果存入 v0。
BB	add-long/2addr vx, vy	计算 vx,vx+1 + vy,vy+1 并将结果存入 vx,vx+1 注1。	BB20 - add-long/2addr v0, v2 计算 v0,v1 + v2,v3 并将结果存入 v0,v1。
BC	sub-long/2addr vx, vy	计算 vx,vx+1 - vy,vy+1 并将结果存入 vx,vx+1 注1。	BC70 - sub-long/2addr v0, v7 计算 v0,v1 - v7,v8 并将结果存入 v0,v1。
BD	mul-long/2addr vx, vy	计算 vx,vx+1 * vy,vy+1 并将结果存入 vx,vx+1 注1。	BD70 - mul-long/2addr v0, v7 计算 v0,v1 * v7,v8 并将结果存入 v0,v1。
BE	div-long/2addr vx, vy	计算 vx,vx+1 / vy,vy+1 并将结果存入 vx,vx+1 注1。	BE20 - div-long/2addr v0, v2 计算 v0,v1 / v2,v3 并将结果存入 v0,v1。
BF	rem-long/2addr vx, vy	计算 vx,vx+1 % vy,vy+1 并将结果存入 vx,vx+1 注1。	BF20 - rem-long/2addr v0, v2 计算 v0,v1 % v2,v3 并将结果存入 v0,v1。
C0	and-long/2addr vx, vy	计算 vx,vx+1 与 vy,vy+1 并将结果存入 vx,vx+1 注1。	C020 - and-long/2addr v0, v2 计算 v0,v1 与 v2,v3 并将结果存入 v0,v1。
C1	or-long/2addr vx, vy	计算 vx,vx+1 或 vy,vy+1 并将结果存入 vx,vx+1 注1。	C120 - or-long/2addr v0, v2 计算 v0,v1 或 v2,v3 并将结果存入 v0,v1。
C2	xor-long/2addr vx, vy	计算 vx,vx+1 异或 vy,vy+1 并将结果存入 vx,vx+1 注1。	C220 - xor-long/2addr v0, v2 计算 v0,v1 异或 v2,v3 并将结果存入 v0,v1。
C3	shl-long/2addr vx, vy	左移 vx,vx+1，vy 指定移动的位置，并将结果存入 vx,vx+1。	C320 - shl-long/2addr v0, v2 以 v2 指定的位置左移 v0,v1，结果存入 v0,v1。
C4	shr-long/2addr vx, vy	右移 vx,vx+1，vy 指定移动的位置，并将结果存入 vx,vx+1。	C420 - shr-long/2addr v0, v2 以 v2 指定的位置右移 v0,v1，结果存入 v0,v1。
C5	ushr-long/2addr vx, vy	无符号右移 vx,vx+1，vy 指定移动的位置，并将结果存入 vx,vx+1。	C520 - ushr-long/2addr v0, v2 以 v2 指定的位置无符号右移 v0,v1，结果存入 v0,v1。
C6	add-float/2addr vx, vy	计算 vx + vy 并将结果存入 vx。	C640 - add-float/2addr v0,v4 计算 v0 + v4 并将结果存入 v0。

C7	sub-float/2addr vx, vy	计算 vx - vy 并将结果存入 vx。	C740 - sub-float/2addr v0,v4 计算 v0 - v4 并将结果存入 v0。
C8	mul-float/2addr vx, vy	计算 vx * vy 并将结果存入 vx。	C810 - mul-float/2addr v0, v1 计算 v0 * v1 并将结果存入 v0。
C9	div-float/2addr vx, vy	计算 vx / vy 并将结果存入 vx。	C910 - div-float/2addr v0, v1 计算 v0 / v1 并将结果存入 v0。
CA	rem-float/2addr vx, vy	计算 vx % vy 并将结果存入 vx。	CA10 - rem-float/2addr v0, v1 计算 v0 % v1 并将结果存入 v0。
CB	add-double/2addr vx, vy	计算 vx,vx+1 + vy,vy+1 并将结果存入 vx,vx+1 [注1]。	CB70 - add-double/2addr v0, v7 计算 v0,v1 + v7,v8 并将结果存入 v0,v1。
CC	sub-double/2addr vx, vy	计算 vx,vx+1 - vy,vy+1 并将结果存入 vx,vx+1 [注1]。	CC70 - sub-double/2addr v0, v7 计算 v0,v1 - v7,v8 并将结果存入 v0,v1。
CD	mul-double/2addr vx, vy	计算 vx,vx+1 * vy,vy+1 并将结果存入 vx,vx+1 [注1]。	CD20 - mul-double/2addr v0, v2 计算 v0,v1 * v2,v3 并将结果存入 v0,v1。
CE	div-double/2addr vx, vy	计算 vx,vx+1 / vy,vy+1 并将结果存入 vx,vx+1 [注1]。	CE20 - div-double/2addr v0, v2 计算 v0,v1 / v2,v3 并将结果存入 v0,v1。
CF	rem-double/2addr vx, vy	计算 vx,vx+1 % vy,vy+1 并将结果存入 vx,vx+1 [注1]。	CF20 - rem-double/2addr v0, v2 计算 v0,v1 % v2,v3 并将结果存入 v0,v1。
D0	add-int/lit16 vx, vy, lit16	计算 vy + lit16 并将结果存入 vx。	D001 D204 - add-int/lit16 v1, v0, #int 1234 // #04d2 计算 v0 + 1234 并将结果存入 v1。
D1	sub-int/lit16 vx, vy, lit16	计算 vy - lit16 并将结果存入 vx。	D101 D204 - sub-int/lit16 v1, v0, #int 1234 // #04d2 计算 v0 - 1234 并将结果存入 v1。
D2	mul-int/lit16 vx, vy, lit16	计算 vy * lit16 并将结果存入 vx。	D201 D204 - mul-int/lit16 v1, v0, #int 1234 // #04d2 计算 v0 * 1234 并将结果存入 v1。
D3	div-int/lit16 vx, vy, lit16	计算 vy / lit16 并将结果存入 vx。	D301 D204 - div-int/lit16 v1, v0, #int 1234 // #04d2 计算 v0 / 1234 并将结果存入 v1。
D4	rem-int/lit16 vx, vy, lit16	计算 vy % lit16 并将结果存入 vx。	D401 D204 - rem-int/lit16 v1, v0, #int 1234 // #04d2 计算 v0 % 1234 并将结果存入 v1。
D5	and-int/lit16 vx, vy, lit16	计算 vy 与 lit16 并将结果存入 vx。	D501 D204 - and-int/lit16 v1, v0, #int 1234 // #04d2 计算 v0 与 1234 并将结果存入 v1。

D6	or-int/lit16 vx, vy, lit16	计算 vy 或 lit16 并将结果存入 vx。	D601 D204 - or-int/lit16 v1, v0, #int 1234 // #04d2 计算 v0 或 1234 并将结果存入 v1。
D7	xor-int/lit16 vx, vy, lit16	计算 vy 异或 lit16 并将结果存入 vx。	D701 D204 - xor-int/lit16 v1, v0, #int 1234 // #04d2 计算 v0 异或 1234 并将结果存入 v1。
D8	add-int/lit8 vx, vy, lit8	计算 vy + lit8 并将结果存入 vx。	D800 0201 - add-int/lit8 v0,v2, #int1 计算 v2 + 1 并将结果存入 v0。
D9	sub-int/lit8 vx, vy, lit8	计算 vy - lit8 并将结果存入 vx。	D900 0201 - sub-int/lit8 v0,v2, #int1 计算 v2 - 1 并将结果存入 v0。
DA	mul-int/lit8 vx, vy, lit8	计算 vy * lit8 并将结果存入 vx。	DA00 0002 - mul-int/lit8 v0,v0, #int2 计算 v0 * 2 并将结果存入 v0。
DB	div-int/lit8 vx, vy, lit8	计算 vy / lit8 并将结果存入 vx。	DB00 0203 - mul-int/lit8 v0,v2, #int3 计算 v2 / 3 并将结果存入 v0。
DC	rem-int/lit8 vx, vy, lit8	计算 vy % lit8 并将结果存入 vx。	DC00 0203 - rem-int/lit8 v0,v2, #int3 计算 v2 % 3 并将结果存入 v0。
DD	and-int/lit8 vx, vy, lit8	计算 vy 与 lit8 并将结果存入 vx。	DD00 0203 - and-int/lit8 v0,v2, #int3 计算 v2 与 3 并将结果存入 v0。
DE	or-int/lit8 vx, vy, lit8	计算 vy 或 lit8 并将结果存入 vx。	DE00 0203 - or-int/lit8 v0, v2, #int 3 计算 v2 或 3 并将结果存入 v0。
DF	xor-int/lit8 vx, vy, lit8	计算 vy 异或 lit8 并将结果存入 vx。	DF00 0203 | 0008: xor-int/lit8 v0, v2, #int 3 计算 v2 异或 3 并将结果存入 v0。
E0	shl-int/lit8 vx, vy, lit8	左移 vy，lit8 指定移动的位置，并将结果存入 vx。	E001 0001 - shl-int/lit8 v1, v0, #int 1 将 v0 左移 1 位，结果存入 v1。
E1	shr-int/lit8 vx, vy, lit8	右移 vy，lit8 指定移动的位置，并将结果存入 vx。	E101 0001 - shr-int/lit8 v1, v0, #int 1 将 v0 右移 1 位，结果存入 v1。
E2	ushr-int/lit8 vx, vy, lit8	无符号右移 vy，lit8 指定移动的位置，并将结果存入 vx。	E201 0001 - ushr-int/lit8 v1, v0, #int 1 将 v0 无符号右移 1 位，结果存入 v1。
E3	unused_E3	未使用	
E4	unused_E4	未使用	

E5	unused_E5	未使用	
E6	unused_E6	未使用	
E7	unused_E7	未使用	
E8	unused_E8	未使用	
E9	unused_E9	未使用	
EA	unused_EA	未使用	
EB	unused_EB	未使用	
EC	unused_EC	未使用	
ED	unused_ED	未使用	
EE	execute-inline {*参数*}, *内联 ID*	根据*内联 ID*[注6]执行内联方法。	EE20 0300 0100 - execute-inline {v1, v0}, inline #0003 执行内联方法#3，参数 v1,v0，其中参数 v1 为"this"的实例，v0 是方法的参数。
EF	unused_EF	未使用	
F0	invoke-direct-empty	用于空方法的占位符，如 Object.<init>。这相当于正常执行了 nop 指令[注6]。	F010 F608 0000 - invoke-direct-empty {v0}, Ljava/lang/Object;.<init>:()V // method@08f6 替代空方法 java/lang/Object;<init>。
F1	unused_F1	未使用	
F2	iget-quick vx, vy, *偏移量*	获取 vy 寄存器中实例指向+*偏移位置*的数据区的值，存入 vx[注6]。	F221 1000 - iget-quick v1, v2, [obj+0010] 获取 v2 寄存器中的实例指向+10H 位置的数据区的值，存入 v1。
F3	iget-wide-quick vx, vy, *偏移量*	获取 vy 寄存器中实例指向+*偏移位置*的数据区的值，存入 vx,vx+1[注6]。	F364 3001 - iget-wide-quick v4, v6, [obj+0130] 获取 v6 寄存器中的实例指向+130H 位置的数据区的值，存入 v4,v5。
F4	iget-object-quick vx, vy, *偏移量*	获取 vy 寄存器中实例指向+*偏移位置*的数据区的对象引用，存入 vx[注6]。	F431 0C00 - iget-object-quick v1, v3, [obj+000c] 获取 v3 寄存器中的实例指向+0CH 位置的数据区的对象引用，存入 v1。
F5	iput-quick vx, vy, *偏移量*	将 vx 寄存器中的值存入 vy 寄存器中的实例指向+*偏移位置*的数据区[注6]。	F521 1000 - iput-quick v1, v2, [obj+0010] 将 v1 寄存器中的值存入 v2 寄存器中的实例指向+10H 位置的数据区。
F6	iput-wide-quick vx, vy, *偏移量*	将 vx,vx+1 寄存器中的值存入 vy 寄存器中的实例指向+*偏移位置*的数据区[注6]。	F652 7001 - iput-wide-quick v2, v5, [obj+0170] 将 v2,v3 寄存器中的值存入 v5 寄存器

			中的实例指向+170H 位置的数据区。
F7	iput-object-quick vx, vy, *偏移量*	将 vx 寄存器中的对象引用存入 vy 寄存器中的实例指向+*偏移位置*的数据区[注6]。	F701 4C00 - iput-object-quick v1, v0, [obj+004c] 将 v1 寄存器中的对象引用存入 v0 寄存器中的实例指向+4CH 位置的数据区。
F8	invoke-virtual-quick {*参数*}, *虚拟表偏移量*	调用虚拟方法，使用目标对象虚拟表[注6]。	F820 B800 CF00 - invoke-virtual-quick {v15, v12}, vtable #00b8 调用虚拟方法，目标对象的实例指向位于 v15 寄存器，方法位于虚拟表#B8 条目，方法所需的参数位于 v12。
F9	invoke-virtual-quick/range {*参数范围*}, *虚拟表偏移量*	调用虚拟方法，使用目标对象虚拟表[注6]。	F906 1800 0000 - invoke-virtual-quick/range {v0..v5},vtable #0018 调用虚拟方法，目标对象的实例指向位于 v0 寄存器，方法位于虚拟表#18H 条目，方法所需的参数位于 v1..v5。
FA	invoke-super-quick {*参数*}, *虚拟表偏移量*	调用父类虚拟方法，使用目标对象的直接父类的虚拟表[注6]。	FA40 8100 3254 - invoke-super-quick {v2, v3, v4, v5}, vtable #0081 调用父类虚拟方法，目标对象的实例指向位于 v2 寄存器，方法位于虚拟表#81H 条目，方法所需的参数位于 v3, v4,v5。
FB	invoke-super-quick/range {*参数范围*}, *虚拟表偏移量*	调用父类虚拟方法，使用目标对象的直接父类的虚拟表[注6]。	F906 1B00 0000 - invoke-super-quick/range {v0..v5}, vtable #001b 调用父类虚拟方法，目标对象的实例指向位于 v0 寄存器，方法位于虚拟表#1B 条目，方法所需的参数位于 v1..v5。
FC	unused_FC	未使用	
FD	unused_FD	未使用	
FE	unused_FE	未使用	
FF	unused_FF	未使用	

注1： Double 和 long 值占用两个寄存器。（例：在 vy 地址上的值位于 vy,vy+1 寄存器）

注2： 偏移量可以是正或负，从指令起始字节起计算偏移量。偏移量在（2 字节每 1 偏移量递增/递减）时解释执行。负偏移量用二进制补码格式存储。偏移量当前位置是指令起始字节。

注3： 比较操作，如果第一个操作数大于第二个操作数返回正值；如果两者相等，返回 0；如果第一个操作数小于第二个操作数，返回负值。

注4： 正常使用没见到过的，从 Android opcode constant list 引入。

注5： 调用参数表的编译比较诡异。如果参数的数量大于 4 并且%4=1，第 5（第 9 或其他%4=1 的）个参数将编译在指令字节的下一个字节的 4 个最低位。奇怪的是，有一种情况不使用这种编译：方法有 4 个参数但用于编译单一参数，指令字节的下一个字节的 4 个最低位空置，将会编译为 40 而不是 04。

注6： 这是一个不安全的指令，仅适用于 ODEX 文件。

表中的 vx、vy、vz 表示某个 Dalvik 寄存器。根据不同指令可以访问 16、256 或 64K 寄存器。表中 lit4、lit8、lit16、lit32、lit64 表示字面值（直接赋值），数字是值所占用位的长度。

long 和 double 型的值占用两个寄存器，例：一个在 v0 寄存器的 double 值实际占用 v0,v1 两个寄存器。boolean 值的存储实际是 1 和 0，1 为真、0 为假；boolean 型的值实际是转成 int 型的值进行操作。所有例子的字节序都采用高位存储格式，例：0F00 0A00 的编译为 0F, 00, 0A, 00 存储。

上表的重要性不言而喻，不仅是分析 Dalvik 虚拟机字节码的重要官方参考资料，而且对我们分析将 smali 文件汇编为 dex 文件时出现的编译错误信息也有非常大的帮助，能够根据编译出错信息中操作码的编号信息定位出错的指令。

3.2 Dalvik 虚拟机字节码的类型、方法和字段的表示方法

3.2.1 类型

Dalvik 字节码有两种类型，基本类型和引用类型。对象和数组是引用类型，其它都是基本类型。

Dalvik 字节码类型描述符

描述符	类型
V	void，只能用于返回值类型
Z	boolean
B	byte
S	short
C	char
I	int
J	long（64 位）
F	float
D	double（64 位）
L	Java 类类型
[	数组类型

每个 Dalvik 寄存器都是 32 位大小，对于小于或者等于 32 位长度的类型来说，一个寄存器就可以存放该类型的值，而像 J、D 等 64 位的类型，它们的值是使用相邻两个寄存器来存储的，如 v0 与 v1、v3 与 v4 等。

Java 中的对象在 smali 中以 Lpackage/name/ObjectName;的形式表示。前面的 L 表示这是一个对象类型，package/name/表示该对象所在的包，ObjectName 是对象的名字，“;”表示对象名称的结束。相当于 java 中的 package.name.ObjectName。例 如：Ljava/lang/String;

相当于 java.lang.String。

“[”类型可以表示所有基本类型的数组。[I 表示一个整型一维数组，相当于 java 中的 int[]。对于多维数组，只要增加[就行了，[[I 相当于 int[][]，[[[I 相当于 int[][][]。注意每一维的最多 255 个。对象数组的表示：[Ljava/lang/String;表示一个 String 对象数组。

3.2.2 方法

方法调用的表示格式：`Lpackage/name/ObjectName;->MethodName(III)Z`。Lpackage/name/ObjectName;表示类型，MethodName 是方法名，III 为参数（在此是 3 个整型参数），Z 是返回类型（bool 型）。函数的参数是一个接一个的，中间没有隔开。

一个更复杂的例子：method(I[[IILjava/lang/String;[Ljava/lang/Object;)Ljava/lang/String;

在 java 中则为：String method(int, int[][], int, String, Object[])

3.2.3 字段

字段，即 java 中类的成员变量，表示格式：

`Lpackage/name/ObjectName;->FieldName:Ljava/lang/String;` 即包名，字段名和字段类型，字段名与字段类型是以冒号“:”分隔。

3.3 Dalvik 虚拟机字节码指令解析

在对 3.1 中的 Dalvik 虚拟机字节码指令表中涉及的指令进行分别解析前，先介绍 Dalvik 虚拟机字节码中寄存器的命名法。

3.3.1 两种不同的寄存器表示法

在 Dalvik 虚拟机字节码中寄存器的命名法中主要有 2 种：v 命名法和 p 命名法。假设一个函数使用到 M 个寄存器，并且该函数有 N 个入参，根据 Dalvik 虚拟机参数传递方式中的规定：入参使用最后的 N 个寄存器中，局部变量使用从 v0 开始的前 M-N 个寄存器。比如，某函数 A 使用了 5 个寄存器，2 个显式的整形参数，如果函数 A 是非静态方法，函数被调用时会传入一个隐式的对象引用，因此实际传入的参数个数是 3 个。根据传参规则，局部变量将使用前 2 个寄存器，参数会使用后 3 个寄存器。

v 命名法采用小写字母“v”开头的方式表示函数中用到的局部变量与参数，所有的寄存器命名从 v0 开始，依次递增。对于上文的函数 A，v 命名法会用到 v0、v1、v2、v3、v4 等 5 个寄存器，v0 与 v1 表示函数 A 的局部变量，v2 表示传入的隐式对象引用，v3 与 v4 表示实际传入的 2 个整形参数。

P 命名法对函数的局部变量寄存器命名没有影响，它的命名规则是：函数的入参从 p0 开始命名，依次递增。对于上文的函数 A，p 命名法会用到 v0、v1、p0、p1、p2 等 5 个寄存器，v0 与 v1 表示函数 A 的局部变量，p0 表示传入的隐式对象引用，p1 与 p2 表示实际传入的 2 个整形参数。此时，p0、p1、p2 实际上分别表示 v2、v3、v4，只是命名不一样而已。

在实际的 Smali 文件中，几乎都是使用了 p 命名法，主要原因是使用 p 命名法能够通过寄存器的名字前缀就能很容易判断寄存器到底是局部变量还是函数的入参。初次学习 smali 语法时容易对寄存器 p0 表示的意义出现混乱，这主要体现在静态方法和非静态方法中。其实只要理解 p 命名法的定义后就可以很清楚的理解。在 smali 语法中，在调用非静态方法时需要传入该方法所在对象的引用，因此此时 p0 表示的是传入的隐式对象引用，从 p1 开始才是实际传入的入参。但是在调用静态方法时，由于静态方法不需要构建对象的引用，因而也就不需要传入该方法所在对象的引用，因此此时从 p0 开始就是实际传入的入参。

在 Dalvik 指令中使用“v 加数字”的方法来索引寄存器，如：v0、v1、v15、v255，但每条指令使用的寄存器索引范围都有限制（因为 Dalvik 指令字节码必须字节对齐），这里我们使用一个大写字母来表示 4 位数据宽度的取值范围，如：指令 move vA, vB，目的寄存器 vA 可使用 v0 ~ v15 的寄存器，源寄存器 vB 可以使用 v0 ~ v15 寄存器。指令 move/from16 vAA, vBBBBB，目的寄存器 vAA 可使用 v0 ~ v255 的寄存器，源寄存器 vB 可以使用 v0 ~ v65535 寄存器。简而言之，当目的寄存器和源寄存器中有一个寄存器的编号大于 15 时，即需要加上 /from16 指令才能得到正确运行。初次学习 Smali 语法时也容易对这一点不能理解，不注意就会导致 Smali 文件汇编为 dex 文件的时候出现编译错误。比如，按照前面总结的 p 命名法，当 p0 实际表示的寄存器编号大于 15 时，此时 Smali 语句 `move v0, p0` 就会编译出错。

3.3.2 空指令

指令助记符	描述
nop	代码对齐，无实际操作

3.3.3 数据操作指令

指令助记符	描述
`move vA, vB`	将 vA 寄存器的内容赋值给 vB，非对象类型
`move/from16 vAA, vBBBB`	--
`move/16 vAAAA, vBBBB`	--
`move-wide vA, vB`	wide 后缀的指令会操作 64 位数据宽度，需使用两个寄存器组成寄存器对。如 move-wide v0, v2 会将 v2、v3 寄存器对内容赋值给 v0、v1 寄存器对。
`move-wide/from16 vAA, vBBBB`	--
`move-wide/16 vAAAA, vBBBB`	--
`move-object vA, vB`	将 vA 寄存器的内容赋值给 vB，对象类型
`move-object/from16 vAA, vBBBB`	--
`move-object/16 vAAAA, vBBBB`	--
`move-result vAA`	必须紧跟着调用指令 invoke 后面，把调用方法的返回值赋值给寄存器 vAA，操作非对象类型。
`move-result-wide vAA`	--
`move-result-object vAA`	操作对象类型。
`move-exception vAA`	必须作为异常捕捉的处理块的第一条指令，把捕捉到的异常类型对象赋值给 vAA 寄存器。

3.3.4 返回指令

指令助记符	描述
`return-void`	返回空类型指令。
`return vAA`	返回 32 位寄存器 VAA 内容，非对象类型数据。
`return-wide vAA`	返回 64 位内容数据，非对象类型数据。
`return-object vAA`	返回对象类型数据。

3.3.5 数据定义指令

指令助记符	描述
const/4 vA, #+B	将 4 位宽度的立即数带符号扩展到 32 位，赋值给 vA 寄存器。
const/16 vAA, #+BBBB	将 16 位宽度的立即数带符号扩展到 32 位，赋值给 vA 寄存器。
const vAA, #+BBBBBBBB	将 32 位宽度的立即数赋值给 vA 寄存器。
const/high16 vAA, #+BBBB0000	将 16 位宽度的立即数右边零扩展到 32 位，赋值给 vAA 寄存器。
const-wide/16 vAA, #+BBBB	将 16 位宽度的立即数带符号扩展到 64 位，赋给 vAA 寄存器对。
const-wide/32 vAA, #+BBBBBBBB	将 32 位宽度的立即数带符号扩展到 64 位，赋给 vAA 寄存器对。
const-wide vAA, #+BBBBBBBBBBBBBBBB	将 64 位宽度的立即数赋给 vAA 寄存器对。
const-wide/high16 vAA, #+BBBB000000000000	将 16 位宽度的立即数右边零扩展到 64 位，赋值给 vAA 寄存器。
const-string vAA, string@BBBB	将字符串常量的引用赋值给 vAA 寄存器。
const-string/jumbo vAA, string@BBBBBBBB	jumbo 后缀表示指令的寄存器的索引范围更大。
const-class vAA, type@BBBB	将一个类的引用赋值给 vAA 寄存器。

3.3.6 锁指令

指令助记符	描述
monitor-enter vAA	获取 vAA 寄存器引用的对象的同步锁。
monitor-exit vAA	释放 vAA 寄存器引用的对象的同步锁。

3.3.7 实例操作指令

指令助记符	描述
check-cast vAA, type@BBBB	把寄存器引用的对象转为指定的类型，如果不行则会抛出一个 ClassCastException 异常。
instance-of vA, vB, type@CCCC	判断 vB 寄存器的引用对象是否可以转化为指定类型，是则给 vA 寄存器赋 1，否则赋 0。
new-instance vAA, type@BBBB	构造一个指定类型的实例，并把引用赋给寄存器。

3.3.8 数组操作指令

指令助记符	描述
array-length vA, vB	获取 vB 引用的数组长度赋值给 vA
new-array vA, vB, type@CCCC	构造一个指定类型和大小的数组，并把引用赋值
filled-new-array {vC, vD, vE, vF, vG}, type@BBBB	构建一个指定类型的数组，数组大小由寄存器列表{vC,…,vG}的 长度指定，元素由寄存器列表赋值，

	初始化后，使用指令 move-result-object 获取构建的数组的引用。
filled-new-array/range {vCCCC .. vNNNN}, type@BBBB	寄存器列表使用连续的寄存器 vCCCC 到 vNNNN。
fill-array-data vAA, +BBBBBBBB *(with supplemental data as specified below in "fill-array-data-payload Format")*	使用指定的数据表来填充 vAA 指定的数组。

3.3.9 异常指令

指令助记符	描述
throw vAA	抛出一个指定类型的异常。

3.3.10 跳转指令

指令助记符	描述
goto +AA	无条件跳转到指定的指令。偏移量为 8 位宽度，且不能为零。
goto/16 +AAAA	偏移量为 16 位宽度，且不能为零。
goto/32 +AAAAAAAA	--
packed-switch vAA, +BBBBBBBB	根据+BBBBBBBB 给定的跳转偏移列表匹配 vAA 寄存器值，跳转到指定指令。偏移表中的匹配值是有规律递增的。
sparse-switch vAA, +BBBBBBBB	偏移表中的匹配值是无规律且可以被指定的。
if-*test* vA, vB, +CCCC |- if-eq |- if-ne |- if-lt |- if-ge |- if-gt |- if-le	条件跳转指令，比较指定两个寄存器 vA 和 vB 的值，满足条件后跳转到指定的指令。 条件：等于(eq)、不等于(ne)、小于(lt)，大于(gt)，小于等于(le)、大于等于(ge)。
if-*test*z vAA, +BBBB |- if-eqz |- if-nez |- if-ltz |- if-gez |- if-gtz |- if-lez	条件跳转指令，比较指定寄存器 vAA 的值与 0 大小，满足条件跳转到指定指令。

3.3.11 比较指令

cmp*kind* vAA, vBB, vCC |- cmpl-float *(lt bias)*	比较浮点数和长整型数的大小。如果 vBB 寄存器大于 vCC 寄存器，则结果为-1，相等结果则为 0，小

|- cmpg-float *(gt bias)* |- cmpl-double *(lt bias)* |- cmpg-double *(gt bias)* |- cmp-long	于结果则为 1。结果赋给 vAA 寄存器。

3.3.12 字段操作指令

arrayop vAA, vBB, vCC |- aget |- aget-*type* |- aput |- aput-*type*	字段操作指令用来对象实例的成员变量进行读与写操作的。分为数组字段、普通字段和静态字段。 vAA 寄存器存放读的结果或写的数据。 vBB 寄存器是数组的引用。 vCC 寄存器是数组读写元素的索引。 *type* *类型*后缀包括 wide、object、boolean、byte、char、short 类型。
iinstanceop vA, vB, field@CCCC |- iget |- iget-*type* |- iput |- iput-*type*	普通字段操作指令。
sstaticop vAA, field@BBBB |- sget |- sget-*type* |- sput |- sput-*type*	静态字段操作指令。

3.3.13 方法调用指令

invoke-*kind* {vC, vD, vE, vF, vG}, meth@BBBB |- invoke-virtual |- invoke-super |- invoke-direct |- invoke-static |- invoke-interface	调用指定的方法，{vC,…,vG}为传入方法的参数列表。具体使用哪种调用方式，视方法的对象类型和方法本身类型而定。 invoke-virtual 调用实例的虚方法，通常成员对象实例的方法都以该指令调用。 invoke-super 调用实例的父类方法。 invoke-direct 调用直接方法，通常私有方法都以该指令调用。 invoke-static 调用静态方法。 invoke-interface 调用接口的方法。
invoke-*kind*/range {vCCCC .. vNNNN}, meth@BBBB |- invoke-virtual/range |- invoke-super/range |- invoke-direct/range |- invoke-static/range |- invoke-interface/range	参数列表使用{vCCCC .. vNNNN}连续的寄存器列表。

3.3.14 数据转换

unop vA, vB |- neg-*type* |- not-*type* |- *type*-to-*type* |- int-to-byte |- int-to-char |- int-to-short	数据类型转换指令，*type* 类型后缀包括 int、long、float、double。

3.3.15 数据运算

binop vAA, vBB, vCC |- add-*type* |- sub-*type* |- mul-*type* |- div-*type* |- rem-*type* |- and-*type1* |- or-*type1* |- xor-*type1* |- shl-*type1* |- shr-*type1* |- ushr-*type1*	对寄存器 vBB 和寄存器 vCC 做算术运算，并把结果赋值给 vAA。 *type* 类型后缀包括 int、long、float、double,。 *type1* 类型后缀只包括 int、long。 add：加法 sub：减法 add：乘法 div： 除法 rem：取模(%) and：与 or：或 xor：异或 shl：有符号数左移 shr：有符号数右移 ushr：无符号数右移
binop/2addr vA, vB	寄存器 vA 和寄存器 vB 做算术运算，结果赋值给寄存器 vA。
binop/lit16 vA, vB, #+CCCC	寄存器 vB 与常量 CCCC 做算术运算，结果赋值给 vA。
binop/lit8 vAA, vBB, #+CC	寄存器 vB 与常量 CC 做算术运算，结果赋值给 vAA。

在以上指令中，在部分指令助记符后添加了 jumbo 后缀，这是在 Android 4.0 开始的扩展指令，增加了寄存器和常量的取值范围。需要引起注意的是，以上指令表中形如 VA 表示寄存器范围为 v0-v15，形如 VAA 表示寄存器范围为 v0-v255，这一点在理解指令时容易被忽略而导致修改 smali 代码时编译出错。比如方法调用指令 invoke 未添加/range 时传入方法的参数列表的寄存器需要在 v0-v15 范围内，如果不在范围内需要将不合格寄存器赋值给合格寄存器，然后再调用方法。

3.4 Smali 格式结构

3.4.1 文件格式

无论是普通类、抽象类、接口类或者内部类，在反编译出的代码中，它们都以单独的 Smali 文件来存放。每个 smali 文件头 3 行描述了当前类的一些信息，格式如下。

```
.class <访问权限> [修饰关键字] <类名>
.super <父类名>
.source <源文件名>
```

打开 HelloWorld.smali 文件，头 3 行代码如下。

```
.class public LHelloWorld;
.super Landroid/app/Activity;
.source "HelloWorld.java"
```

第 1 行“.class”指令指定了当前类的类名。在本例中，类的访问权限为 public，类名为“LHelloWorld;”，类名开头的 L 是遵循 Dalvik 字节码的相关约定，表示后面跟随的字符串为一个类。

第 2 行的“.super ”指令指定了当前类的父类。本例中的“LHelloWorld;”的父类为“Landroid/app/Activity;”。

第 3 行的“.source”指令指定了当前类的源文件名。经过混淆的 dex 文件，反编译出来的 smali 代码可能没有源文件信息，因此“.source”行的代码可能为空。

前 3 行代码过后就是类的主体部分了，一个类可以由多个字段或方法组成。

smali 文件中字段的声明使用“.field”指令。字段有静态字段与实例字段两种。静态字段的声明格式如下。

```
#static fields
.field <访问权限> static [修饰关键字] <字段名>:<字段类型>
```

baksmali 在生成 Smali 文件时，会在静态字段声明的起始处添加“static fields”注释，Smali 文件中的注释与 Dalvik 语法一样，也是以井号“#”开头。“.field”指令后面跟着的是访问权限，可以是 public、private、protected 之一。修饰关键字描述了字段的其它属性，如 synthetic。指令的最后是字段名与字段类型，使用冒号“:”分隔，语法上与 Dalvik 也是一样的。

实例字段的声明与静态字段类似，只是少了 static 关键字，它的格式如下。

```
#instance fields
.field <访问权限> [修饰关键字] <字段名>:<字段类型>
```

比如以下的实例字段声明。

```
#instance fields
.field private btn:Landroid/widget/Button;
```

第 1 行的“instance fields”是 baksmali 生成的注释，第 2 行表示一个私有字段 btn，它的类型为“Landroid/widget/Button;”。如果一个类中含有方法，那么类中必然会有相关方法的反汇编代码，Smali 文件中方法的声明使用“.method”指令。方法有直接方法与虚方法两种。直接方法的声明格式如下。

```
#direct methods
.method <访问权限> [修饰关键字] <方法原型>
```

```
    <.locals>
    [.parameter]
    [.prologue]
    [.line]
    <代码体>
.end method
```

“direct methods”是 baksmali 添加的注释，访问权限和修饰关键字与字段的描述相同，方法原型描述了方法的名称、参数与返回值。“.locals ”指定了使用的局部变量的个数。“.parameter”指定了方法的参数，与 Dalvik 语法中使用“.parameters”指定参数个数不同，每个“.parameter”指令表明使用一个参数，比如方法中有使用到 3 个参数，那么就会出现 3 条“.parameter”指令。“.prologue”指定了代码的开始处，混淆过的代码可能去掉了该指令。“.line”指定了该处指令在源代码中的行号，同样的，混淆过的代码可能去除了行号信息。

虚方法的声明与直接方法相同，只是起始处的注释为“virtual methods”。

如果一个类实现了接口，会在 smali 文件中使用“.implements”指令指出。相应的格式声明如下。

```
#interfaces
.implements <接口名>
```

“#interfaces”是 baksmali 添加的接口注释，“.implements”是接口关键字，后面的接口名是 DexClassDef 结构中 interfacesOff 字段指定的内容。

如果一个类使用了注解，会在 smali 文件中使用“.annotation”指令指出。注解的格式声明如下。

```
#annotations
.annotation [注解属性] <注解类名>
    [注解字段=值]
.endannotation
```

注解的作用范围可以是类、方法或字段。如果注解的作用范围是类，“.annotation”指令会直接定义在 smali 文件中，如果是方法或字段，“.annotation”指令则会包含在方法或字段定义中。例如下面的代码。

```
#instance fields
.field public sayWhat:Ljava/lang/String;
.annotation runtime LMyAnnoField;
    info="Hellomyfriend"
.end annotation
.end field
```

实例字段 sayWhat 为 String 类型，它使用了 MyAnnoField 注解，注解字段 info 值为“Hello myfriend”。将其转换为 Java 代码为：

```
    @MyAnnoField(info="Hellomyfriend")
    public String sayWhat;
```

3.4.2 类的结构

无论普通类、抽象类、接口类还是内部类，反编译的时候会为每个类单独生成一个 Smali 文件，但是内部类相存在相对比较特殊的地方。

1) 内部类的文件是“[*外部类*]$[*内部类*].smali”的形式来命名的，匿名内部类文件以“[*外部类*]$[*数字*].smali”来命名。
2) 内部类访问外部类的私有方法和变量时，都要通过编译器生成的“合成方法”来间接访问。
3) 编译器会把外部类的引用作为第一个参数插入到会内部类的构造器参数列表。
4) 内部类的构造器中是先保存外部类的引用到一个“合成变量”，再初始化外部类，最后才初始化自身。

以下面代码 java 代码为例：

```
public class HelloWorld {
    private String mHello = "Hello World!";
    public void sayHello() {
        System.out.println("Hello!");
    }
    private void say(String s) {
        System.out.println(s);
    }
    private class InterClass {
        public InterClass(int i) {}
        void func() {
            sayHello();
            say(mHello);
        }
    }
}
```

反编译后生成 HelloWorld.smali 和 HelloWorld$InterClass.smali 两个文件，其中关键代码如下：

HelloWorld 文件:

```
# 合成方法访问 mHello
.method static synthetic access$0(Lcom/smali/helloworld/HelloWorld;)Ljava/lang/String;
    .locals 1
    .parameter
```

```
    .prologue
    .line 7
    iget-object v0, p0, Lcom/smali/helloworld/HelloWorld;->mHello:Ljava/lang/String;
    return-object v0
.end method
#合成方法调用 say()
.method static synthetic access$1(Lcom/smali/helloworld/HelloWorld;Ljava/lang/String;)V
    .locals 0
    .parameter
    .parameter
    .prologue
    .line 22
    invoke-direct {p0, p1}, Lcom/smali/helloworld/HelloWorld;->say(Ljava/lang/String;)V
    return-void
.end method
```

HelloWorld$InterClass.smali 文件:

```
.method public constructor <init>(Lcom/smali/helloworld/HelloWorld;I)V    # 插入父类参数
    .locals 0
    .parameter
    .parameter "i"
    .prologue
    .line 27
    iput-object p1, p0, Lcom/smali/helloworld/HelloWorld$InterClass;
            ->this$0:Lcom/smali/helloworld/HelloWorld;    # 保存外部类的引用
    invoke-direct {p0}, Ljava/lang/Object;-><init>()V    #初始化父类
    return-void
.end method

.method func()V
    .locals 2
    .prologue
    .line 29
    iget-object v0, p0, Lcom/smali/helloworld/HelloWorld$InterClass;
            ->this$0:Lcom/smali/helloworld/HelloWorld;    # 引用外部类
    invoke-virtual {v0}, Lcom/smali/helloworld/HelloWorld;
            ->sayHello()V    #调用外部类公共方法
    .line 30
    iget-object v0, p0, Lcom/smali/helloworld/HelloWorld$InterClass;
            ->this$0:Lcom/smali/helloworld/HelloWorld;

    iget-object v1, p0, Lcom/smali/helloworld/HelloWorld$InterClass;
            ->this$0:Lcom/smali/helloworld/HelloWorld;
    #调用合成方法访问外部类私有变量 mHello
```

```
    invoke-static {v1}, Lcom/smali/helloworld/HelloWorld;
            ->access$0(Lcom/smali/helloworld/HelloWorld;)Ljava/lang/String;
    move-result-object v1
    #调用合成方法访问外部类私有方法 say()
    invoke-static {v0, v1}, Lcom/smali/helloworld/HelloWorld;
            ->access$1(Lcom/smali/helloworld/HelloWorld;Ljava/lang/String;)V
    .line 31
    return-void
.end method
```

---

4 如何分析和修改 Smali 代码

一个完整的的 Android 程序反编译后的代码量可能非常庞大，并且反编译后的 Smali 源码相比 java 的可读性差太多了，我们应该如何定位关键代码，分析并修改它们。

4.1 定位分析的方法

4.1.1 关键信息查找法

程序运行时会呈现给我们很多信息，如提示的文字内容、Log 输出的信息和 Activity TaskRecord 等信息，那么可以从这些信息入手来定位关键的代码。

比如，我们想查找程序显示 Toast 时上下文代码，Toast 提示的文字内容则会存放到 strings.xml 文件或硬编码到程序代码中，在资源文件中的字符串会有一个 id 索引，只需在反编译的代码中全文检索这个 id 即可找到显示该 Toast 的代码；如果是后者，则在反编译代码中查找这个字符串本身即可。

如在 Log 中分析到程序发出了一个广播，根据广播的 Action 字符串查找所有 smali 代码，也可定位到所有发出和接收该广播的多处代码位置，再逐个分析代码上下文不难定位时何处发出的广播。

对于涉及到程序 UI 逻辑的分析，通常借助 android-sdk 中的工具 hierarchyviewer 来快速分析定位是程序哪个 Activity 甚至是哪个 View 的相关代码。

4.1.2 代码动态调试法

遇到分析的代码逻辑较为复杂，程序运行与预期逻辑不符，通常使用动态调试法。可以把一段打印 Log 代码注入到反编译的代码中，重新编译打包后运行，查看注入位置执行情况和状态数据，方便可以定位分析代码的逻辑。但是注入 Log 代码可能需要多次，当分析大型程序时，就显得显得效率低下，这时可以采用注入栈跟踪信息来分析程序，具体实现即为构建一个 Exception 的对象，然后调用该对象的 printStackTrack 方法插入在需要调试的代码处，即可打印出函数调用间的层次关系，这一方法也可以用 DDMS 的 Method Profiling 工具进行跟踪。

当然也可以借助 eclipse 或 Netbeans 的强大的断点调试功能，查看反编译代码运行时的函数调用栈和变量信息，帮助快速理解代码逻辑和定位问题。

4.2 代码修改方法

Smali 代码相对复杂冗长，对于需要实现的功能，直接写 Smali 代码既费时间又容易出错，因此通常采用另一种做法，就是先把功能用 Java 源码的方式实现，然后反编译得到 Smali

代码，再把 Smali 代码合并到目标代码中。

以上面提到的注入 Log 输出的代码为例，需要可以输出变量信息和打印当前调用栈的功能，那么先在一个简单 APK Demo 中完成以下的 java 代码。

```
    public void myLogCaller() {
        myLog("Here is a smali log.");
    }

    public void myLog(Object o) {
        Log.d("SmaliLog", o.toString());
        StackTraceElement st[] = Thread.currentThread().getStackTrace();
        for (int i = 0; i < st.length; i++) {
            Log.d("SmaliLog", "    at " + st[i].toString());
        }
    }
```

反编译这个 Demo，得到该 java 代码的 Smali 源码：

```
.method myLogCaller()V
    .locals 1

    .prologue
    .line 38
    const-string v0, "Here is a smali log."
    invoke-static {v0}, Lcom/smali/helloworld/MainActivity;->myLog(Ljava/lang/Object;)V

    .line 39
    return-void
.end method

.method private static myLog(Ljava/lang/Object;)V
    .locals 5
    .parameter "o"

    .prologue
    .line 42
    const-string v2, "SmaliLog"
    invoke-virtual {p0}, Ljava/lang/Object;->toString()Ljava/lang/String;
    move-result-object v3
    invoke-static {v2, v3}, Landroid/util/Log;->d(Ljava/lang/String;Ljava/lang/String;)I

    .line 43
    invoke-static {}, Ljava/lang/Thread;->currentThread()Ljava/lang/Thread;
```

```
    move-result-object v2
    invoke-virtual {v2}, Ljava/lang/Thread;->getStackTrace()[Ljava/lang/StackTraceElement;
    move-result-object v1
    ……
    invoke-static {v2, v3}, Landroid/util/Log;->d(Ljava/lang/String;Ljava/lang/String;)I

    .line 44
    add-int/lit8 v0, v0, 0x1

    goto :goto_0
.end method
```

具体打印日志和调用栈功能实现已经在 mylog()方法实现，只需把整个方法的 Smali 代码都添加到目标代码的文件中，关键是调用 mylog()方法的 Smali 语句注入到需要打印的位置，注入语句如：

```
    #vA 为需要打印的变量
    invoke-static {vA}, Lcom/smali/helloworld/MainActivity;->myLog(Ljava/lang/Object;)V
```

---

5 参考资料：

[1].Smali.Anassembler/disassemblerforAndroid'sdexformat.
https://code.google.com/p/smali/
[2].Android-apktool.AtoolforreverseengineeringAndroidapkfiles.
https://code.google.com/p/android-apktool/
[3].DalvikVMInstructionFormats.
http://source.android.com/tech/dalvik/instruction-formats.html
[4].SmaliTypesMethodsAndFields.
https://code.google.com/p/smali/wiki/TypesMethodsAndFields
[5].SmaliTypesMethodsAndFields.
http://source.android.com/tech/dalvik/dalvik-bytecode.html
[6]. 丰生强.Android 软件安全与逆向分析.2013-02-01