JVM Tutorial Part 3

by Ashok Kumar *on* 08:38:00 *in* Core Java, JAVA, JVM

Execution Engine

This is the core of the JVM. Execution engine can communicate with various memory areas of JVM. Each thread of a running Java application is a distinct instance of the virtual machine's execution engine. The byte code that is assigned to the runtime data areas in the JVM via class loader is executed by the execution engine.

The execution engine reads the Java Byte code in the unit of instruction. It is like a CPU executing the machine command one by one. Each command of the byte code consists of a 1-byte OpCode and additional Operand. The execution engine gets one OpCode and execute task with the Operand, and then executes the next OpCode. Execution engine mainly contain 2 parts.

1. Interpreter
2. JIT Compiler

Whenever any java program is executing at the first time interpreter will comes into picture and it converts one by one byte code instruction into machine level instruction. JIT compiler (just in time compiler) will comes into picture from the second time onward if the same java program is executing and it gives the machine level instruction to the process which are available in the buffer memory. The main aim of JIT compiler is to speed up the execution of java program.

1. Interpreter

It is responsible to read byte code and interpret into machine code (native code) and execute that machine code line by line. The problem with interpret is it interprets every time even some method invoked multiple times which effects performance of the system. To overcome this problem SUN people introduced JIT compilers in 1.1 V.

2. JIT Compiler

The JIT compiler has been introduced to compensate for the disadvantages of the interpreter. The main purpose of JIT compiler is to improve the performance. Internally JIT compiler maintains a separate count for every method. Whenever JVM across any method call, first that method will be interpreted normally by the interpreter and JIT compiler increments the corresponding count variable.

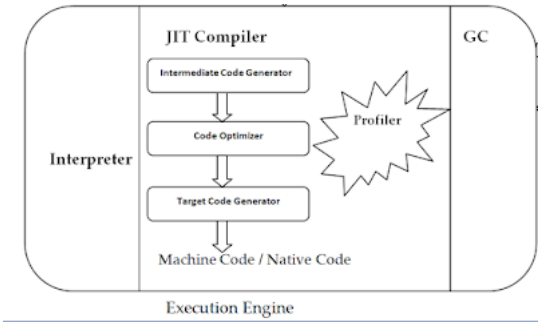
This process will be continued for every method once if any method count reaches thread hold value then JIT compiler identifies that method is a repeatedly used method (Hotspot) immediately JIT compiler compiles that method and generates corresponding native code. Next time JVM come across that method call then JVM directly uses native code and executes it instead of interpreting once again, so that performance of the system will be improved. Threshold is varied from JVM to JVM. Some advanced JIT

compilers will recompile generated native code if count reaches threshold value second time so that more optimized code will be generated.

Profiler which is the part of JIT compiler is responsible to identify **Hotspot** (Repeated Used Methods).

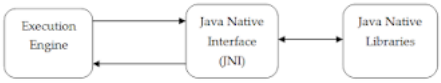
Note

JVM interprets total program line by line at least once. JIT compilation is applicable only for repeatedly invoked method but not for every method.



Java Native Interface (JNI)

JNI is acts as a bridge (Mediator) for java method calls and corresponding native libraries.



Class File Structure

```
ClassFile {
   u4            magic_number;
   u2            minor_version;
   u2            major_version;
   u2            constant_pool_count;
   cp_info       constant_pool[constant_pool_count-1];
   u2            access_flags;
   u2            this_class;
   u2            super_class;
   u2            interfaces_count;
   u2            interfaces[interfaces_count];
   u2            fields_count;
   field_info    fields[fields_count];
   u2            methods_count;
   method_info   methods[methods_count];
   u2            attributes_count;
   attribute_info attributes[attributes_count];
}
```

magic_number

* The first 4 bytes of class file is magic number.

* This is a predefined value to identify the Java class file.

* This value should be 0xCAFEBABE.

Ashok Kumar Apr 26, 2018

Hive Architecture
Ashok Kumar Apr 26, 2018

Apache Hive Introduction
Ashok Kumar Apr 26, 2018

Hadoop Counters
Ashok Kumar Apr 17, 2018

Speculative Execution in Hadoop
Ashok Kumar Apr 16, 2018

TAGS

ADVANCED JAVA (15) BIGDATA (49)
CASSANDRA (4) CORE JAVA (47)
DATABASE (22) DATASTRUCTURES (5)
FRAMEWORKS (19) HADOOP (27)
HBASE (15) HIBERNATE (15) HIVE (2)
JAVA (63) JAVA DESIGN PATTERNS (11)
JAVA REFLECTION (1) JENKINS (4) JSVC
JVM (3) KAFKA (2) LOG4J (1)
NETAPP (2) PL SQL (10) PROTOBUF (2)
REGULAR EXPRESSION (1) SPRING (4)
SQL (12) STORAGE (5) TESTNG (4)
XML (7) ZOOKEEPER (1)

BLOG ARCHIVE

May (21)

* JVM will use this value to identify whether the class file is valid or not and also to know whether the class file is generated by valid compiler or not.

* Whenever we are executing a Java class if JVM unable to find magic_number then we will get runtime error saying java.lang.ClassFormatError : Incompatible magic number

major and minor versions

* major and minor versions represent class file version

* JVM will use these versions to identify which version of compiler generates the current .class file

* If a class file has major version number M and minor version number m, we denote the version of its class file format as M.m.

* Major and minor versions both are allocates 2 bytes

* The possible values are

Code

```
 major  minor   Java platform version
 45       3           1.0
 45       3           1.1
 46       0           1.2
 47       0           1.3
 48       0           1.4
 49       0           1.5
 50       0           1.6
 51       0           1.7
 52       0           1.8
```

E.g

Code

```
package com.ashok.jvm.test;

import java.io.*;

public class ClassVersionChecker {
    private static void checkClassVersion() throws Exception {
        DataInputStream in = new DataInputStream(new FileInputStream("D://Ashok /Te
        int magic = in.readInt();
        if (magic != 0xcafebabe) {
              System.out.println(" It is not a valid class!");
        }
        int minor = in.readUnsignedShort();
        int major = in.readUnsignedShort();
        System.out.println( major + " . " + minor);
        in.close();
   }
   public static void main(String[] args) throws Exception {
        checkClassVersion();
   }
}
```

Note:

Higher version JVM can always run class files generated by lower version compiler but lower version JVM can't run class files generated by higher version compiler. If we are trying to run then we will get run time

exception saying UnsupportedClassVersionError:Test : unsupported major.minor version.

constant_pool_count

The value of the constant_pool_count item is equal to the number of entries in the constant_pool table plus one. The constant pool table is where most of the literal constant values are stored. This includes values such as numbers of all sorts, strings, identifier names, references to classes and methods, and type descriptors.

constant_pool[]

It represents information about constants present in the constant table. The constant_pool is a table of structures (§4.4) representing various string constants, class and interface names, field names, and other constants that are referred to within the ClassFile structure and its substructures. The format of each constant_pool table entry is indicated by its first "tag" byte.

CONSTANT_Class

Tag : 7

Description : The name of a class

CONSTANT_Fieldref

Tag : 9

Description : The name and type of a Field, and the class of which it is a member.

CONSTANT_Methodref

Tag : 10

Description : The name and type of a Method, and the class of which it is a member.

CONSTANT_InterfaceMethodref

Tag : 11

Description : The name and type of a Interface Method, and the Interface of which it is a member.

CONSTANT_String

Tag : 8

Description : The index of a CONSTANT_Utf8 entry.

CONSTANT_Integer

Tag : 3

Description : 4 bytes representing a Java integer.

CONSTANT_Float

Tag : 4

Description : 4 bytes representing a Java float.

CONSTANT_Long

Tag : 5

Description : 8 bytes representing a Java long.

CONSTANT_Double

Tag : 6

Description : 8 bytes representing a Java double.

CONSTANT_NameAndType

Tag : 12

Description : The Name and Type entry for a field, method, or interface.

CONSTANT_Utf8

Tag : 1

Description : 2 bytes for the length, then a string in Utf8 (Unicode) format.

access_flags

Access flags follows the Constant Pool. It is a 2 byte entry that indicates whether the file defines a class or an interface, whether it is public or abstract or final in case it is a class. Below is a list of some of the access flags and their interpretation.

Flag Name	Value	Interpretation
ACC_PUBLIC	0x0001	Declared `public`; may be accessed from outside its package.
ACC_FINAL	0x0010	Declared `final`; no subclasses allowed.
ACC_SUPER	0x0020	Treat superclass methods specially when invoked by the *invokespecial* instruction.
ACC_INTERFACE	0x0200	Is an interface, not a class.
ACC_ABSTRACT	0x0400	Declared `abstract`; must not be instantiated.
ACC_SYNTHETIC	0x1000	Declared synthetic; not present in the source code.
ACC_ANNOTATION	0x2000	Declared as an annotation type.
ACC_ENUM	0x4000	Declared as an `enum` type.

Note

A class may be marked with the ACC_SYNTHETIC flag to indicate that it was generated by a compiler and does not appear in source code.

this_class

Next 2 bytes after access_flags is this_class. It represents the fully qualified name of the current class.

super_class

Next 2 bytes after this_class is super_class. It represents the fully qualified name of the super class.

interfaces_count

Next 2 bytes after super_class is interfaces_count. It represents the number of interfaces implemented by the current class.

interfaces[]

It represents the names of interfaces implemented by the current class.

fields_count

It represents the number of fields present in the current class.

fields[]

It represents field information present in the current class.

methods_count

It represents the number of methods present in the current class.

methods[]

It represents method information present in the current class.

attributes_count

It represents the number of attributes present in the current class.

attributes[]

It represents attribute information present in the current class.

That's it guys. This is all about JVM Tutorial. Let me know your comments and suggestions about this tutorial. Thank you.

Previous Tutorial JVM Tutorial Part 2

Tags # Core Java # JAVA

About Ashok Kumar

I'm Ashok Kumar Java Developer from Bhimavaram, India. I have a Master Degree in Computer Applications from S.R.K.R Engineering College, Bhimavaram. I achieved Gold Medal in my Master Degree. I am a part time blogger. I am passionate to learn the new Technologies. I would like to enhance my skills and share those to others. Every one wants to learn new Technologies in depth but they want to do this by spending more time to get in depth information. My blog may save your time and learn new Technologies in depth and easy way.

Newer Article
Apache Cassandra Tutorial

Older Article
JVM Tutorial Part 2

RELATED POSTS:

JVM Tutorial Part 3 JVM Tutorial Part 2 JVM Tutorial

4 comments:

Abhishek Bandiya
23 September 2017 at 09:41

Nice Tutorial. Thanks.

Reply

Replies

1. 

Ashok Kumar
15 November 2017 at 04:18

Thank you Abhishek

Reply

Atul Jadhav
15 November 2017 at 04:15

very nice tuts.

Reply

Replies

1. 

Ashok Kumar
15 November 2017 at 04:18

Thank you Atul Jadhav

Reply