Smali Introduction Manual

Smali grammar learning summary

1, About Smali

First of all, when it comes to smali, we'll talk about reverse, which is usually a means for security engineers (reverse Engineers) to systematically analyze three-party apps and do hacking and other malicious elements because of some interests (apk secondary packaging insert ads, hacking charging applications, malicious code implantation, plagiarism api, etc.).

Of course, technology is a double-edged sword. It lies in the person who uses technology and no longer the technology itself.

Back to the main point, SmalI is a loose Jasmin/dedexer syntax, which is Davlik's register language, similar to assembly language in syntax. One of the biggest differences between Dalvik VM and JVM is that Dalvik VM is register based. Register based means that all operations in SmalI must go through registers.

Smali and Baksmali are the names of compilers and decompilers in Icelandic respectively. You may ask why it's Icelandic, because Dalvik is the name of an Icelandic fishing village.

2, Smali syntax

1. Data type

In Davlik bytecode, registers are 32-bit and can support any type. Long/Double is represented by two registers.
There are two types of Dalvik bytecode: basic type and reference type (including object and array).

(1) Basic types

type	explain
V	void can only be used for return value types
Z	boolean
B	byte
S	short
C	char
I	int
J	long (64 bit)
F	float
D	double (64 bit)

(2) Reference type

type	explain
L	Object type (Lpackage/ObjectName; is equivalent to package.ObjectName; in java)
[I	Represents an integer one-dimensional array, equivalent to java int [];
[Ljava/lang/String	An array of objects representing a String

① Object type:

"L": indicates that this is an object type
"package/ObjectName ": the package and class name of the object, such as ljava / Lang / String = > java.lang.string
"; ": indicates the end of the object name

② Array representation:

"[I ": represents an integer one-dimensional array, equivalent to java's int [];
For multi-dimensional arrays, just add "[", [[i = > int [] []; note: each dimension has a maximum of 255;

③ Object array representation:

[Ljava/lang/String represents an array of String objects;

2. Basic grammar

2.1 expression

Java source code:

```
public void smaliExpression(){
    //Addition operation
    int a = 1;
    double b = 2.5;
    double c = a + b;

    //Subtraction operation
    double d = b - a;

    //Multiplication operation
    double e = a * b;

    //Division operation
    double f = b / a;

    //XOR operation
    int g = 3;
    int h = a ^ g;

    //Three mesh operation
    int i = a > b?a:g;
}
```

Smali Code:

```
.method public smaliExpression()V
    .locals 15

    .line 16
    const/4 v0, 0x1    #1

    .line 17
    .local v0, "a":I
    const-wide/high16 v1, 0x4004000000000000L    # 2.5

    .line 18
    .local v1, "b":D
    int-to-double v3, v0   //Strong 1 of int type to 1.0 of double

    add-double/2addr v3, v1  //Add two double types

    .line 21
    .local v3, "c":D
    int-to-double v5, v0

    sub-double v5, v1, v5    //Subtract V5 = V1-V5

    .line 24
    .local v5, "d":D
    int-to-double v7, v0

    mul-double/2addr v7, v1   //multiplication

    .line 27
    .local v7, "e":D
    int-to-double v9, v0

    div-double v9, v1, v9   //division

    .line 30
    .local v9, "f":D
    const/4 v11, 0x3

    .line 31
    .local v11, "g":I
    xor-int v12, v0, v11  //XOR

     //Exclusive or statement
    .line 34
    .local v12, "h":I
    int-to-double v13, v0

    cmpl-double v13, v13, v1  //Cmpl double comparison

    if-lez v13, :cond_0

    move v13, v0

    goto :goto_0
```

```
    :cond_0
    move v13, v11       //Give the value of register v11 to v13

    .line 35
    .local v13, "i":I
    :goto_0
    return-void
.end method
```

It is arranged as follows:

java operators	samli operator
addition	add-double/2addr
subtraction	sub-double
multiplication	mul-double/2addr
division	div-double
XOR	xor-int
Three mesh operation	Conditional statement

You can see that the trinomial operator itself is a conditional statement, so let's look at the details of the conditional statement.

2.2 conditional statement

Compare the code first.
java:

```
    public void smaliIf(){
        int a = 1, b = 2;
        int c = 0;
        if(a > b){
            c = a;
        }
        if(a < b){
            c = b;
        }
        if(a>=b){
            c = a;
        }
        if(a <= b){
            c = b;
        }
        if(a==b){
            c = a;
        }
        if(a != b){
            c = b;
        }
    }
```

samli:

```
.method public smaliIf()V
    .locals 3

    .line 38
    const/4 v0, 0x1

    .local v0, "a":I
    const/4 v1, 0x2

    .line 39
    .local v1, "b":I
    const/4 v2, 0x0

    .line 40
    .local v2, "c":I
    if-le v0, v1, :cond_0

    .line 41
    move v2, v0

    .line 43
    :cond_0
    if-ge v0, v1, :cond_1

    .line 44
    move v2, v1
```

```
    .line 46
    :cond_1
    if-lt v0, v1, :cond_2

    .line 47
    move v2, v0

    .line 49
    :cond_2
    if-gt v0, v1, :cond_3

    .line 50
    move v2, v1

    .line 52
    :cond_3
    if-ne v0, v1, :cond_4

    .line 53
    move v2, v0

    .line 55
    :cond_4
    if-eq v0, v1, :cond_5

    .line 56
    move v2, v1

    .line 58
    :cond_5
    return-void
.end method
```

The summary is as follows:

```
"If EQ vA, vB,: cond",: cond ",: cond",: cond ",, if vA is equal to vB

"If ne vA, vB,: cond * *" if vA is not equal to vB, skip to: cond * *, otherwise continue to exec

"If LT vA, vB,: cond * *" if vA is less than vB, skip to: cond * *, otherwise continue to execute

"If Ge vA, vB,: cond * *" if vA is greater than or equal to vB, skip to: cond * *, otherwise cont

"If GT vA, vB,: cond * *" if vA is greater than vB, skip to: cond * *, otherwise continue to exec

"If Le vA, vB,: cond * *" if vA is less than or equal to vB, skip to: cond * *, otherwise continu

"If EqZ vA,: cond",: cond ",: cond",: cond ",, if vA equals 0

"If Nez vA,: cond * *" if vA is not equal to 0, skip to: cond * *, otherwise continue to execute

"If ltz vA,: cond",: cond ",: cond",: cond ",, if vA is less than 0, skip to: cond" *, otherwise

"If Gez vA,: cond * *" if vA is greater than or equal to 0, skip to: cond * *, otherwise continue

"If GTZ vA,: cond",: cond ",: cond",: if vA is greater than 0, skip to: cond "*, otherwise contir

"If lez vA,: cond",: cond ",: cond",: cond ",: if vA is less than or equal to 0
```

Where "*" in cond "* is the number (representing 1,2,3.) , if there are multiple conditions in a method, the number cannot be repeated.

2.3 loop statement

Again, check the code first:
java:

```
    public void smaliWhile(){
        //while
        int a = 0;
        while(a<=3){
            a++;
        }

        //for
        int b = 0;
        for(int i = 0;i<3;i++){
            b++;
        }

        //do...while
```

```
        //do...while
        int c = 0;
        do{
            c++;
        }while (c <= 3);
    }
```

smali:

```
.method public smaliWhile()V
    .locals 5

    .line 62
    const/4 v0, 0x0

    move v1, v0

    .line 63
    .local v1, "a":I
    :goto_0
    const/4 v2, 0x3

    if-gt v1, v2, :cond_0   //If a > 3, jump to cond u 0

    .line 64
    add-int/lit8 v1, v1, 0x1   //if condition above does not exist, a increases by 1

    goto :goto_0  //Loop body, continue to execute downward from goto 0

    .line 68
    :cond_0
    const/4 v3, 0x0

    .line 69
    .local v3, "b":I
    move v4, v3

    move v3, v0

    .local v3, "i":I
    .local v4, "b":I
    :goto_1
    if-ge v3, v2, :cond_1      // i >= 3

    .line 70
    add-int/lit8 v4, v4, 0x1   // b++;

    .line 69
    add-int/lit8 v3, v3, 0x1   //i++;

    goto :goto_1     //Circulation main body

    .line 74
    .end local v3    # "i":I
    :cond_1
    nop             //It means empty operation, do nothing

    .line 76
    .local v0, "c":I
    :cond_2
    add-int/lit8 v0, v0, 0x1

    .line 77
    if-le v0, v2, :cond_2

    .line 78
    return-void
.end method
```

So the key point of the loop is to use: goto "* mark at the beginning of the loop body, goto: goto" * mark at the place where the loop is executed, and judge whether to jump out of the loop and still use the above conditional statements.

2.4 try catch statement

Continue with code
java:

```
    public void smaliTryCatch(){
        Object a = null;
        try {
            a = null;
```

```
        }catch (Exception e){
            a.toString();
        }finally {
            a = new Object();
        }
    }
```

smali:

```
.method public smaliTryCatch()V
    .locals 2

    .line 81
    const/4 v0, 0x0

    .line 83
    .local v0, "a":Ljava/lang/Object;
    const/4 v0, 0x0

    .line 87        //v1: new Object();
    new-instance v1, Ljava/lang/Object;

    invoke-direct {v1}, Ljava/lang/Object;-><init>()V

    move-object v0, v1

    .line 88
    nop

    .line 89
    return-void
.end method
```

The try catch here is very strange. It seems that the catch part performs the nop (null operation). Is it smali who will not handle exceptions? I don't know. If you know anything, please let me know.
Here is a new Object process. What is the new Object like in smali? This is the content of. line87.

This is the basic syntax. Let's use a simple class file to see the smali class file structure.

3. Class file structure

Because the Smali code of a class file is relatively long, we explain it separately.

3.1 header information

```
.class public Lcom/justart/samlidemo/MainActivity;
.super Landroid/app/Activity;
.source "MainActivity.java"
```

. class means class path package + class name
. super represents the path and address of the parent class
. source indicates the source file name

3.2 construction method

```
# direct methods
.method public constructor <init>()V
    .locals 0

    .line 6
    invoke-direct {p0}, Landroid/app/Activity;-><init>()V

    return-void
.end method
```

Because I did not override the constructor in the source code, the default nonparametric constructor directly calls the nonparametric constructor of the parent class Activity.

3.3 other methods

Take the onCreate method of Activity as an example:

```
# virtual methods   //Representation is a virtual method
.method protected onCreate(Landroid/os/Bundle;)V
    .locals 1
    .param p1, "savedInstanceState"    # Landroid/os/Bundle;

    .line 10
    invoke-super {p0, p1}, Landroid/app/Activity;->onCreate(Landroid/os/Bundle;)V

    .line 11
    const/high16 v0, 0x7f050000
```

```
    invoke-virtual {p0, v0}, Lcom/justart/samlidemo/MainActivity;->setContentView(I)V

    .line 12
    return-void
.end method
```

(1) Method starts with. Method and ends with. end method;
(2) V at the end of the first line of the method indicates that the return type is void. For other return types, see the data type in Section 1;
(3) The method parameter also follows the smali data type, which indicates that the parameter is a Bundle object type;
(4) . param represents the parameters of the method, and the default parameter is p0;
(5) Finally, the return type of the method here represents the return void.

3.4 appendix

Here is a brief summary of some commonly used keywords in the class:

Key word	Explain
.class	Define class type package name + class name
.super	Define the path and address of the parent class
.source	Represents the source file name
filed	Definition field
.method...end method	Definition method
.prologue	Here we go
.annotation...end annotation	Definition annotation
.implements	Define interface instructions
.local	Specifies the number of local variables in the method
.registers	Specifies the total number of registers used within the method
.prologue	Represents the beginning of code in a method
.line	Represents the specified line in the java source file
.paramter .param	Parameter specified for method

Three, summary

The article is relatively rough, I hope you can give some suggestions and point out the mistakes, thank you!

Let's talk about my learning experience. After combing through the smali syntax, I found that it is helpful to understand the execution process of java code, as well as the introduction knowledge related to registers. After being familiar with smali, I can debug three parties or modify the APP. Please refer to another article of mine for simple modification of three party APP code Decompile three party apk and add debug log.

Just Enough
Published 9 original articles, won praise 10, and visited 1289
Private letter follow

Tags: Java less Assembly Language jvm

Posted on *Fri, 06 Mar 2020 04:42:21 -0500* by **ShadowX**