JVM internal principle (6) - one of the basics of Java bytecode

JVM internal principle (6) - one of the basics of Java bytecode

Introduction

Version: Java SE 7

Why do you need to know Java bytecode?

Whether you're a Java developer, architect, CxO, or a regular user of a smartphone, Java bytecode is in front of you, and it's the foundation of the Java virtual machine.

Directors, managers, and non-technical people can do it easily: all they need to know is that the development team is working on the next version of development, and Java bytecode is silently running on the JVM platform.

Simply put , Java bytecode is an intermediate representation of Java code (eg, a class file) that is executed inside the JVM, so why do you care about it? Because without Java bytecode, Java programs can't run because it defines how Java developers write code.

From a technical perspective , the JVM converts Java bytecode into native code at runtime as JIT compilation. If no Java bytecode is running behind it, the JVM cannot be compiled and mapped to native code.

Many IT professionals may not have time to learn assembler or machine code, and Java bytecode can be thought of as something similar to the underlying code. But when something goes wrong, understanding the basic operating principle of the JVM is very helpful in solving the problem.

In this article, you'll learn how to read and write JVM bytecode, better understand how the runtime works, and the ability to structure some key libraries.

This article will cover the following topics:

- How to get a list of bytecodes
- How to read bytecode
- How the language structure is mapped by the compiler: local variables, method calls, conditional logic
- Introduction to ASM
- How bytecode works in other JVM languages such as Groovy and Kotlin

table of Contents

    - Detailed local variables
    - Process control
    - Arithmetic operations and conversion
    - New &
    - Method call and parameter passing
- Part II: ASM
    - ASM and tools
- Part III: Javassist
- to sum up

Introduction to Java bytecode

The Java bytecode is the form in which the instructions in the JVM run. Java programmers usually don't need to know how Java bytecode works. But understanding the underlying details of the platform can make us a better programmer (we all want to be better programmers, isn't it?)

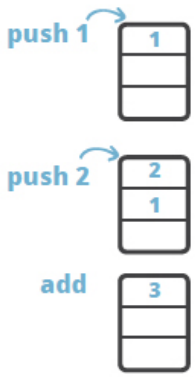
Understanding the bytecode and how the Java compiler generates bytecode is as much a knowledge as C or C++ programmers have assembly language.

Understanding bytecode is critical to writing program tools and program analysis, and applications can modify the bytecode and adapt the behavior of the application to different domains. Performance analysis tools, mocking frameworks, AOP, programmers need to have a thorough understanding of Java bytecode in order to write these tools.

basis

Let's take a very basic example to show you how Java bytecode works. Take a look at this simple expression, 1 + 2, expressed in inverse Polish as 1 2 + . What are the benefits of using reverse Polish markers here? Because this expression can be easily calculated using the stack:

After executing the "add" instruction, result 3 is at the top of the stack.



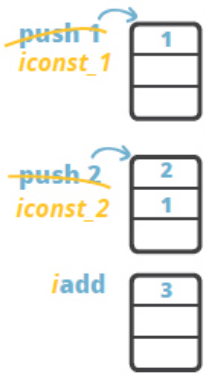
The computational model of Java bytecode is a stack-oriented programming language. The above example is the same as the Java bytecode directive. The only difference is that the opcode has some specific syntax:

Essay classification

Apache (7)
AutoTest(7)
CAS(17)
CXF(5)
Dubbo(1)
Elastic(43)
ElasticSearch(43)
ELK(8)
IT(2)
Java(15)
JavaScript(1)
Java concurrent programming (5)
JBoss(3)
JMeter (3)
JNDI(1)
LoadTest(2)
Log(32)
Mac(7)
Maven(1)
Midware(1)
Nginx(1)
Organizations(1)
Redis(1)
Sahi(3)
Selenium(4)
Servlet(1)
SOA(7)
Spring(1)
SSO(14)
TDD(1)
Test(10)
UML(1)
Vagrant(3)
VM(3)
Web Service(5)
Wildfly(1)
Distributed (3)
High availability HA (2)
Architecture
Interview
Agile (1)
History notes
Favorite repost (1)
Algorithm (7)
Source code analysis (1)

Essay file

March 2017 (5)
January 2017 (6)
December 2016 (9)
November 2016 (3)
March 2016 (29)



The opcodes **iconst_1** and **iconst_2** push the constants 1 and 2 respectively into the stack. The instruction **iadd sums** the two integers and **pushes** the result onto the top of the stack.

Basic characteristics

As the name implies, the **Java bytecode** consists of one byte of instructions, so the opcode has 256 possibilities. The actual number of instructions is slightly less than the number allowed, and there are about 200 opcodes used. Some opcodes are reserved for debugger operations.

An instruction consists of a type prefix and an operation name. For example, the "i" prefix means "integer", so the iadd directive means that the summation operation is for integers.

Depending on the nature of the instructions, we can divide them into four categories:

- **Stack operation instructions, including iterations of local variables**
- **Process control instruction**
- **Object operations, including method calls**
- **Arithmetic and type conversion**

There are also instructions for some special tasks, such as synchronizing and throwing exceptions.

Javap

To get a list of compiled class files, you can use the javap tool, a standard Java class file decompiler that ships with the JDK.

Let's use an application (moving average calculator) as an example:

```
public class Main {
    public static void main(String[] args){
        MovingAverage app = new MovingAverage();
    }
}
```

After the class file is compiled, in order to get the above bytecode list, you can execute the following command:

```
javap -c Main
```

The results are as follows:

```
Compiled from "Main.java"
public class algo.Main {
  public algo.Main();
    Code:
    0: aload_0
    1: invokespecial #1         // Method java/lang/Object."":()V
    4: return

  public static void main(java.lang.String[]);
    Code:
    0: new           #2         // class algo/MovingAverage
    3: dup
    4: invokespecial #3             // Method algo/MovingAverage."":()V
```


latest comment

1. Re: JMeter (1) - JMeter and WebDriver installation and testing (101 Tutorial)

May I ask the landlord var wait = new pkg.WebDriverWait (WDS.browser, 1) What does this mean?
Also, how can I make his execution slower, too fast to see ==

--No high-rise building in the northwest

2. Re: ElasticSearch 2 (12) - Shard number tuning (ElasticSearch performance)

Hello blogger! I learned a lot from your blog post, thank you!
There is a question I would like to ask, if the number of shards is 1, can it solve the problem of deep paging?

--PlayInLife

3. Re: IT architecture standards for IT architecture - mind map

Can't see the picture blurred!

--huababy

4. Re:ELK Stack (2) -- ELK + Redis collects Nginx logs

learned!

- Strict specifications - Kungfu to home

5. Re:ELK Stack (2) -- ELK + Redis collects Nginx logs

Xuexile~~

- Strict specifications - Kungfu to home

Reading the leaderboard

1. ElasticSearch 2 (6) - Plugin installs Head, Kopf and Bigdesk (22834)

2. ELK Performance (1) - Logstash Performance and Alternatives (16446)

3. [Java concurrent programming (a)] Thread pool FixedThreadPool vs CachedThreadPool ... (9023)

4. ElasticSearch 2 (12) - Shard number tuning (ElasticSearch performance) (8044)

5. CAS (5) - Browser access to the CAS server configuration in Nginx proxy mode (6213)

Comment leaderboard

```
    7: astore_1
    8: return
}
```

You can find the default constructor and a main method. Java programmers may know that if you don't specify any constructors for your class, there will still be a default constructor, but you may not realize where it is. Yes, right here! This default constructor exists in the compiled class, so it is generated by the Java compiler.

The constructor body is empty, but some instructions are still generated. why? Each constructor will call super() , right? This is not naturally generated, which is why bytecode instructions generate default constructors. Basically this is the call to super().

The main method creates an instance of the MovingAverage class and returns.

Perhaps you have noticed that some instructions reference the numeric parameters #1, #2, #3. These are all references to the constant pool. So how do we find these constants? How to check the constant pool in the list? You can decompile a class by using javap with the **-verbose** parameter:

```
$ javap -c -verbose HelloWorld
```

Some of the following printed sections are interesting:

```
Classfile /Users/anton/work-src/demox/out/production/demox/algo/Main.class
  Last modified Nov 20, 2012; size 446 bytes
  MD5 checksum ae15693cf1a16a702075e468b8aaba74
  Compiled from "Main.java"
public class algo.Main
  SourceFile: "Main.java"
  minor version: 0
  major version: 51
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Methodref          #5.#21         //  java/lang/Object."":()V
   #2 = Class              #22            //  algo/MovingAverage
   #3 = Methodref          #2.#21         //  algo/MovingAverage."":()V
   #4 = Class              #23            //  algo/Main
   #5 = Class              #24            //  java/lang/Object
```

There is a lot of information about the class here: When was it compiled, and what is the MD5 checksum? It is compiled from which *.java file, what is the version that follows Java, and so on.

We can also see accessor flags: ACC_PUBLIC and ACC_SUPER. The ACC_PUBLIC flag is intuitively easier to understand: our class is public, so the access token indicates that it is public. But what does ACC_SUPER do? ACC_SUPER was introduced to solve the problem of calling the super method with the invokespecial directive. It can be understood as a defect patch for Java 1.0, and only then can it find the super class method correctly. Starting with Java 1.1, the compiler always generates an ACC_SUPER access ID in bytecode.

You can also find the constant definition represented in the constant pool:

```
 #1 = Methodref          #5.#21         //java/lang/Object."":()V
```

The definition of a constant is configurable, meaning that a constant can also consist of other constants that refer to the same table.

Other details can be found when using the javap -verbose parameter. The method can output more information:

```
public static void main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=2, args_size=1
```

The access identifier is also generated in the method, and you can see how much stack depth is required for a method to execute, how many parameters to receive, and how many parameters the local variable table needs to hold for local variables.

JVM stack model

1. CAS (1) - configure CAS to Tomcat (s erver) under Mac (8)

2. What is one of the ultimate solutions to JavaScript closures? Basic concepts (4)

3. Apache CXF implements Web Service (1) - implements a pure JAX-WS web se rvice without heavyweight web containe r and Spring (4)

4. ElasticSearch 2 (9) - Under ElasticSe arch (illustrated story) (4)

5. JMeter (1) - JMeter and WebDriver In stallation and Testing (101 Tutorial) (4)

Recommended leaderboard

1. ElasticSearch 2 (10) - Under ElasticS earch (in-depth understanding of Shard and Lucene Index) (4)

2. CAS (5) - Browser access to the CAS server configuration in Nginx proxy mod e (4)

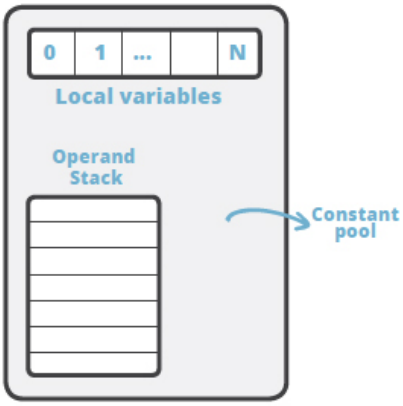
3. ELK Performance (4) - Best Practices for Large-Scale Elasticsearch Cluster Per formance (3)

4. ELK Performance (1) - Logstash Perfo rmance and Alternatives (2)

5. Anatomy of the Elasticsearch cluster - one (2)

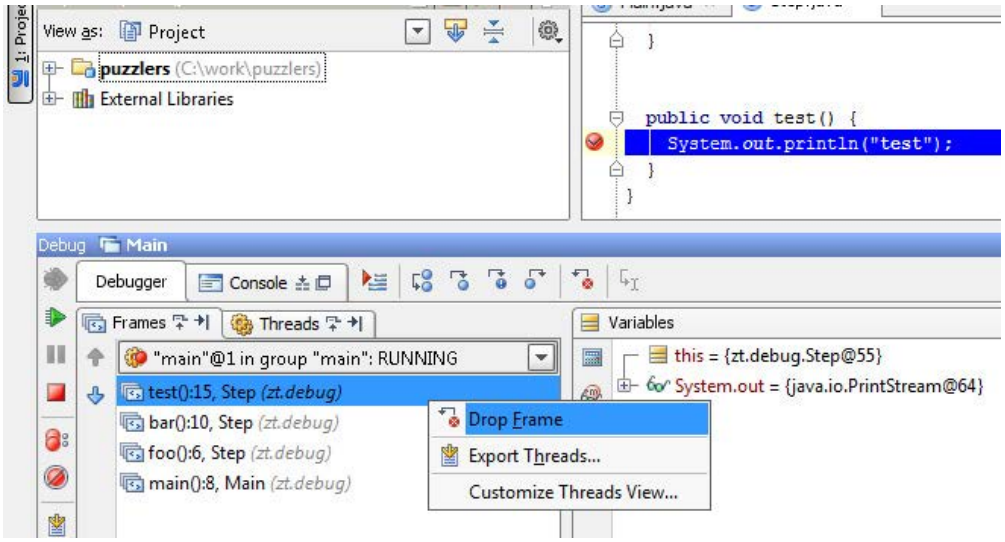
In order to understand the bytecode in more detail, we have a conception of the execution model of the bytecode. The JVM is a virtual machine based on stack mode. Each thread has a JVM stack for storing stack information. Every time a method is called, it is created. The 包括 includes the operand stack, a list of local variables, and a reference to the runtime constant pool of the current method of the current class. These can all be seen in the Main class of the decompiled at the beginning.

The local variable array is also known as the local variable list, which includes the parameters of the method and is also used to hold the value of the local variable. The size of the local variable list is determined at compile time, depending on the size of the numeric and local variables and the parameters of the method.



The operand stack is a last in, first out (LIFO) stack that is used to push and pop values. Its size is also determined at compile time. Some opcode instructions push values onto the operand stack; some perform pop operations, compute them, and push the results onto the stack. The operand stack is also used to receive the return value of the method.

In the debugging tool, we can roll back, but the state of the field does not fall back to the previous state.



What is inside the method body?

When looking at the list of bytecodes in the HelloWorld example, you might wonder what the numbers before each instruction mean? Why are the intervals between numbers not equal?

```
0: new           #2       // class algo/MovingAverage
3: dup
4: invokespecial #3       // Method algo/MovingAverage."":()V
7: astore_1
8: return
```

Cause: Some opcodes have parameters that require a bytecode list space. For example, new occupies three positions in the list: one is reserved for itself and the other two are reserved for input parameters. Therefore, the next instruction *dup is* at the position of index index 3.

As shown below, we consider the method body as an array:

Each instruction has its own hexadecimal representation. We can get the body of the method as a hexadecimal string as follows:

Open the class file with a hex editor to find the string:

```
0000140 00 01 b1 00 00 00 02 00 09 00 00 00 06 00 01 00
0000150 00 00 03 00 0a 00 00 00 0c 00 01 00 00 00 05 00
0000160 0b 00 0c 00 00 00 09 00 0d 00 0e 00 01 00 08 00
0000170 00 00 41 00 02 00 02 00 00 00 09 bb 00 02 59 b7
0000180 00 03 4c b1 00 00 00 02 00 09 00 00 00 0a 00 02
0000190 00 00 00 08 00 08 00 0f 00 0a 00 00 00 16 00 02
00001a0 00 00 00 09 00 0f 00 10 00 00 00 08 00 01 00 11
00001b0 00 12 00 01 00 01 00 13 00 00 00 02 00 14
```

It is also possible to modify the bytecode through the hex editor, although it is easier to do so. In addition to this, there are some simpler ways to use bytecode manipulation tools such as ASM or Javassist.

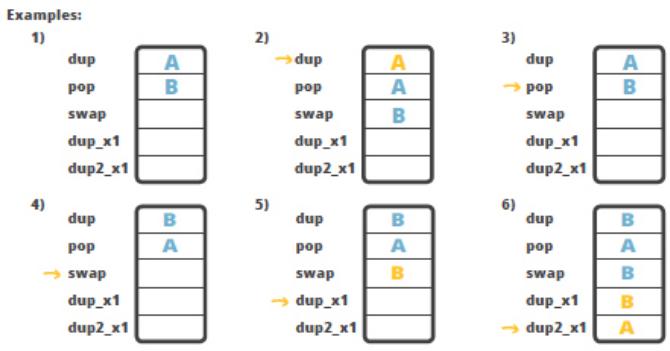
It doesn't have much to do with this knowledge point yet, but now you know what the source of these numbers is.

Detailed local stack

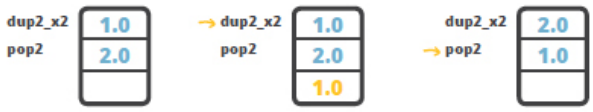
There are many ways to operate the stack. We have already mentioned some basic stack manipulation instructions: stacking or popping values. The swap instruction swaps the two values at the top of the stack.

Here are some examples of instructions that operate on values in the stack. Some basic instructions: dup and pop. The dup instruction repeats the value at the top of the stack and pushes it back onto the stack. The pop instruction removes the value at the top of the stack.

There are also some more complex instructions like **swap** , **dup_x1** and **dup2_x1** . The swap instruction, like its name, swaps the two values at the top of the stack, such as A and B swap locations; **dup_x1** copies and inserts the value at the top of the stack into the bottom of the stack (such as 5). **Dup2_x1** copies and inserts the two values at the top of the stack to the bottom of the stack (such as 6).



The dup_x1 and **dup2_x1** instructions look a bit confusing - why would anyone need this behavior - copy the value of the top of the stack and insert it at the bottom of the stack? Here are some more practical examples: How do you exchange two values of type double? The problem here is that the double type takes up two positions on the stack, which means that if we have two double values, we will have four positions on the stack. In order to exchange two double values we might think of using the **swap** instruction, but the problem is that it can only operate on one word of the instruction, that is, it cannot operate on double, and the instruction **swap2** does not exist. Instead, we can use the **dup2_x2** instruction to copy the two values at the top of the stack and insert them into the bottom of the stack, then we can use the **pop2** instruction. This way, you can successfully swap two double values.



Detailed local variables

The stack is used for execution. Local variables are used to store intermediate results and interact directly with the stack.

Now let's add some code to the previous example:

```
public static void main(String[] args) {
  MovingAverage ma = new MovingAverage();

  int num1 = 1;
  int num2 = 2;

  ma.submit(num1);
  ma.submit(num2);

  double avg = ma.getAvg();
}
```

I provide two values for the MovingAverage class and let him calculate the average of the current values. The resulting bytecode is as follows:

```
Code:
    0: new           #2          // class algo/MovingAverage
    3: dup
    4: invokespecial #3          // Method algo/MovingAverage."":()V
    7: astore_1

    8: iconst_1
    9: istore_2

    10: iconst_2
    11: istore_3

    12: aload_1
    13: iload_2
    14: i2d
    15: invokevirtual #4         // Method algo/MovingAverage.submit:(D)V

    18: aload_1
    19: iload_3
    20: i2d
    21: invokevirtual #4         // Method algo/MovingAverage.submit:(D)V

    24: aload_1
    25: invokevirtual #5         // Method algo/MovingAverage.getAvg:()D
    28: dstore      4
    LocalVariableTable:
    Start  Length  Slot  Name   Signature
            0   31  0  args   [Ljava/lang/String;
            8   23  1  ma     Lalgo/MovingAverage;
            10      21  2  num1   I
            12      19  3  num2   I
            30      1   4  avg    D
```

After creating the local variable of type MovingAverage, store the value in the local variable **ma** , using the **astore_1** command: 1 is the sequence number of **ma** in the local variable table (LocalVariableTable).

Next, the instructions **iconst_1** and **iconst_2 are** used to load constants 1 and 2 onto the stack, and then store them in the 2 and 3 positions of the LocalVariableTable via the **istore_2** and **istore_3** instructions.

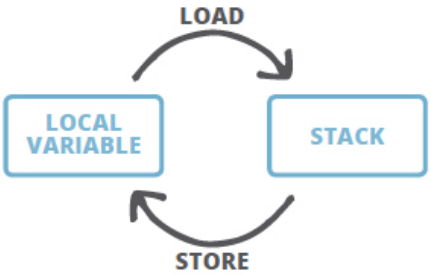
Note that the instruction that calls the class store is actually a pop operation, which is why in order to use the variable value again, we need to load it into the stack again. For example, in the above list, before calling the submit method, we need to load the value of the parameter into the stack again:

```
12: aload_1
13: iload_2
14: i2d
15: invokevirtual #4 // Method algo/MovingAverage.submit:(D)V
```

The result returned after calling the getAvg() method is pushed onto the stack and re-entered into the local variable. The dstore directive is used because the target variable is of type double.

```
24: aload_1
25: invokevirtual #5 // Method algo/MovingAverage.getAvg:()D
28: dstore 4
```

The more interesting thing is that the first position of the local variable list (LocalVariableTable) is occupied by the method parameters. In our current example, it's a static method, where no reference to this in the table points to the 0 position. However, for non-static methods, this will point to the 0 position.



This part aside, once you want to assign a local variable, it also means that you want to use the corresponding instructions stored them together (**Store**), for example, **astore_1** . The store directive always pops the value at the top of the stack. The corresponding **load** instruction takes the value out of the local variable list and writes it to the stack, but this value is not removed from the local variable.

Process control

The flow control instructions organize the execution order according to different situations. If-Then-Else, ternary opcodes, various loops, and even various error handling opcodes are also part of the **Java bytecode** flow control. Now these concepts have become jumps and gotos.

Now let's make some changes to the example so that it can handle any number of numbers passed into the submit method of the MovingAverage class:

```
MovingAverage ma = new MovingAverage();
for (int number : numbers) {
    ma.submit(number);
}
```

Suppose the variable numbers is a static field of the same class. The bytecode corresponding to the loop iteration on numbers is as follows:

```
0: new #2 // class algo/MovingAverage
3: dup
4: invokespecial #3 // Method algo/MovingAverage."":()V
7: astore_1
8: getstatic #4 // Field numbers:[I
11: astore_2
12: aload_2
13: arraylength
14: istore_3
15: iconst_0
16: istore 4
18: iload 4
20: iload_3
21: if_icmpge 43
24: aload_2
25: iload 4
27: iaload
28: istore 5
30: aload_1
31: iload 5
```

```
    33: i2d
    34: invokevirtual #5       // Method algo/MovingAverage.submit:(D)V
    37: iinc          4, 1
    40: goto          18
    43: return
  LocalVariableTable:
  Start  Length  Slot  Name   Signature
      30       7     5  number I
      12      31     2  arr$   [I
      15      28     3  len$   I
      18      25     4  i$     I
       0      49     0  args   [Ljava/lang/String;
       8      41     1  ma     Lalgo/MovingAverage;
      48     1   2  avg    D
```

Instructions at 8 and 16 are used to organize loop control. You can see that there are three variables in the local variable list (*LocalVariableTable*) that are not reflected in the source code: *arr$* , *len$* , *i$* , which are loop variables. The number field stored in the variable *arr$* , the length of the loop *len$* comes from the array length instruction *arraylength* . The loop counter, *i$ is* incremented by the *iinc* command after each loop .

The first instruction of the loop body is used to compare the loop counter with the length of the array:

```
18: iload 4
20: iload_3
21: if_icmpge 43
```

We load *i$* and *len$* into the stack and call *if_icmpge* to compare the size of the value. *The if_icmpge* directive means that if a value is greater than or equal to another value, in this case if *i$ is* greater than or equal to *len$* then execution is performed from the statement marked 43. If the condition is not met, the loop continues to the next iteration.

At the end of the loop, the loop counter is incremented by 1 and the loop jumps back to the position where the loop condition begins.

```
37: iinc          4, 1       // increment i$
40: goto        18         // jump back to the beginning of the loop
```

Arithmetic operations and conversion

As you can see, in **Java bytecode** , there are a series of instructions that can perform arithmetic operations. In fact, a large part of the instruction set is used to represent arithmetic operations. There are various instructions for adding, subtracting, multiplying, dividing, and taking negative integers, long integers, double precisions, and floating point numbers. In addition to this, there are many instructions for converting between different types.

Arithmetic opcodes and their types

Type conversion occurs when, for example, we want to assign an integer value to a long variable.

	add +	sub -	mult. *	divide /	remainder %	negate -()
int	**iadd**	**isub**	**imul**	**idiv**	**irem**	**ineg**
long	**ladd**	**lsub**	**lmul**	**ldiv**	**lrem**	**lneg**
float	**fadd**	**fsub**	**fmul**	**fdiv**	**frem**	**fneg**
double	**dadd**	**dsub**	**dmul**	**ddiv**	**drem**	**dneg**

Type conversion opcodes

In our case, the integer value is passed as a parameter to the submit() method that actually receives the double precision. It can be seen that the type conversion opcode is applied before the method is actually called:

From \ To	int	long	float	double	byte	char	short
int	-	i2l	i2f	i2d	i2b	i2c	i2s
long	l2i	-	l2f	l2d	-	-	-
float	f2i	f2l	-	f2d	-	-	-
double	d2i	d2l	d2f	-	-	-	-

```
31: iload       5
33: i2d
34: invokevirtual #5     // Method algo/MovingAverage.submit:(D)V
```

This means that we push the local variable value into the stack as an integer, and then convert it to a double with the i2d directive so that it can be passed as a parameter.

The only instruction that does not require a value on the stack is the incremental instruction, **iinc** , which directly manipulates the value on the local variable table (LocalVariableTable). All other operations use the stack.

New & <init> & <clinit>

There is a keyword new in Java and a new directive in bytecode instructions. When we create an instance of the MovingAverage class:

```
MovingAverage ma = new MovingAverage();
```

The compiler generates a series of opcodes of the form:

```
0: new #2 // class algo/MovingAverage
3: dup
4: invokespecial #3 // Method algo/MovingAverage."":()V
```

When you see the **new** , **dup,** and **invokespecial** instructions, this usually represents the creation of a class instance!

You may ask, why are three instructions instead of one? The new directive creates the object, but it does not call the constructor, but it does call the invokespecial directive: it calls a special method, which is actually a constructor. Because the constructor call doesn't return a value, the object is initialized after the object calls this method, but the stack is empty at this point, and we can't do anything after the object is initialized. That's why we need to copy references in the stack ahead of time, and we can assign object instances to local variables or fields after the constructor returns. Therefore, the next instruction is usually one of the following instructions:

- **Astore {N}** or **astore_{N}** – assigns a value to a local variable, and {N} is the location of the variable in the local variable table.
- **Putfield** – assign values to instance fields
- **Putstatic** – assign a value to a static variable

Before calling the constructor, there is another similar method called before. It is the static initializer for this class. The class's static initializer is not called directly, but is triggered by the following directives: new, getstatic, putstatic, or invokestatic. That is, if you create an instance of a class, access a static field, or call a static method, the static initializer will be fired.

In fact, there are many ways to trigger a static initializer, see The Java® Language Specification - Java SE 7 Edition

Method call and parameter passing

In the instantiation of the class, we briefly introduce the method call: the method called by the invokespecial instruction will call the constructor. However, there are some instructions that are also used as method calls:

- **Invokestatic** As the name suggests, it calls the static method of the class. Here it is the fastest instructi on to call the method.

- **Invokespecial** As we know, directives are used to call constructors. But it is also used to call private me thods of the same class, as well as methods that the parent class can access.
- **Invokevirtual is** used to call public, protected, and private methods, if the target object of the method i s a concrete type.
- **The invokeinterface is** used to call methods that belong to an interface.

So what is the difference between *invokevirtual* and *invokeinterface* ?

This is indeed a good question. Why do we need both *invokevirtual* and *invokeinterface* , why not use *invokevirtual everywhere* ? The interface method is also a public method! Ok, this is all for the optimization of method calls. First, the method is parsed and then called. For example, with *invokestatic* we know that the specific method is called: it is static and belongs to only one class. With *invokespecial* our options are a limited list, making it easier to choose a resolution strategy, meaning that the runtime can find the method you need faster.

The difference between *invokevirtual* and *invokeinterface* is not so obvious. We provide a very simple explanation of the difference between the two instructions. Imagine a class definition that includes a list of method definitions, all of which are numbered by location. Here is an example: Class A has methods method1 and method2 and a subclass B. Subclass B inherits method1 and overwrites method2, and declares method method3. Notice that method1 and method2 are in the same index subscript position in class A and class B.

```
class A
    1: method1
    2: method2

class B extends A
    1: method1
    2: method2
    3: method3
```

This means that if the runtime wants to call the method method2, it will always be found in position 2. Now, before explaining *invokevirtual* and *invokeinterface* , let class B extend interface X to define a new method methodX :

```
class B extends A implements X
    1: method1
    2: method2
    3: method3
    4: methodX
```

The new method is in subscript 4 and looks the same as method3. However, if there is another class C, the interface is implemented, but it is not the same as the structure of A and B:

```
class C implements  X
    1: methodC
    2: methodX
```

The location of the interface method is not the same as the location in class B, which is why the *invoke interface* is more rigorous at runtime, which means that it has fewer inference assumptions than the *invokeinterface* in the method parsing process .

reference

Reference source:

The Java® Language Specification - Java SE 7 Edition

The Java® Language Specification - Chapter 6. The Java Virtual Machine Instruction Set

2015.01 A Java Programmer's Guide to Byte Code

2012.11 Mastering Java Bytecode at the Core of the JVM

2011.01 Java Bytecode Fundamentals

2001.07 Java bytecode: Understanding bytecode makes you a better programmer

Wiki: Java bytecode

Wiki: Java bytecode instruction listings

End

Category: Java

Richaaaard
Followers - 13
Followers - 82
+Add attention

1 0

« Previous: JVM Internal Principles (4) - JVM Structure of Basic Concepts
» Next: JVM Internal Principles (7) - Java Bytecode Basics 2

Posted @ 2016-12-23 15:25 Richaaaard Reading (1561) Comments (1) Edit Collection

comment list

#1st Floor 2017-06-06 21:55 anbokeyuan

Good posts are a little difficult for me, I am a rookie to study slowly.

Support (0) against (0)

Refresh the comment refresh page back to the top

Registered Users Sign in to post a comment, please log in or register , visit the website home page.

The latest IT news :
· Tencent to the hidden disease: to be a great company rather than a huge game company
· Firefox Nightly to join the encryption SNI support
· Yan Ning in "Science" issued: Analyze the fine structure of the electronically controlled sodium ion channel
· How should NLP learn, How to teach? Interview with Professor Dan Jurafsky of Stanford University
· Ping An and GitHub reached a strategic partnership to become their first Greater China cloud management service provider
» More News...

Latest Knowledge Base Article :
· Alibaba Cloud's madman
· Why do Java programmers have to master Spring Boot?
· In learning, there is a more important ability than mastering knowledge
. How to recruit a reliable programmer
. A story to understand the "blockchain"
» More Knowledge Base Articles...

Today in history:
2015-12-23 JBoss Wildfly (1) —— 7.2.0.Final compilation