

周爱民
@aimingoo
aiming@gmail.com
blog.csdn.net/aimingoo
...

smali汇编语言的设计与实现

概要

- 1、一些简单的背景
- 2、一些语言基础
- 3、对象与数组的实现
- 4、基本控制结构的实现
- 5、指令与编码的一些细节

HTC SENSE的经典界面



全屏来电大头贴！原生的！



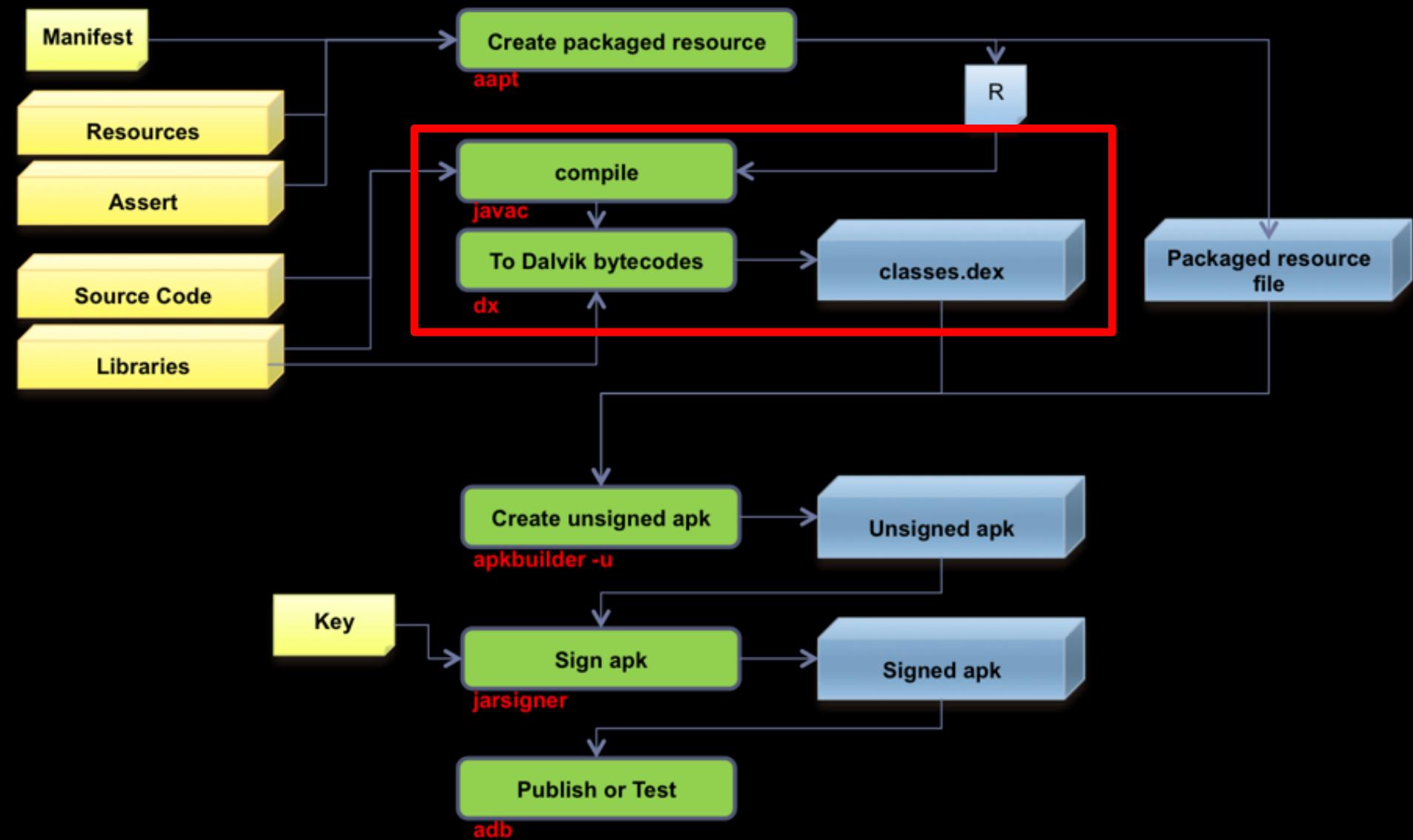
全屏来电大头贴！原生的！



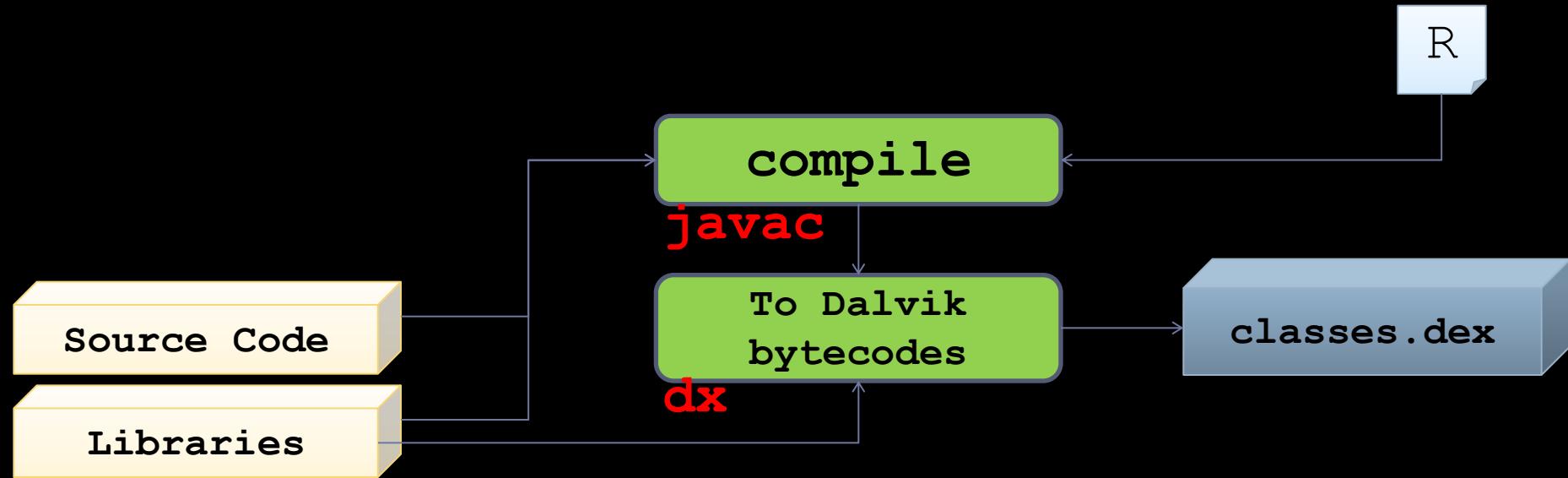
要怎么做？

- 1、解开.apk
- 2、解出资源文件与classes.dex
- 3、从classes.dex反编译得到.smali
- 4、修改.smali，并编译回classes.dex
- 5、替换.apk中的classes.dex
- 6、重新签名
- 7、OK

关于一个.apk



classes.dex的秘密



最基本的工具: dexdump

```
> %ANDROID_TOOLS%\dexdump -d classes.dex
```

```
036794:           | [036794] ....equals:(Ljava/lang/Object;)Z  
0367a4: 1215    | 0000: const/4 v5, #int 1 // #1  
0367a6: 1204    | 0001: const/4 v4, #int 0 // #0  
0367a8: 3376 0400 | 0002: if-ne v6, v7, 0006 // +0004  
0367ac: 0152    | 0004: move v2, v5  
0367ae: 0f02    | 0005: return v2  
0367b0: 3907 0400 | 0006: if-nez v7, 000a // +0004  
0367b4: 0142    | 0008: move v2, v4  
0367b6: 28fc    | 0009: goto 0005 // -0004  
0367b8: 6e10 5113 0600 | 000a: invoke-virtual {v6}, ...; // method@1351  
0367be: 0c02    | 000d: move-result-object v2  
0367c0: 6e10 5113 0700 | 000e: invoke-virtual {v7}, ...; // method@1351
```

```
> %ANDROID_TOOLS%\dexdump -d classes.dex > classes.dump
```

```
> dir classes.dex
```

2012/08/14 03:48	695,260 classes.dex
2013/10/25 01:47	10,780,469 classes.dump

如何得到.smali代码(1)：手写代码

```
> type HelloWorld.smali
```

```
.class public LHelloWorld;
.superLjava/lang/Object;

.method public static main([Ljava/lang/String;)V
    .registers 2

    sget-object v0, Ljava/lang/System;->out:Ljava/io/PrintStream;
    const-string v1, "Hello World!"

    invoke-virtual {v0, v1}, Ljava/IOC/PrintStream;->println(Ljava/lang/String;)V

    return-void
.end method
```

```
> java -jar smali.jar -o classes.dex HelloWorld.smali
```

```
> zip HelloWorld.zip classes.dex
```

```
> adb push HelloWorld.zip /sdcard
```

```
> adb shell dalvikvm -cp /sdcard>HelloWorld.zip Foo
```

```
Hello, world
```

如何得到.smali代码(2)：反编译与编译

```
> java -jar apktool-1.4.3.jar d -f -r test.apk  
> dir .\test
```

2012/08/15 16:39	24,276	AndroidManifest.xml
2012/08/15 16:39	44	apktool.yml
2012/08/15 16:39	362,484	resources.arsc
2012/08/15 16:39	<DIR>	res
2012/08/15 16:39	<DIR>	smali

```
> java -jar apktool-1.4.3.jar b .\test test2.apk
```

...

如何得到.smali代码(3)：从java源码

```
> type Foo.java
```

```
class Foo {  
    public static void main(String[] args) {  
        System.out.println("Hello, world");  
    }  
}
```

```
> javac Foo.java
```

```
> %ANDROID_TOOLS%\dx.bat --dex --output=foo.jar Foo.class
```

```
> adb push foo.jar /sdcard
```

```
> adb shell dalvikvm -cp /sdcard/foo.jar Foo
```

```
Hello, world
```

```
> java -jar apktool-1.4.3.jar d -f foo.jar
```

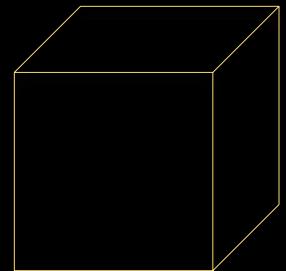
```
> dir .\foo.jar.out\smali
```

```
2013/10/25 02:25
```

```
594 Foo.smali
```

```
> java -jar aksmali-1.4.0.jar -lsb -o.\ -x foo.jar
```

```
> dir .\Foo.smali
```



概览：编码的三种基本行为

编码的三种基本行为

A: 准备数据

B: 运算

C: 代码声明与描述（伪指令）

编码的三种基本行为

A: 准备数据

- A1: 对象
- A2: 数组
- A3: 基础数据 (32 / 64位值)

编码的三种基本行为

A3：基础数据
(32/64位值)

布尔值	Boolean (Z)
字符	Char
字节	Byte
短整型	Short
整型	Int
浮点数	Float
双精度浮点数	Double
长整型	LongInt (J)
(无值)	Void

编码的三种基本行为

B: 运算

- B1: 调用对象方法
- B2: 调用vm指令

编码B2：调用vm指令

主要是面向基本数据类型的： $115 + 14 * 3$ 个

- 类型转换
- 算术运算
- 比较与跳转
- 成员存取 (aget/put, iget/put, sget/put)

有些指令是unused的： 26个

vm指令：类型转换 (1)

	Long	Float	Double	Int	Byte	Char	Short
Long		long-to-float	long-to-double	long-to-int			
Float	float-to-long		float-to-double	float-to-int			
Double	double-to-long	double-to-float		double-to-int			
Int	int-to-long	int-to-float	int-to-double		int-to-byte	int-to-char	int-to-short

vm指令：类型转换 (2)

	包装类	从字符串解析到 对象实例 (invoke-static)	从字符串解析到 基础类型 (invoke-static)	从基础类型转换到 对象实例 (invoke-static)	将对象转换到 基础类型 (invoke-virtual)
Z	java.lang.Boolean	valueOf(s)	parseBoolean(s)	valueOf(b)	booleanValue()
C	java.lang.Character		(注 2)	valueOf(c)	charValue()
B	java.lang.Byte		parseByte(s) parseByte(s, radix)	valueOf(b)	
S	java.lang.Short	decode(nm) valueOf(s)	parseShort(s) parseShort(s, radix)	valueOf(s)	byteValue() shortValue()
I	java.lang.Integer	valueOf(s, radix)	parseInt(s) parseInt(s, radix)	valueOf(i)	intValue() longValue()
J	java.lang.Long		parseLong(s) parseLong(s, radix)	valueOf(l)	floatValue() doubleValue()
F	java.lang.Float	valueOf(s)	parseFloat(s)	valueOf(f)	(注 1)
D	java.lang.Double	valueOf(s)	parseDouble(s)	valueOf(d)	

vm指令：算术运算

add-int v0, v1, v2

示例	Int	long	float	double
vAA = vBB + vCC	add-int sub-int mul-int div-int rem-int and-int or-int xor-int shl-int shr-int ushr-int	add-long sub-long mul-long div-long rem-long and-long or-long xor-long shl-long shr-long ushr-long	add-float sub-float mul-float div-float rem-float	add-double sub-double mul-double div-double rem-double

v0 = v0 + v1

add-int v0, v1

示例	Int	long	float	double
Rx = Rx + Ry	add-int/2addr	add-long/2addr	add-float/2addr	add-double/2addr

v0 = v0 + x

add-int v0, x

示例	Int/lit8	Int/lit16		
Rx = Ry + lit	add-int/lit8	add-int/lit16		

vm指令： 比较与跳转

	Float	Double	Long	值 < 32bits	0
格式	Op vAA, vBB, vCC			Op vA, vB, +CCCC	Op vAA, +BBBB
指令	cmpl-float cmpg-float	cmpl-double cmpg-double	cmp-long	if-eq if-ne if-lt if-ge if-gt if-le	if-eqz if-nez if-ltz if-gez if-gtz if-lez

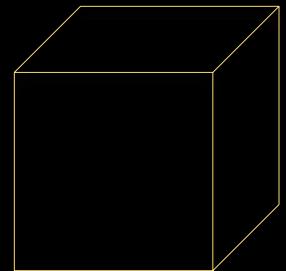
vm指令：成员存取

	对象属性	静态成员	数组元素
格式	...		
指令	iget iget-wide iget-object iget-boolean iget-byte iget-char iget-short iput iput-wide iput-object iput-boolean iput-byte iput-char iput-short	sget sget-wide sget-object sget-boolean sget-byte sget-char sget-short sput sput-wide sput-object sput-boolean sput-byte sput-char sput-short	aget aget-wide aget-object aget-boolean aget-byte aget-char aget-short aput aput-wide aput-object aput-boolean aput-byte aput-char aput-short

编码的三种基本行为

C: 伪指令

- 声明类
- 声明枚举
- 声明特定结构
- 声明注解
- 其它



一切都是对象？

最关键的指令: new-instance

```
## 使用v0创建StringBuilder
const-string v0, "Hello, "
new-instance v1, Ljava/lang/StringBuilder;
invoke-direct {v1, v0}, Ljava/lang/StringBuilder; ->
    <init>(Ljava/lang/String;)V

## append " World!", or more...
const-string v0, " World!"
invoke-virtual {v1, v0}, Ljava/lang/StringBuilder; ->
    append(Ljava/lang/String;)Ljava/lang/StringBuilder;

## 将结果返回到v2寄存器
invoke-virtual {v1}, Ljava/lang/StringBuilder; ->
    toString()Ljava/lang/String;
move-result-object v2
```

也可以调用类静态方法

```
## 整数5
const v0, 5

## 调用类静态方法java.lang.Integer.toString()
invoke-static{v0}, Ljava/lang/Integer;->
    toString(I)Ljava/lang/String;

## 返回结果值（字符串对象）到v1
move-result-object v1
```

枚举类型

```
## 枚举直接量 (enum literal)
.enum classType->fieldName:classType
```

```
## 将枚举直接量作为类静态成员
.field public static aStaticField: LMyClasses/AEnum; =
.enum LMyClasses/AEnum; ->enumValue1:LMyClass/AEnum;
```

```
## 将枚举直接量置入寄存器
sget-object v1,
    LMyClasses/AEnum; ->enumValue2:LMyClass/AEnum;
```

枚举类型（概览）

```
## 声明当前类与父类
## 声明该类型所有的枚举值（静态类成员）
...
## 声明一个私有的值数组
.field private static final synthetic $VALUES: [LMyClass/AEnum;
## 私有构造方法，调用java.lang.Enum的构造方法来初始化枚举类
.method private constructor <init>(Ljava/lang/String;I)V
...
## 实现Enum.values()方法，返回this.$VALUES数组的一个副本
.method public static values() [LMyClass/AEnum;
...
## 实现Enum.valueOf()方法，通过字符串来得到一个枚举值
.method public static valueof(Ljava/lang/String;) LMyClass/AEnum;
...
## 类构造方法：填写静态成员的值，并将值抄入$VALUES
.method static constructor <clinit>()V
...
```

枚举类型（示例）

```
## 声明当前类与父类
.class public final enum LMyClass/AEnum;
.super Ljava/lang/Enum;

## 声明一个私有的值数组
.field private static final synthetic $VALUES:[LMyClass/AEnum;

## 声明该类型所有的枚举值（静态成员）
.field public static final enum enumValue1:LMyClass/AEnum;
.field public static final enum enumValue2:LMyClass/AEnum;

## 私有构造方法，调用java.lang.Enum的构造方法来初始化枚举类
## （固定写法）
.method private constructor <init>(Ljava/lang/String;I)V
    .registers 3
    invoke-direct {p0, p1, p2}, Ljava/lang/Enum;-><init>(Ljava/lang/String;I)V
    return-void
.end method

## 实现Enum.values()方法，返回this.$VALUES数组的一个副本
## 通常用于for遍历，例如：for (AEnum enum: AEnum.values()) { ... }
## （除类名不同外，可采用固定结构）
.method public static values() [LMyClass/AEnum;
    .registers 1

    # 1. 取this.$VALUES
    sget-object v0, LMyClass/AEnum;->$VALUES:[LMyClass/AEnum;

    # 2. 克隆一个this.$VALUES数组的副本
    invoke-virtual {v0}, LMyClass/AEnum;->clone()Ljava/lang/Object;
    move-result-object v0

    # 3. 类型转换并返回结果值
    check-cast v0, LMyClass/AEnum;
    return-object v0
.end method

## 实现Enum.valueOf()方法，通过字符串来得到一个枚举值
## 通常用于类型转换或Java在语法层面实现switch等
## （除类名不同外，可采用固定结构）
.method public static valueof(Ljava/lang/String;)LMyClass/AEnum;
    .registers 2

    # 1. 声明类类型常量到v0
    const-class v0, LMyClass/AEnum;

    # 2. 为该枚举类调用java.lang.Enum.valueOf()方法
    invoke-static {v0, p0}, Ljava/lang/Enum;->
        valueof(Ljava/lang/Class;Ljava/lang/String;)Ljava/lang/Enum;
    move-result-object v1

    # 3. 类型转换并返回结果值
    check-cast v1, LMyClass/AEnum;
    return-object v1
.end method

.method static constructor <clinit>()V
...
.end method

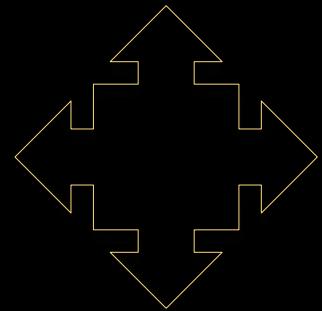
## 完善上例中的类构造方法
## （续上例，替换上例中的最后一个构造方法）
.method static constructor <clinit>()V
    .registers 3

    # 步骤一. 初始化各个类静态成员
    # 静态成员: MyClass.AEnum.enumValue1
    # 1.1 创建一个成员值(一个对象)，并声明它的字符串值与枚举序号
    new-instance v0, LMyClass/AEnum;
    const-string v1, "enumValue1"
    const/4 v2, 0
    # 1.2 调用上例中的内部构造方法
    # (进而会调用到java.lang.Enum中的构造方法来初始化该枚举值)
    invoke-direct {v0, v1, v2}, LMyClass/AEnum;-><init>(Ljava/lang/String;I)V
    # 1.3 写初始化值
    sput-object v0, LMyClass/AEnum;->enumValue1:LMyClass/AEnum;

    # 静态成员: MyClass.AEnum.enumValue2
    # (步骤同上)
    new-instance v0, LMyClass/AEnum;
    const-string v1, "enumValue2"
    const/4 v2, 1
    invoke-direct {v0, v1, v2}, LMyClass/AEnum;-><init>(Ljava/lang/String;I)V
    sput-object v0, LMyClass/AEnum;->enumValue2:LMyClass/AEnum;

    # 步骤二. 将各个类静态成员值填入内部的$VALUES数组
    # 2.1 创建数组，其大小为枚举成员个数
    const/4 v0, 2
    new-array v0, v0, [LMyClass/AEnum;
    # 2.2 向数据写入各个成员值
    sget-object v1, LMyClass/AEnum;->enumValue1:LMyClass/AEnum;
    const/4 v2, 0
    aput-object v1, v0, v2
    sget-object v1, LMyClass/AEnum;->enumValue2:LMyClass/AEnum;
    const/4 v2, 1
    aput-object v1, v0, v2
    # 2.3 将数组写入this.$VALUES
    sput-object v0, LMyClass/AEnum;->$VALUES:LMyClass/AEnum;

    return-void
.end method
```



复杂结构：数组

数组的一般使用

```
##  
## 需要使用2个寄存值，设为v0, v1；其中v0为操作的数组。  
##  
  
## 数组的元素个数  
const v1, <arrayElements>  
  
## 以指定数组类型 (arrayType) 创建一个定长数组，例如 [I，即元素类型为I  
new-array v0, v1, <arrayType>  
  
## 置值或取值 (with index 5, etc.)  
const v2, 5  
aget v1, v0, v2      # v1 = v0[v2]  
aput v1, v0, v2      # v0[v2] = v1
```

带初值的数组

```
## 数组的元素个数
const v1, 2                                # <arrayElements>

## 以指定数组类型 (arrayType) 创建一个定长数组, 例如 [I, 即元素类型为 I
new-array v0, v1, I                         # <arrayType>

## 将标签所指向的初值填入数组中
fill-array-data v0, :array_0    # <label_Array-Data>
...
:array_0
.array-data 0x4
    0x14t 0x0t 0x2t 0x1t
    0x15t 0x0t 0x2t 0x1t
.end array-data
```



流程控制

只有最简单的流程控制指令

异常其实只有一个 throw 指令

```
## 抛出异常对象v3
:try_start
    const-string v2, 'Hi, Error!!!'
    new-instance v3, Ljava/lang/Exception;
    invoke-direct {v3, v2}, Ljava/lang/Exception; ->
        <init>(Ljava/lang/String;)V
    throw v3
:try_end

## 伪指令: .catch / .catchall
.catch Ljava/lang/Exception; {:try_start .. :try_end} :handler_1

:goto_0
return-void

:handler_1
## 将异常对象置入v0
move-exception v0

## 重新抛出异常v0
## - throw将导致流程变更，因此可以不再需要goto语句 (goto :goto_0)
throw v0
```

多重分支 (packed-switch)

packed-switch指令示例（各个分支语句的构造）

1. switch指令

packed-switch v0, :pswitch_data_0

:goto _switch_default ## <<-这里的default分支的起点

.....

2. switch语句结束位置（后续其它指令自此开始）

:goto _switch_end

.....

3. 退出当前方法

return-void

4. 各switch分支

:pswitch_0

.....

goto :goto _switch_end

:pswitch_1

.....

goto :goto _switch_end

:pswitch_2

:pswitch_3

.....

5. 上述指令所引用的标签

:pswitch_data_0

.packed-switch **0x3e8**

:pswitch_0 #0x3e8

:pswitch_3 #0x3e9

:pswitch_1 #0x3ea

:pswitch_2 #0x3eb

.end packed-switch

多重分支 (sparse-switch)

packed-switch指令示例（各个分支语句的构造）

1. switch指令

sparse-switch v0, :sswitch_data_0

:goto _switch_default ## <<-这里的default分支的起点

.....

2. switch语句结束位置（后续其它指令自此开始）

:goto _switch_end

.....

3. 退出当前方法

return-void

4. 各switch分支

:sswitch_0

:sswitch_1

.....

:sswitch_2

:sswitch_3

.....

5. 上述指令所引用的标签

:sswitch_data_0

.sparse-switch 0x3e8

0x1 -> :sswitch_0

0x2 -> :sswitch_3

0x3 -> :sswitch_1

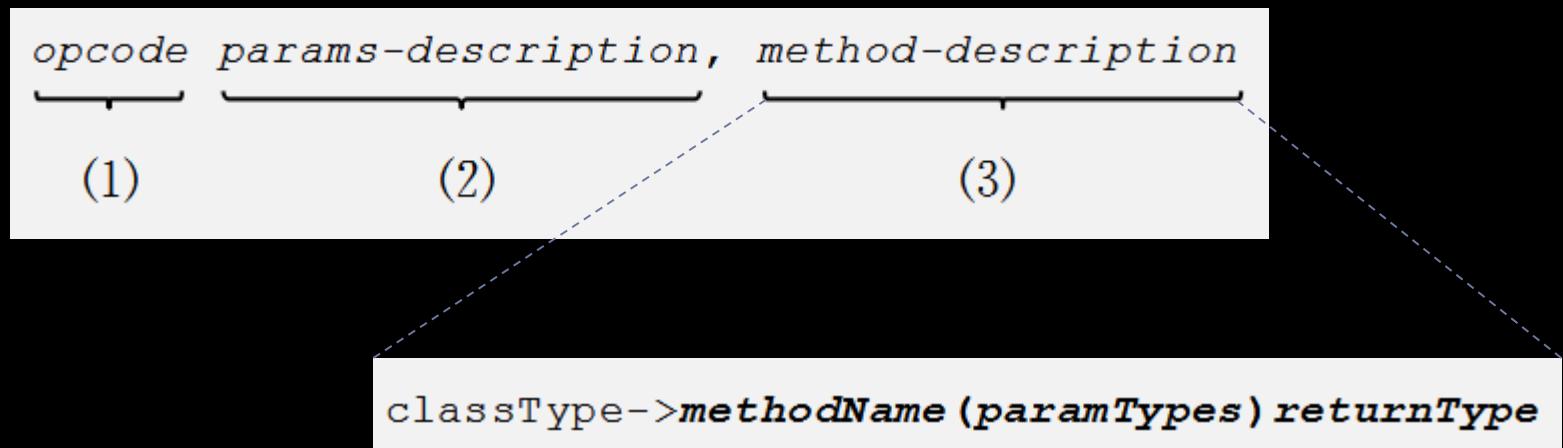
0x7f99 -> :sswitch_2

.end sparse-switch

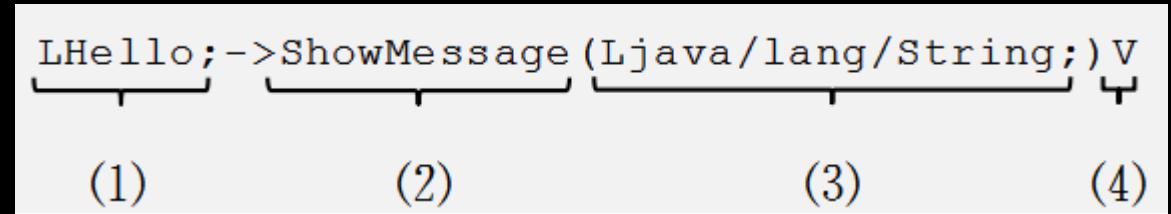
调用方法

指令	含义	语法限制
invoke-virtual	调用 <u>虚方法</u>	被调用方法不能使用 private、static 或 final 修饰，也不能是构造方法。
invoke-super	调用 <u>父类同名方法</u>	父类同名方法将作为 <u>虚方法</u> 调用，具有与 invoke-virtual 相同的限制。
invoke-direct	调用一般方法	被调用方法不能使用 static 修饰，可以是私有方法、构造器方法，或非重写的方法。
invoke-static	调用静态方法	被调用方法是使用 static 修饰的方法。
invoke-interface	调用接口方法	被调用方法是一个接口方法。

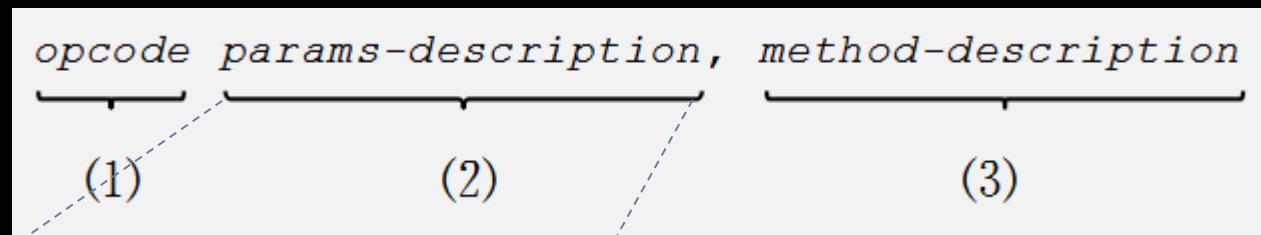
调用方法的两类指令



sample



调用方法的两类指令



{ vA, vB, .. }

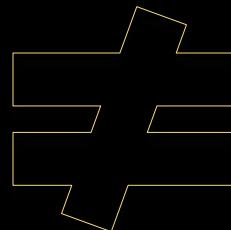
{ vX .. vY }

sample

```
invoke-virtual      {v0, v1}, Ljava/lang/String;->equals(Ljava/lang/Object;)Z
invoke-virtual/range {v0..v1}, Ljava/lang/String;->equals(Ljava/lang/Object;)Z
```

退出方法，返回值

返回	值获取	含义	注
return-void		没有返回值	
return xx	move-result yy	返回或获取 32 位值	1
return-wide xx	move-result-wide yy	返回或获取 64 位值	1,2
return-object xx	move-result-object yy	返回或获取对象引用	1



指令与编码

操作数: 寄存器

以下操作数可以放入寄存器以便进一步运算:

- 立即值 (literal value)
- 对象实例 (object references)

以下操作数仅能作为指令编码, 没有指令用于将它们放入寄存器。

- 类型索引 (type index)
- 字符串索引 (string index)
- 字段索引 (field index)
- 方法索引 (method index)
- 位置标签 (labeled address)

goto
goto/16
goto/32
throw
*if-**
sparse-switch
packed-switch
fill-array-data

nop 和 *return-void* 两个指令没有操作数。

那些不能放进寄存器的操作数

```
## file: Typetest.smali
```

```
.method public static main([Ljava/lang/String;)V
    .locals 4

    const-string v1, "abcd"

    return-void
.end method
```

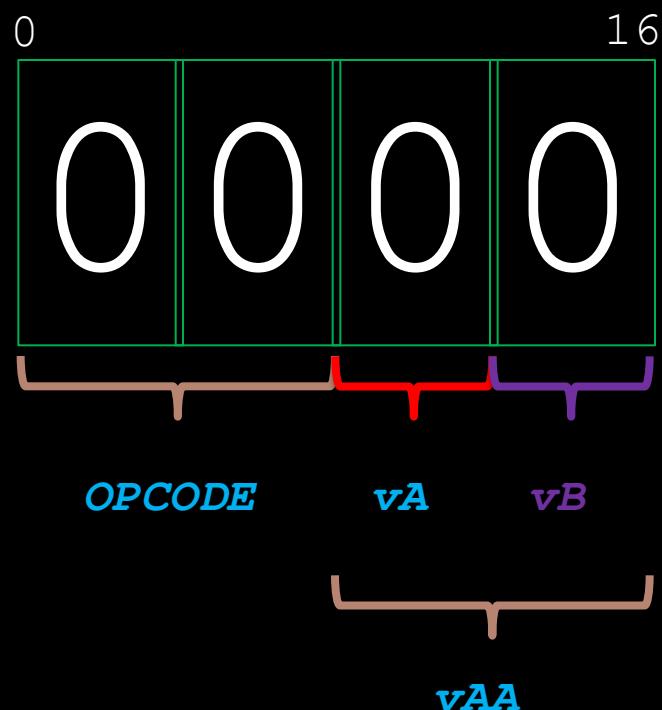
```
> java -jar smali-1.4.0.jar Typetest.smali -o Typetest.dex
> %ANDROID_TOOLS%\dexdump.exe -d Typetest.dex
```

```
00010c: [00010c] Typetest.main:([Ljava/lang/String;)V
00011c: 1a01 0700 | 0000: const-string v1, "abcd" // string@0007
000120: 0e00 | 0002: return-void
```

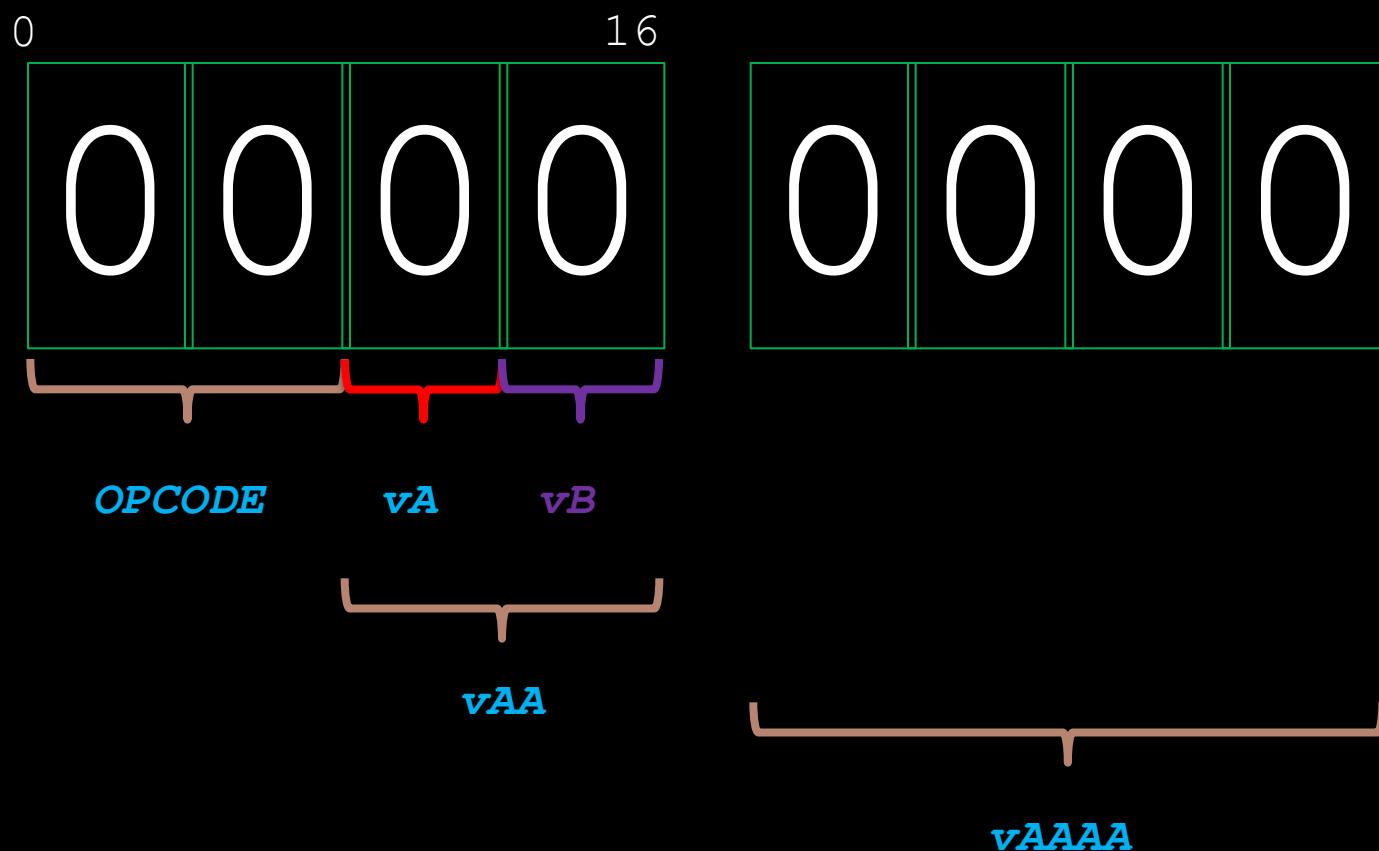
操作数：寄存器

编码规格	序号取值	表示法	称谓
0	0..15	vA	4 bits register
00	0..255	vAA	8 bits register
0000	0..65535	vAAAA	16 bits register

操作数：指令对齐与寄存器使用的关系

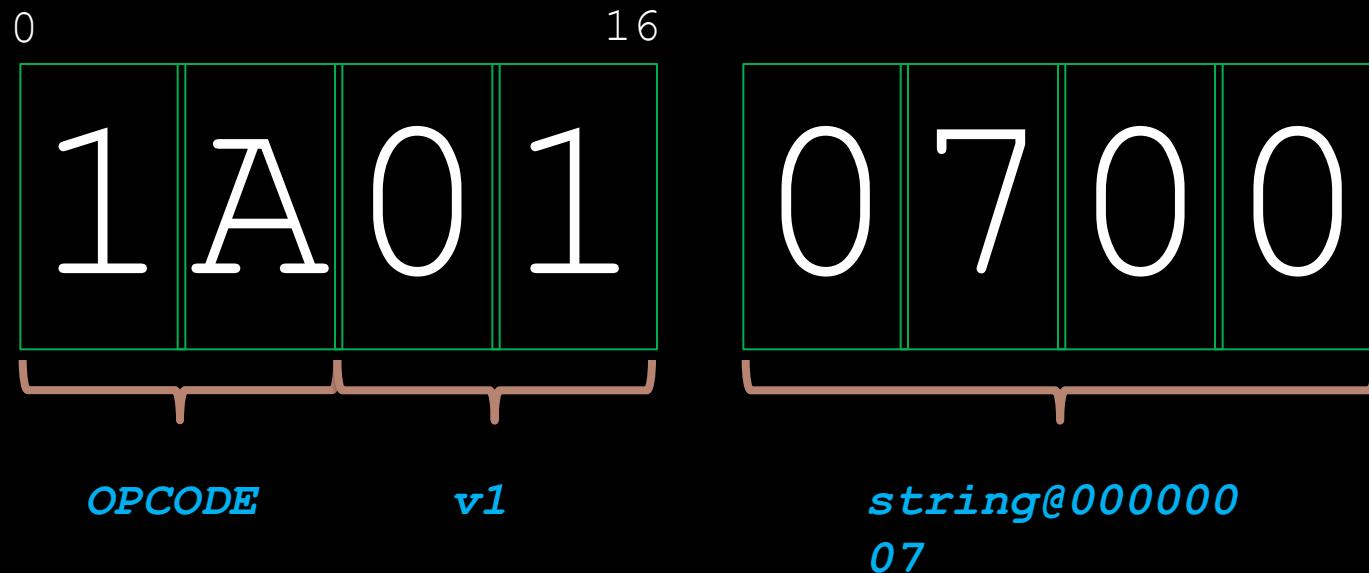


操作数：指令对齐与寄存器使用的关系



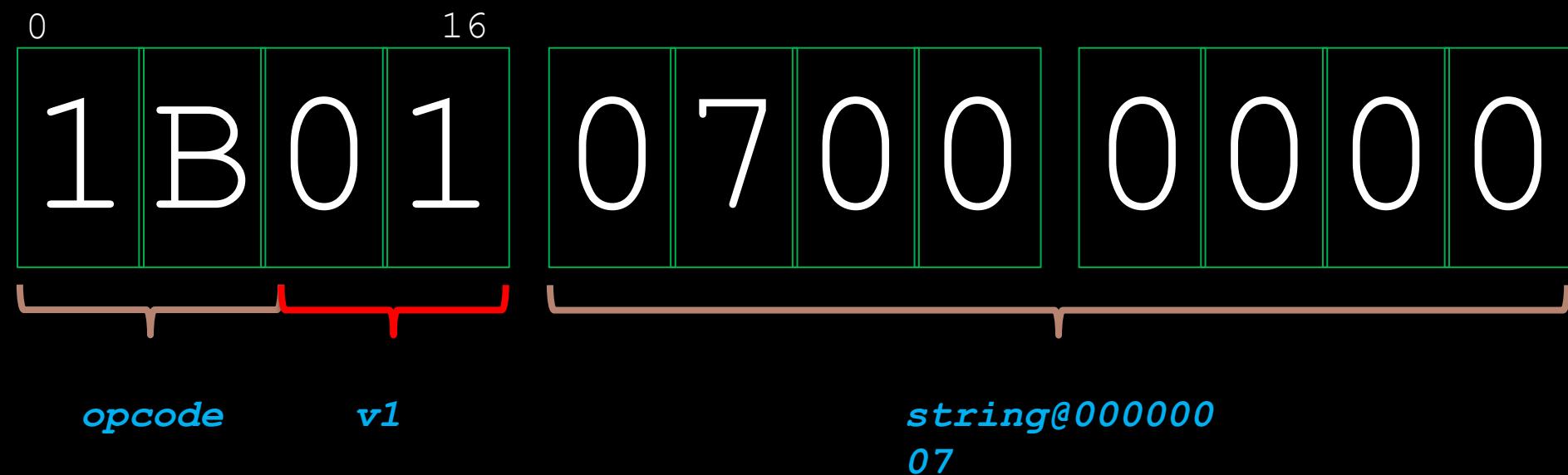
所以,

const-string, v1, "abcd"



所以,

const-string/jumbo , v1 , "abcd"



可执行的指令都在method中

对象初始化方法（实例构造方法）

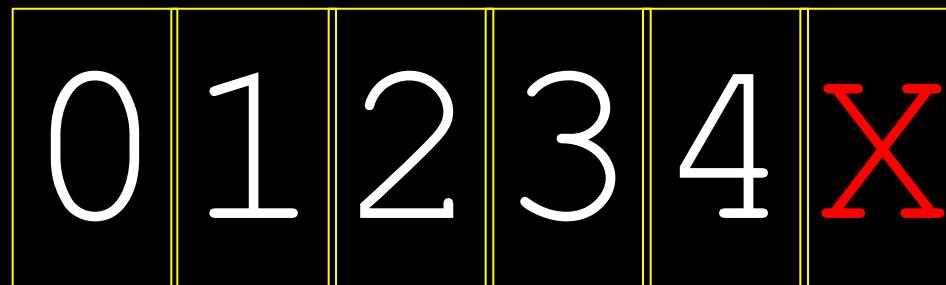
.method public constructor <init>() V

 .locals 5 **# .register 6**

 ## ...

.end method

p0 p1..pn



v0 v1 v2 v3 v4 v5



可用寄存器

可执行的指令都在method中

类静态方法

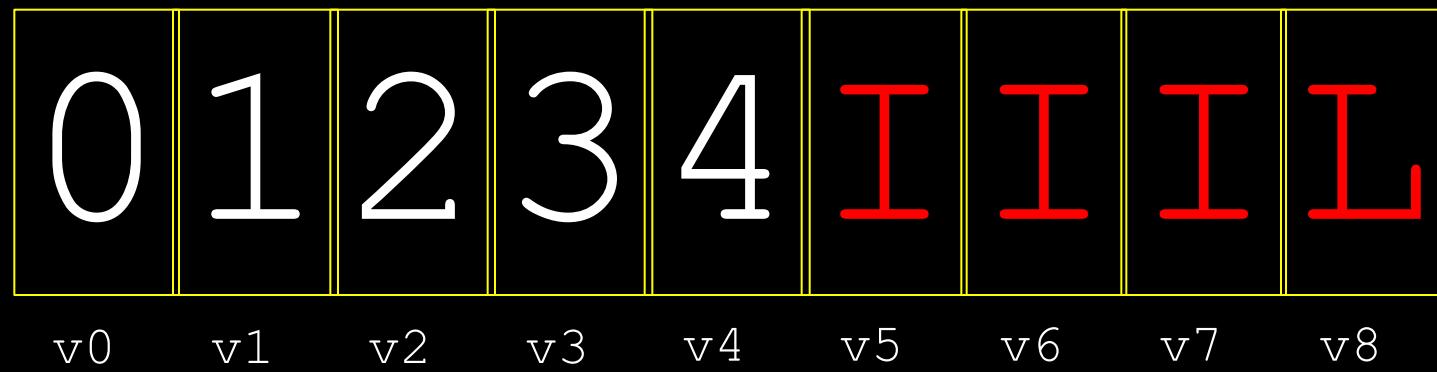
.method static foo(IIILjava/Lang/String;)V

 .locals 5 # .register 9

...

.end method

p0 p1 p2 p3



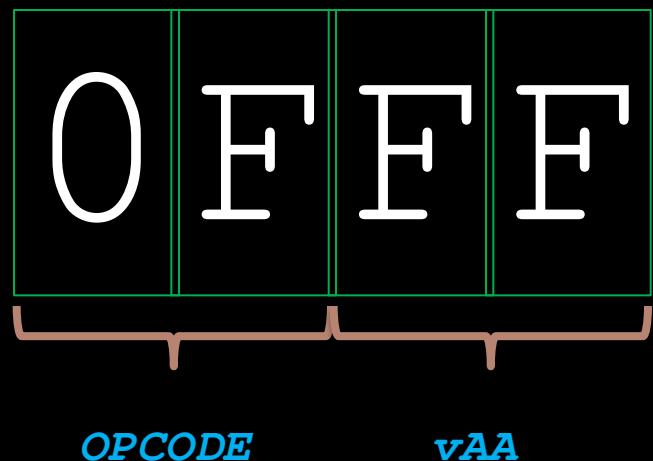
可用寄存器

返回值只能使用0..255号寄存器

```
## 公开方法foo2
.method public foo2() I
    .locals 400 # .register 401
```

```
## ...
```

```
move/from16 v255, v321      0          16
return v255
.end method
```

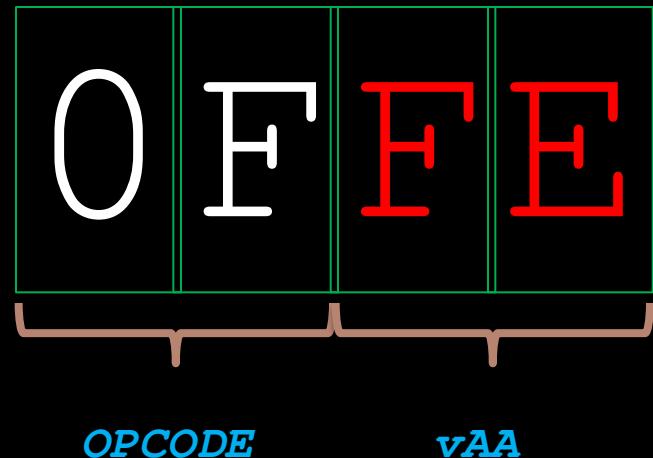


对于return-wide, 要小心一点儿

```
## 公开方法foo2
.method public foo2() I
    .locals 400 # .register 401
```

```
## ...
```

```
move/from16 v254, v321      0          16
    return-wide v254
.end method
```



两种方法调用

method@0003, dex: 6e54 0300 1632

invoke-virtual {v6, v1, v2, v3, v4}, Ljava/lang/Object;->it(IIII)V

6	E	5	4	0	3	0	0	1	6	3	2
---	---	---	---	---	---	---	---	---	---	---	---

method@0003, dex: 7405 0300 0100

invoke-virtual/range {v1..v5}, Ljava/lang/Object;->it(IIII)V

7	4	0	5	0	3	0	0	0	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---

End.

