

第3章 进入Android Dalvik虚拟机

虽然Android平台使用Java语言来开发应用程序，但Android程序却不是运行在标准Java虚拟机上的。可能是为了解决移动设备上软件运行效率的问题，也可能是为了规避与Oracle公司的版权纠纷。Google为Android平台专门设计了一套虚拟机来运行Android程序，它就是Dalvik Virtual Machine（Dalvik虚拟机）。本章将讨论Dalvik虚拟机的特性及基于Dalvik字节码的汇编语言知识。

3.1 Dalvik虚拟机的特点——掌握Android程序的运行原理

本节主要介绍Dalvik的基本特性以及工作原理，这对掌握Android程序运行原理尤为重要。

3.1.1 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人们的视野。它的作者是丹·伯恩斯坦（Dan Bernstein），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：

- 体积小，占用内存空间小；
- 专有的DEX可执行文件格式，体积更小，执行速度更快；
- 常量池采用32位索引值，寻址类方法名、字段名、常量更快；
- 基于寄存器架构，并拥有一套完整的指令系统；
- 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；
- 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。

3.1.2 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。

传统的Java程序经过编译，生成Java字节码保存在class文件中，Java虚拟机通过解码class文件中的内容来运行程序。而Dalvik虚拟机运行的是Dalvik字节码，所有的Dalvik字

节码由 Java 字节码转换而来，并被打包到一个 DEX（Dalvik Executable）可执行文件中。Dalvik 虚拟机通过解释 DEX 文件来执行这些字节码。

2. Dalvik 可执行文件体积更小。

Android SDK 中有一个叫 dx 的工具负责将 Java 字节码转换为 Dalvik 字节码。dx 工具对 Java 类文件重新排列，消除在类文件中出现的所有冗余信息，避免虚拟机在初始化时出现反复的文件加载与解析过程。一般情况下，Java 类文件中包含多个不同的方法签名，如果其他的类文件引用该类文件中的方法，方法签名也会被复制到其类文件中，也就是说，多个不同的类会同时包含相同的方法签名，同样地，大量的字符串常量在多个类文件中也被重复使用。这些冗余信息会直接增加文件的体积，同时也会严重影响虚拟机解析文件的效率。dx 工具针对这个问题专门做了处理，它将所有的 Java 类文件中的常量池进行分解，消除其中的冗余信息，重新组合形成一个常量池，所有的类文件共享同一个常量池。dx 工具的转换过程如图 3-1 所示。由于 dx 工具对常量池的压缩，使得相同的字符串、常量在 DEX 文件中只出现一次，从而减小了文件的体积。

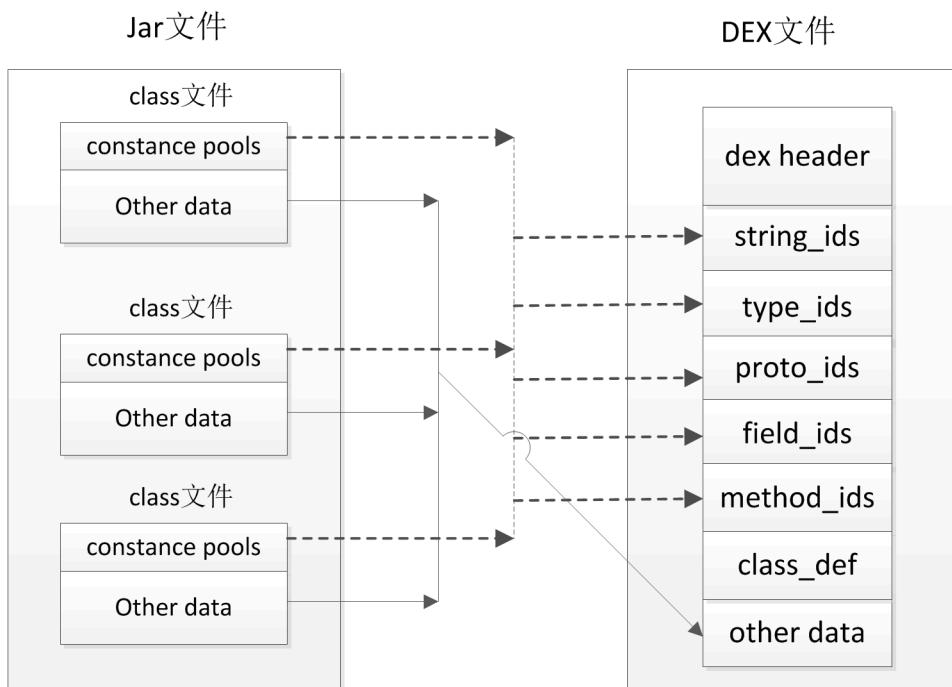


图3-1 Java文件转换为DEX文件

3. Java 虚拟机与 Dalvik 虚拟机架构不同。

Java 虚拟机基于栈架构。程序在运行时虚拟机需要频繁的从栈上读取或写入数据，这个

过程需要更多的指令分派与内存访问次数，会耗费不少CPU时间，对于像手机设备资源有限的设备来说，这是相当大的一笔开销。

Dalvik虚拟机基于寄存器架构。数据的访问通过寄存器间直接传递，这样的访问方式比基于栈方式要快很多。下面通过一个实例来对比一下Java字节码与Dalvik字节码的区别。测试代码如下。

```
public class Hello {  
    public int foo(int a, int b) {  
        return (a + b) * (a - b);  
    }  
  
    public static void main(String[] argc) {  
        Hello hello = new Hello();  
        System.out.println(hello.foo(5, 3));  
    }  
}
```

将以上内容保存为Hello.java。打开命令提示符，执行命令“javac Hello.java”编译生成Hello.class文件。然后执行命令“dx --dex --output=Hello.dex Hello.class”生成dex文件。

接下来使用javap反编译Hello.class查看foo()函数的Java字节码，执行以下命令。

```
javap -c -classpath . Hello
```

命令执行后得到如下代码：

```
public int foo(int, int);  
Code:  
 0:  iload_1  
 1:  iload_2  
 2:  iadd  
 3:  iload_1  
 4:  iload_2  
 5:  isub  
 6:  imul  
 7:  ireturn
```

使用dexdump.exe(位于Android SDK的platform-tools目录中)查看foo()函数的Dalvik字节码，执行以下命令。

```
dexdump.exe -d Hello.dex
```

命令执行后整理输出结果，可以得到如下代码。

```
Hello.foo: (II) I
```

```

0000: add-int v0, v3, v4
0002: sub-int v1, v3, v4
0004: mul-int/2addr v0, v1
0005: return v0

```

注意 如果使用 JDK1.7 编译 Hello.java，生成的 Hello.class 默认的版本会比较低，使用 dx 生成 dex 文件会提示 class 文件无效。解决方法是强制指定 class 文件的版本，执行如下命令重新编译。

```
javac -source 1.6 -target 1.6 Hello.java
```

使用 dx 工具编译 Hello.class 时，如果提示无法找到 Hello.class 文件，可以将 Hello.class 文件与 dx 放同一目录后重新编译。

查看上面的 Java 字节码，发现 foo() 函数一共占用了 8 个字节，代码中每条指令占用 1 个字节，并且这些指令都没有参数。那么这些指令是如何存取数据的呢？Java 虚拟机的指令集被称为零地址形式的指令集，所谓零地址形式，是指指令的源参数与目标参数都是隐含的，它通过 Java 虚拟机中提供的一种数据结构“求值栈”来传递的。

对于 Java 程序来说，每个线程在执行时都有一个 PC 计数器与一个 Java 栈。PC 计数器以字节为单位记录当前运行位置距离方法开头的偏移量，它的作用类似于 ARM 架构 CPU 的 PC 寄存器与 x86 架构 CPU 的 IP 寄存器，不同的是 PC 计数器只对当前方法有效，Java 虚拟机通过它的值来取指令执行。Java 栈用于记录 Java 方法调用的“活动记录”（activation record），Java 栈以帧（frame）为单位保存线程的运行状态，每调用一个方法就会分配一个新的栈帧压入 Java 栈上，每从一个方法返回则弹出并撤销相应的栈帧。每个栈帧包括局部变量区、求值栈（JVM 规范中将其称为“操作数栈”）和其他一些信息。局部变量区用于存储方法的参数与局部变量，其中参数按源码中从左到右顺序保存在局部变量区开头的几个 slot 中。求值栈用于保存求值的中间结果和调用别的方法的参数等，JVM 运行时它的状态结构如图 3-2 所示。

结合代码来理解上面的理论知识。由于每条指令占用一个字节空间，foo() 函数 Java 字节码左边的偏移量就是程序执行到每一行代码时 PC 的值，并且 Java 虚拟机最多只支持 0xff 条指令。第 1 条指令 iload_1 可分成两部分：第一部分为下划线左边的 iload，它属于 JVM（Java 虚拟机）指令集中 load 系列中的一条，i 是指令前缀，表示操作类型为 int 类型，load 表示将局部变量存入 Java 栈，与之类似的有 lload、fload、dload 分别表示将 long、float、double 类型的数据进栈；第二部分为下划线右边的数字，表示要操作具体哪个局部变量，索引值从 0 开始计数，iload_1 表示将第二个 int 类型的局部变量进栈，这里第二个局部变量是存放在局部变量区 foo() 函数的第二参数。同理，第 2 条指令 iload_2 取第二个参数。第 3 条指令 iadd 从栈顶弹出两个 int 类型值，将值相加，然后把结果压回栈顶。第 4、5 条指令分别再次

压入第一个参数与第二个参数。第6条指令 isub 从栈顶弹出两个 int 类型值，将值相减，然后把结果压回栈顶。这时求值栈上有两个 int 值了。第7条指令 imul 从栈顶弹出两个 int 类型值，将值相乘，然后把结果压回栈顶。第8条指令 ireturn 函数返回一个 int 值。到这里，foo() 函数就执行完了。关于 Java 虚拟机字节码的其它内容，笔者在此就不展开了，读者可以在以下网址找到一份完整的 Java 字节码指令列表：http://en.wikipedia.org/wiki/Java_bytecode_instruction_listings。

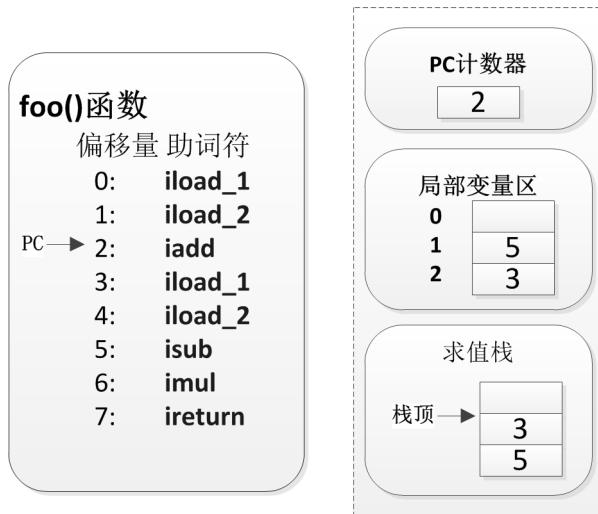


图3-2 JVM运行状态

比起 Java 虚拟机字节码，上面的 Dalvik 字节码显得简洁很多，只有 4 条指令就完成了上面的操作。第一条指令 add-int 将 v3 与 v4 寄存器的值相加，然后保存到 v0 寄存器，整个指令的操作中使用到了三个参数，v3 与 v4 分别代表 foo() 函数的第一个参数与第二个参数，它们是 Dalvik 字节码参数表示法之一 v 命名法，另一种是 p 命名法，本章 3.2 小节会详细介绍 Dalvik 汇编语言。第二条指令 sub-int 将 v3 减去 v4 的值保存到 v1 寄存器。第三条指令 mul-int/2addr 将 v0 乘以 v1 的值保存到 v0 寄存器。第四条指令返回 v0 的值。

Dalvik 虚拟机运行时同样为每个线程维护一个 PC 计数器与调用栈，与 Java 虚拟机不同的是，这个调用栈维护一份寄存器列表，寄存器的数量在方法结构体的 registers 字段中给出，Dalvik 虚拟机会根据这个值来创建一份虚拟的寄存器列表。Dalvik 虚拟机运行时的状态如图 3-3 所示。

通过上面的分析可以发现，基于寄存器架构的 Dalvik 虚拟机与基于栈架构的 Java 虚拟机相比，由于生成的代码指令减少了，程序执行速度会更快一些。

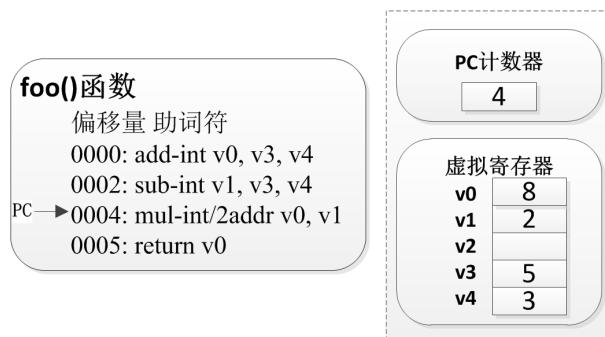


图3-3 Dalvik VM运行状态

3.1.3 Dalvik 虚拟机是如何执行程序的

Android 系统的架构采用分层思想，这样的好处是拥有减少各层之间的依赖性、便于独立分发、容易收敛问题和错误等优点。Android 系统由 Linux 内核、函数库、Android 运行时、应用程序框架以及应用程序组成。如图 3-4 的 Android 系统架构所示，Dalvik 虚拟机属于 Android 运行时环境，它与一些核心库共同承担 Android 应用程序的运行工作。

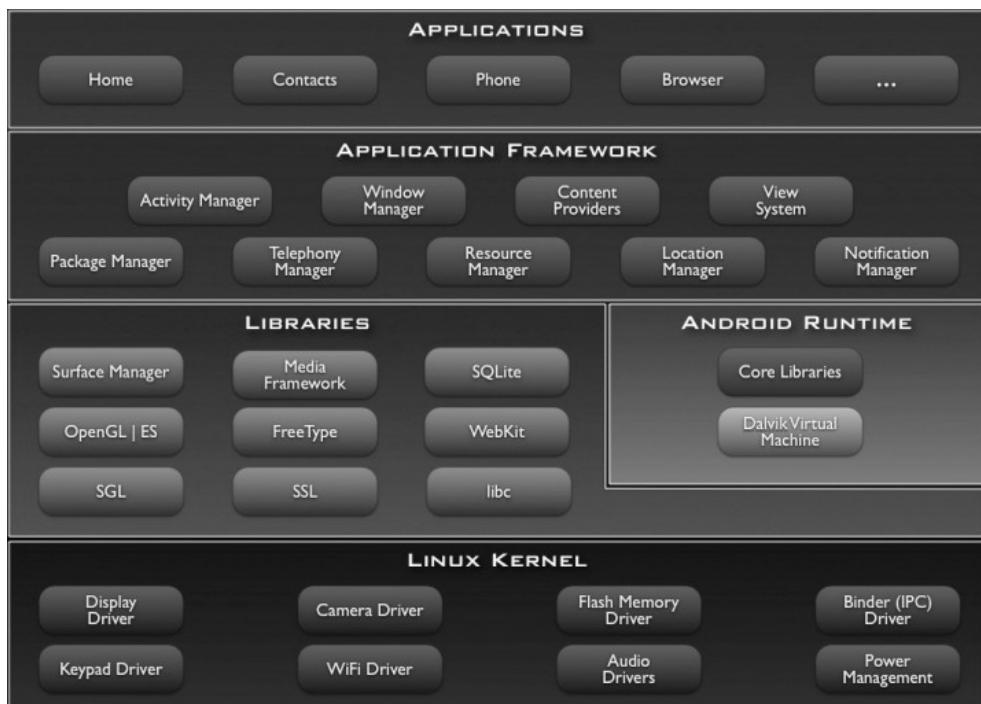


图3-4 Android系统结构

Android系统启动加载完内核后，第一个执行的是init进程，init进程首先要做的是设备的初始化工作，然后读取inic.rc文件并启动系统中的重要外部程序Zygote。Zygote进程是Android所有进程的孵化器进程，它启动后会首先初始化Dalvik虚拟机，然后启动system_server并进入Zygote模式，通过socket等候命令。当执行一个Android应用程序时，system_server进程通过Binder IPC方式发送命令给Zygote，Zygote收到命令后通过fork自身创建一个Dalvik虚拟机的实例来执行应用程序的入口函数，这样一个程序就启动完成了。整个流程如图3-5所示。

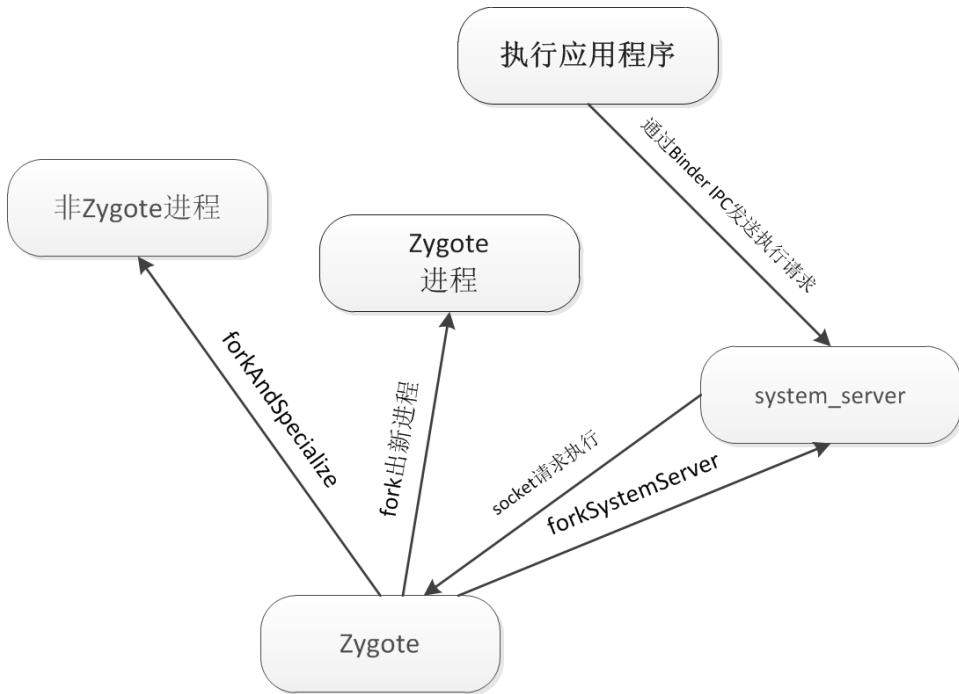


图3-5 Zygote启动进程

Zygote提供了三种创建进程的方法：

- fork()，创建一个Zygote进程（这种方式实际不会被调用）；
- forkAndSpecialize()，创建一个非Zygote进程；
- forkSystemServer()，创建一个系统服务进程。

其中，Zygote进程可以再fork()出其他进程，非Zygote进程则不能fork其他进程，而系统服务进程在终止后它的子进程也必须终止。

当进程fork成功后，执行的工作就交给了Dalvik虚拟机。Dalvik虚拟机首先通过loadClassFromDex()函数完成类的装载工作，每个类被成功解析后都会拥有一个ClassObject类型的数据结构存储在运行时环境中，虚拟机使用gDvm.loadedClasses全局哈希表来存储与

查询所有装载进来的类，随后，字节码验证器使用 `dvmVerifyCodeFlow()` 函数对装入的代码进行校验，接着虚拟机调用 `FindClass()` 函数查找并装载 `main` 方法类，随后调用 `dvmInterpret()` 函数初始化解释器并执行字节码流。整个过程如图 3-6 所示。

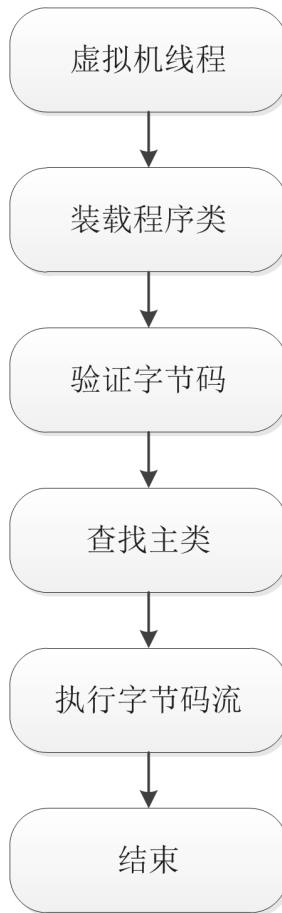


图3-6 Dalvik虚拟机执行程序流程

3.1.4 关于 Dalvik 虚拟机 JIT（即时编译）

JIT (Just-in-time Compilation, 即时编译)，又称为动态编译，是一种通过在运行时将字节码翻译为机器码的技术，使得程序的执行速度更快。Android 2.2 版本系统的 Dalvik 虚拟机引入了 JIT 技术，官方宣称新版的 Dalvik 虚拟机比以往执行速度快 3~6 倍。

主流的 JIT 包含两种字节码编译方式：

- method 方式：以函数或方法为单位进行编译。
- trace 方式：以 trace 为单位进行编译。

method 方式很好理解，那什么是 trace 方式呢？在函数中一般很少是顺序执行代码的，多数的代码都分成了好几条执行路径，其中函数的有些路径在实际运行过程中是很少被执行的，这部分路径被称为“冷路径”，而执行比较频繁的路径被称为“热路径”。采用传统的 method 方式会编译整个方法的代码，这会使得在“冷路径”上浪费很多编译时间，并且耗费更多的内存；trace 方式编译则能够快速地获取“热路径”代码，使用更短的时间与更少的内存来编译代码。

目前，Dalvik 虚拟机默认采用 trace 方式编译代码，同时也支持采用 method 方式来编译。关于 JIT 的详细内容本书不做深入的探讨，有兴趣的读者可以参看其它相关资料。

3.2 Dalvik 汇编语言基础为分析 Android 程序做准备

Dalvik 虚拟机为自己专门设计了一套指令集，并且制定了自己的指令格式与调用规范。我们将 Dalvik 指令集组成的代码称为 Dalvik 汇编代码，将这种代码表示的语言称为 Dalvik 汇编语言（Dalvik 汇编语言并不是正式的语言，只是本书描述 Dalvik 指令集代码的一种称呼）。本节主要介绍 Dalvik 汇编语言的相关基础知识。

3.2.1 Dalvik 指令格式

一段 Dalvik 汇编代码由一系列 Dalvik 指令组成，指令语法由指令的位描述与指令格式标识来决定。位描述约定如下：

- 每 16 位的字采用空格分隔开来。
- 每个字母表示四位，每个字母按顺序从高字节开始，排列到低字节。每四位之间可能使用竖线“|”来表示不同的内容。
- 顺序采用 A~Z 的单个大写字母作为一个 4 位的操作码，op 表示一个 8 位的操作码。
- “Ø”来表示这字段所有位为 0 值。

以指令格式“A|G|op BBBB F|E|D|C”为例：

指令中间有两个空格，每个分开的部分大小为 16 位，所以这条指令由三个 16 位的字组成。第一个 16 位是“A|G|op”，高 8 位由 A 与 G 组成，低字节由操作码 op 组成。第二个 16 位由 BBBB 组成，它表示一个 16 位的偏移值。第三个 16 位分别由 F、E、D、C 共四个 4 字节组成，在这里它们表示寄存器参数。

单独使用位标识还无法确定一条指令，必须通过指令格式标识来指定指令的格式编码。它的约定如下：

- 指令格式标识大多由三个字符组成，前两个是数字，最后一个字母。
- 第一个数字是表示指令有多少个 16 位的字组成。
- 第二个数字是表示指令最多使用寄存器的个数。特殊标记“r”标识使用一定范围内的寄存器。

- 第三个字母为类型码，表示指令用到的额外数据的类型。取值见表 3-1。

还有一种特殊的情况是末尾可能会多出另一个字母，如果是字母 s 表示指令采用静态链接，如果是字母 i 表示指令应该被内联处理。

表 3-1 指令格式标识的类型码

助记符	位大小	说 明
b	8	8 位有符号立即数
c	16, 32	常量池索引
f	16	接口常量（仅对静态链接格式有效）
h	16	有符号立即数（32 位或 64 位数的高值位，低值位为 0）
i	32	立即数，有符号整数或 32 位浮点数
l	64	立即数，有符号整数或 64 位双精度浮点数
m	16	方法常量（仅对静态链接格式有效）
n	4	4 位的立即数
s	16	短整型立即数
t	8, 16, 32	跳转、分支
x	0	无额外数据

以指令格式标识 22x 为例：

第一个数字 2 表示指令有两个 16 位字组成，第二个数字 2 表示指令使用到 2 个寄存器，第三个字母 x 表示没有使用到额外的数据。

另外，Dalvik 指令对语法做了一些说明，它约定如下：

- 每条指令从操作码开始，后面紧跟参数，参数个数不定，每个参数之间采用逗号分开。
- 每条指令的参数从指令第一部分开始，op 位于低 8 位，高 8 位可以是一个 8 位的参数，也可以是两个 4 位的参数，还可以为空，如果指令超过 16 位，则后面部分依次作为参数。
- 如果参数采用“vX”的方式表示，表明它是一个寄存器，如 v0、v1 等。这里采用 v 而不用 r 是为了避免与基于该虚拟机架构本身的寄存器命名产生冲突，如 ARM 架构寄存器命名采用 r 开头。
- 如果参数采用“#+X”的方式表示，表明它是一个常量数字。
- 如果参数采用“+X”的方式表示，表明它是一个相对指令的地址偏移。
- 如果参数采用“kind@X”的方式表示，表明它是一个常量池索引值。其中 kind 表示常量池类型，它可以是“string”（字符串常量池索引）、“type”（类型常量池索引）、“field”（字段常量池索引）或者“meth”（方法常量池索引）。

以指令“op vAA, string@BBBB”为例：

指令用到了1个寄存器参数vAA，并且还附加了一个字符串常量池索引string@BBBB，其实这条指令格式代表着const-string指令。

在Android4.0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指令的所有格式。读者可以通过它了解Dalvik指令更加完整的信息。

注意 在Android4.1源码Dalvik/docs目录中，instruction-formats.html已经被移除了。

3.2.2 DEX文件反汇编工具

目前DEX可执行文件主流的反汇编工具有BakSmali与Dedexer。两者的反汇编效果都不错，在语法上也有着很多的相似处。下面通过代码对比两者的语法差异，测试代码采用上一节的Hello.java，首先使用dx工具生成Hello.dex文件，然后在命令提示符下输入以下命令使用baksimali.jar反汇编Hello.dex：

```
java -jar baksimali.jar -o baksimaliout Hello.dex
```

命令成功执行会在baksimaliout目录下生成Hello.smali文件，使用文本编辑器打开它，foo()函数代码如下。

```
# virtual methods
.method public foo(I)I
    .registers 5
    .parameter
    .parameter
    .prologue
    .line 3
    add-int v0, p1, p2
    sub-int v1, p1, p2
    mul-int/2addr v0, v1
    return v0
.end method
```

执行以下命令使用ddx.jar(Dedexer的jar文件)反汇编Hello.dex：

```
java -jar ddx.jar -d ddxout Hello.dex
```

命令成功执行后，会在ddxout目录下生成Hello.ddx文件，使用文本编辑器打开它，foo()函数代码如下。

```
.method public foo(I)I
    .limit registers 5
```

```

; this: v2 (LHello;)
; parameter[0] : v3 (I)
; parameter[1] : v4 (I)
.line 3
    add-int v0,v3,v4
    sub-int v1,v3,v4
    mul-int/2addr v0,v1
    return v0
.end method

```

两种反汇编代码大体的结构组织是一样的，在方法名、字段类型与代码指令序列上它们保持一致，具体的差异表现在一些语法细节上。对比之下，可以发现如下不同点：

- 前者使用`.registers` 指令指定函数用到的寄存器数目，后者在`.registers` 指令前加了`limit` 前缀。
- 前者使用寄存器`p0` 作为`this` 引用，后者使用寄存器`v2` 作为`this` 引用。
- 前者使用一条`.parameter` 指令指定函数一个参数，后者则使用`parameter` 数组指定参数寄存器。
- 前者使用`.prologue` 指令指定函数代码起始处，后者却没有。
- 两者寄存器表示法不同，前者使用`p` 命名法，后者使用`v` 命名法。（具体差异下一节进行讲解）

BakSmali 提供反汇编功能的同时，还支持使用 Smali 工具打包反汇编代码重新生成 dex 文件，这个功能被广泛应用于 apk 文件的修改、补丁、破解等场合，因而更加受到开发人员的青睐，本书 Dalvik 指令的语法将采用 Smali 语法格式。

3.2.3 了解 Dalvik 寄存器

Dalvik 虚拟机基于寄存器架构，在代码中大量地使用到了寄存器。Dalvik 虚拟机是作用于特定架构的 CPU 上运行的，在设计之初采用了 ARM 架构，ARM 架构的 CPU 本身集成了多个寄存器，Dalvik 将部分寄存器映射到了 ARM 寄存器上，还有一部分则通过调用栈进行模拟。注意：Dalvik 中用到的寄存器都是 32 位的，支持任何类型，64 位类型用 2 个相邻寄存器表示。

Dalvik 虚拟机支持多少个虚拟寄存器呢？通过查看 Dalvik 指令格式表，可以发现类似“`OP|op AAAA BBBB`”的指令，它的语法为“`op vAAAA, vBBBB`”，其中每个大写字母代表 4 位，AAAA、BBBB 最大值是 2 的 16 次方即 65536，寄存器采用`v0` 作起始值，因此，它的取值范围是`v0~v65535`。

Dalvik 虚拟机又是如何虚拟地使用寄存器的呢？这个还得从上面章节讲到的 Dalvik 虚拟机的调用栈说起，Dalvik 虚拟机为每个进程维护一个调用栈，这个调用栈其中一个作用就是用来“虚拟”寄存器，由上一节我们知道，每个函数都在函数头部使用`.registers` 指令指定

函数用到的寄存器数目，当虚拟机执行到这个函数时，会根据寄存器的数目分配适当的栈空间，这些空间就是用来存放寄存器实际的值的！虚拟机通过处理字节码，对寄存器进行读与写的操作，其实都是在写栈空间。Android SDK 中有一个名为 `dalvik bytecode.Opcodes` 的接口，它定义了一份完整的 Dalvik 字节码列表。处理这些字节码的函数为一个宏 `HANDLE_OPCODE()`，这份 Dalvik 字节码列表中每个字节码的处理过程可以在 Android 源码的 `dalvik\vm\mterp\c` 目录中找到，拿 `OP_MOVE` 来举例，`OP_MOVE.cpp` 内容如下：

```
HANDLE_OPCODE($opcode /*vA, vB*/)
    vdst = INST_A(inst);
    vsrc1 = INST_B(inst);
    ILOGV(" | move%s v%d, v%d %s(v%d=0x%08x) ",
        (INST_INST(inst) == OP_MOVE) ? "" : "-object", vdst, vsrc1,
        kSpacing, vdst, GET_REGISTER(vsrc1));
    SET_REGISTER(vdst, GET_REGISTER(vsrc1));
    FINISH(1);
OP_END
```

`INST_A` 是用来获取 `vA` 寄存器地址的宏，右边的 `A` 表示寄存器的“名称”，可以是其他的字母或长度，如 `INST_AA`、`INST_B` 等分别是获取 `vAA` 与 `vB` 的地址。在 `OP_MOVE.cpp` 文件同目录的 `header.cpp` 文件中，`INST_A` 与 `INST_B` 的声明如下：

```
#define INST_A(_inst)      (((_inst) >> 8) & 0x0f)
#define INST_B(_inst)      (((_inst) >> 12))
```

这里的 `_inst` 为一个 16 位的指令，`INST_A` 将 `_inst` 右移 8 位然后与 `0x0f` 相与，也就是获取了 `_inst` 高 8 位的低 4 位作为 `vdst` 的值，而 `INST_B` 将 `_inst` 右移 12 位，也就是获取 `_inst` 的最高 4 位作为 `vsrc1` 的值。

`ILOGV` 用来输出调试信息。

`SET_REGISTER` 用来设置寄存器的值，`GET_REGISTER` 用来获取寄存器的值。另外，操作的寄存器可以是其它的大小与类型，如 `WIDE`、`FLOAT`，相关的宏函数则是 `GET_REGISTER_WIDE`、`GET_REGISTER_FLOAT`。在 `header.cpp` 文件中，`GET_REGISTER` 与 `SET_REGISTER` 的声明如下：

```
# define GET_REGISTER(_idx)    (fp[(_idx)])
# define SET_REGISTER(_idx, _val)  (fp[(_idx)] = (_val))
```

`fp` 为 ARM 栈帧寄存器，在虚拟机运行到某个函数时它指向函数的局部变量区，其中就维护着一份寄存器值的列表，`GET_REGISTER` 宏以 `_idx` 为索引返回一个“寄存器”的值，而 `SET_REGISTER` 则是以 `_idx` 为索引，设置相应寄存器的值。如果 Dalvik 虚拟机开启了寄存器数目验证，即`#ifdef CHECK_REGISTER_INDICES` 为真时，在进行寄存器读写操作时，虚拟机会首先判断 `_idx` 是否小于 `curMethod->registersSize`，如果条件不成立则说明寄存器超

出引用范围，此时虚拟机会通过 `assert(!"bad reg")` 抛出异常。

最后，由 FINISH 宏来完成一条指令的执行。FINISH 的功能由 ADJUST_PC 宏来完成，主要是计算当前指令占用的长度，然后将 PC 寄存器加上计算出的偏移，这样一条指令执行完成后，PC 计数器会指向下一条将要执行的指令。

3.2.4 两种不同的寄存器表示方法——v 命名法与 p 命名法

前面曾多次提到 v 命名法与 p 命名法，它们是 Dalvik 字节码中两种不同的寄存器表示方法。下面我们来看看，它们在表现上有一些什么样的区别。

假设一个函数使用到 M 个寄存器，并且该函数有 N 个参数，根据 Dalvik 虚拟机参数传递方式中的规定：参数使用最后的 N 个寄存器中，局部变量使用从 v0 开始的前 M-N 个寄存器。如前面的小节中，`foo()` 函数使用到了 5 个寄存器，2 个显式的整形参数，其中 `foo()` 函数是 Hello 类的非静态方法，函数被调用时会传入一个隐式的 Hello 对象引用，因此，实际传入的参数数量是 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 命名法采用以小写字母“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 `foo()` 函数，v 命名法会用到 v0、v1、v2、v3、v4 等五个寄存器，v0 与 v1 用来表示函数的局部变量寄存器，v2 表示被传入的 Hello 对象的引用，v3 与 v4 分别表示两个传入的整形参数。

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：函数中引入的参数命名从 p0 开始，依次递增。对于 `foo()` 函数，p 命名法会用到 v0、v1、p0、p1、p2 等五个寄存器，v0 与 v1 同样用来表示函数的局部变量寄存器，p0 表示被传入的 Hello 对象的引用，p1 与 p2 分别表示两个传入的整形参数。

对于有 M 个寄存器及 N 个参数的函数 `foo()` 来说，v 命名法与 p 命名法的表现形式如表 3-2 所示。通过观察可以发现，使用 p 命名法表示的 Dalvik 汇编代码，通过寄存器的前缀更容易判断寄存器到底是局部变量寄存器还是参数寄存器，在 Dalvik 汇编代码较长、使用寄存器较多的情况下，这种优势将更加明显。

表 3-2 v 命名法与 p 命名法

v 命名法	p 命名法	寄存器含义
v0	v0	第一个局部变量寄存器
v1	v1	第二个局部变量寄存器
...	...	中间的局部变量寄存器依次递增且名称相同
vM-N	p0	第一个参数寄存器
...	...	中间的参数寄存器分别依次递增
vM-1	pN-1	第 N 个参数寄存器

3.2.5 Dalvik字节码的类型、方法与字段表示方法

Dalvik字节码有着一套自己的类型、方法与字段表示方法，这些方法与Dalvik虚拟机指令集一起组成了一条条的Dalvik汇编代码。

1. 类型

Dalvik字节码只有两种类型，基本类型与引用类型。Dalvik使用这两种类型来表示Java语言的全部类型，除了对象与数组属于引用对象外，其他的Java类型都是基本类型。BakSmali严格遵守了DEX文件格式中的类型描述符（DEX文件格式将在第四章进行介绍）定义。类型描述符对照如表3-3所示。

表3-3 Dalvik字节码类型描述符

语 法	含 义
V	void，只用于返回值类型
Z	boolean
B	byte
S	short
C	char
I	int
J	long
F	float
D	double
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。

L类型可以表示Java类型中的任何类。这些类在Java代码中以package.name.ObjectName方式引用，到了Dalvik汇编代码中，它们以Lpackage/name/ObjectName;形式表示，注意最后有个分号，L表示后面跟着一个Java类，package/name/表示对象所在的包，ObjectName表示对象的名称，最后的分号表示对象名结束。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。[后面紧跟基本类型描述符，如[I表示一个整型一维数组，相当于Java中的int[]]。多个[在一起时可用来表示多维数组，如[[I表示int[][]]，[[[I表示int[][][]]。注意多维数组的维数最大为255个。

L 与 [可以同时使用来表示对象数组。如 [Ljava/lang/String; 就表示 Java 中的字符串数组。

2. 方法

方法的表现形式比类名要复杂一些，Dalvik 使用方法名、类型参数与返回值来详细描述一个方法。这样做一方面有助于 Dalvik 虚拟机在运行时从方法表中快速地找到正确的方法，另一方面，Dalvik 虚拟机也可以使用它们来做一些静态分析，比如 Dalvik 字节码的验证与优化。

方法格式如下：

Lpackage/name/ObjectName;->MethodName(III)Z

在这个例子中，*Lpackage/name/ObjectName;-* 应该理解为一个类型，*MethodName* 为具体的方法名，*(III)Z* 是方法的签名部分，其中括号内的 III 为方法的参数（在此为三个整型参数），Z 表示方法的返回类型（boolean 类型）。

下面是一个更为复杂的例子：

```
method([[IILjava/lang/String;[Ljava/lang/Object;)Ljava/lang/String;
```

按照上面的知识，将其转换成 Java 形式的代码应该为：

```
String method(int, int[], int, String, Object[])
```

BakSmali 生成的方法代码以.method 指令开始，以.end method 指令结束，根据方法类型的不同，在方法指令开始前可能会用井号 “#” 加以注释。如 “# virtual methods” 表示这是一个虚方法，“# direct methods” 表示这是一个直接方法。

3. 字段

字段与方法很相似，只是字段没有方法签名域中的参数与返回值，取而代之的是字段的类型。同样，Dalvik 虚拟机定位字段与字节码静态分析时会用到它。字段的格式如下：

Lpackage/name/ObjectName;->FieldName:Ljava/lang/String;

字段由类型（*Lpackage/name/ObjectName;*）、字段名（*FieldName*）与字段类型（*Ljava/lang/String;*）组成。其中字段名与字段类型中间用冒号 “:” 隔开。

BakSmali 生成的字段代码以.field 指令开头，根据字段类型的不同，在字段指令的开始可能会用井号 “#” 加以注释，如 “# instance fields” 表示这是一个实例字段，“# static fields” 表示这是一个静态字段。

3.3 Dalvik 指令集

在 Android 4.0 源码 Dalvik/docs 目录下提供了一份指令集文档 [dalvik-bytecode.html](#)，里面详细列举了 Dalvik 支持的所有指令，不过该文档在 Android 4.1 源码中已经去除。