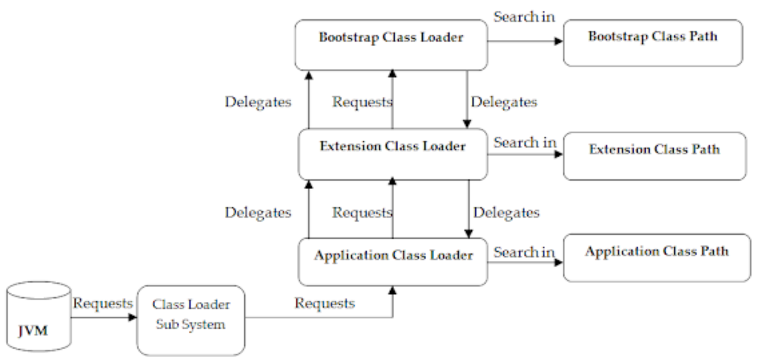
JVM Tutorial Part 2

by Ashok Kumar *on* 08:32:00 *in* Core Java, JAVA, JVM

How Java Class loader works?

Class loader sub system follows delegation hierarchy algorithm. The algorithm simply looks like as following.



JVM execute java program line by line. Whenever JVM come across a particular class first JVM will check weather this .class file is already loaded or not. If it is loaded JVM uses that loaded class from method area otherwise JVM will requests class loader sub system to load the .class file then class loader sub system sends that request to application class loader.

Application class loader won't load that requested class, simply it delegates to extension class loader. Extension class loader also won't load that requested class, simply it delegates to boot strap class loader. Now boot strap class loader search in boot strap class path. If the class is found in boot strap class path then loaded otherwise it delegates to extension class loader.

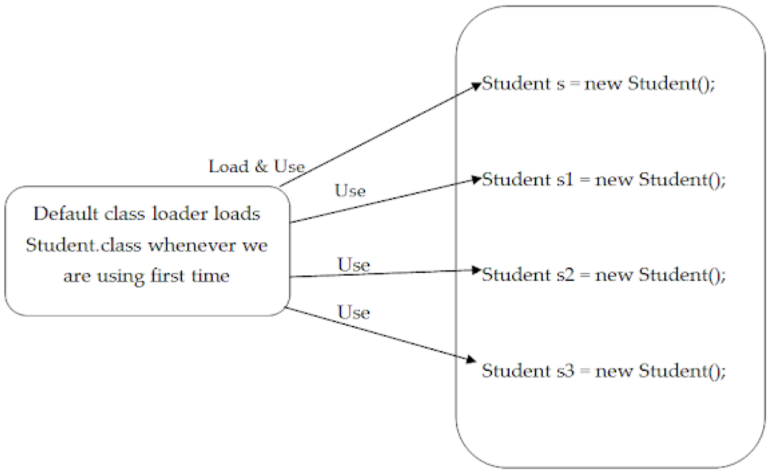
Now extension class loader searches in extension class path. If the class is found in extension class path then loaded otherwise it delegates to application class loader. Application class loader now searches in application class path.

If the class is found in application class path then loaded. Suppose boot strap class loader unable find, extension class loader unable to find, application class loader unable to find then we will get run time exception called "**ClassNotFound**" exception. This is the algorithm that class loader sub system follows. This algorithm is called “Delegation hierarchy algorithm”.

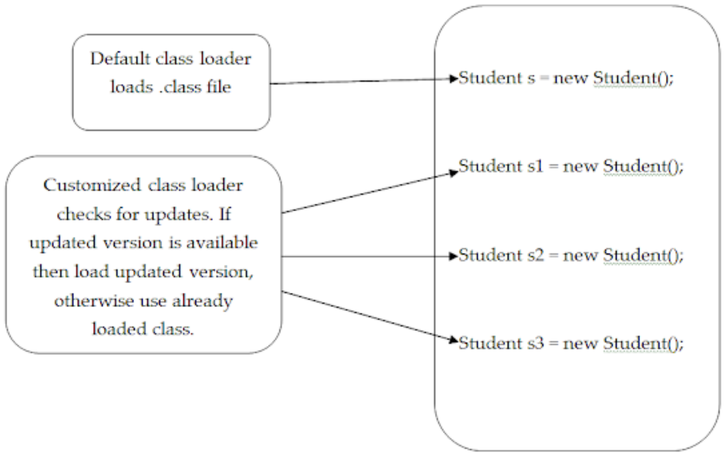
Here the highest priority will be bootstrap class path, if the class not found in bootstrap class path the next level priority is extension class path, if the class not found in extension class path the next level priority is application class path.

Customized class loader

Sometimes we may not satisfy with default class loader mechanism then we can go for Customized class loader. For example


Default class loader loads Student.class whenever we are using first time
Load & Use
Use
Use
Use
Student s = new Student();
Student s1 = new Student();
Student s2 = new Student();
Student s3 = new Student();

Default class loader loads .class file only once even though we are using multiple times that class in our program. After loading .class file if it is modified outside , then default class loader won't load updated version of .class file on fly, because .class file already there in method area. To overcome this problem we are going to customized class loader.


Default class loader loads .class file
Customized class loader checks for updates. If updated version is available then load updated version, otherwise use already loaded class.
Student s = new Student();
Student s1 = new Student();
Student s2 = new Student();
Student s3 = new Student();

Whenever we are using a class, customised class loader checks whether updated version is available or not. If it is available then load updated version otherwise use already loaded existing .class file, so that updated version available to our program.

Code
```
public class CustomizedClassLoader extends ClassLoader {
   public Class loadClass(String cname) throws ClassNotFoundException {
       // Check whether updated version available or not. If updated version is ava
          load updated version and returns corresponding class "Class" object. Othe
          Class object of already loaded .class
```

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

June (18)

```
   }
}
```

Code

```
class CustomClassLoaderTest {
   public static void main( String arg[]) {
      Student s1 = new Student(); // Default class loader loads Student.class
      .
      .
      CustomizedClassLoader c = new CustomizedClassLoader();
      c.loadClass(Student); // Customized class loader checks updates and load upda
      .
      .
      c.loadClass(Student); // Customized class loader checks updates and load upda
   }
}
```

Note

While designing/developing web servers and application servers usually we can go for customized class loaders to customized class loading mechanism.

2. Various Memory Areas in JVM

Whenever a Java virtual machine runs a program, it needs memory to store many things, including byte codes and other information it extracts from loaded class files, objects the program instantiates, parameters to methods, return values, local variables, and intermediate results of computations. The Java virtual machine organizes the memory it needs to execute a program into several runtime data areas.

Various memory areas of JVM are

1. Method Area
2. Heap Area
3. Stack Area
4. PC Registers
5. Native Method Stack

1. Method Area

* For every JVM one method area will be available

* Method area will be created at the time of JVM start up.

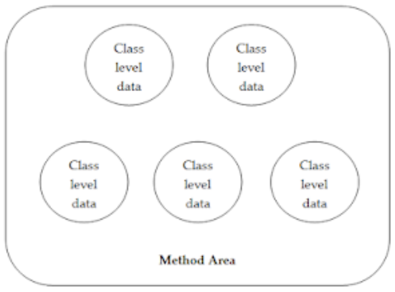
* Inside method area class level binary data including static variables will be stored

* Constant pools of a class will be stored inside method area.

* Method area can be accessed by multiple threads simultaneously.

* The size of the method area need not be fixed. As the Java application runs, the virtual machine can expand and contract the method area to fit the application's needs.

* All threads share the same method area, so access to the method area's data structures must be designed to be thread-safe.



2. Heap Area

* For every JVM one heap area will be available

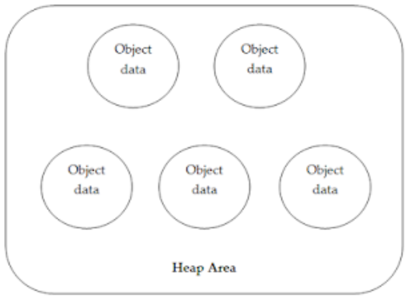
* Heap area will be created at the time of JVM start up.

* Objects and corresponding instance variables will be stored in the heap area.

* Every array in java is object only hence arrays also will be stored in the heap area.

* Heap area can be access by multiple threads and hence the data stored in the heap area is not thread safe.

* Heap area need not be continued.



Display heap memory statistics

A java application can communicate with JVM by using **Runtime** class object. A **Runtime** class is a singleton class and we can create Runtime object by using getRuntime() method.

Code
```
Runtime run = Runtime.getRunner();
```

Once we got runtime object we can call the following methods on that object.

1. maxMemory()

It returns number of bytes of maximum memory allocated to the heap.

2. totalMemory()

It returns number of bytes of total memory allocated to the heap.

3. freeMemory()

It returns number of bytes of free memory present in the heap.

E.g

Code

```
public class HeapSpaceDemo {
   public static void main(String[] args) {
      Runtime runtime = Runtime.getRuntime();
      System.out.println("Maximum memory " +runtime.maxMemory());
      System.out.println("Total memory " +runtime.totalMemory());
      System.out.println("Free memory " +runtime.freeMemory());
   }
}

Output
Maximum memory 889192448
Total memory 60293120
Free memory 58719832
```

Set Maximum and Minimum heap size

Heap memory is a finite memory based on our requirement we can increase or decrease heap size. We can use following options for your requirement

-Xmx

To set maximum heap size , i.e., maxMemory

java -Xmx512m HeapSpaceDemo

Here mx = maximum size

512m = 512 MB

HeapSpaceDemo = Java class name

-Xms

To set minimum heap size , i.e., total memory

java -Xms65m HeapSpaceDemo

Here ms = minimum size

65m = 65 MB

HeapSpaceDemo = Java class name

or, you can set a minimum maximum heap size at a time

java -Xms256m -Xmx1024m HeapSpaceDemo

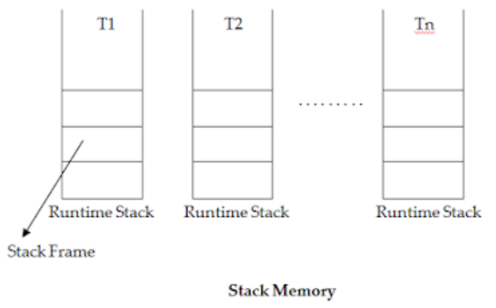
3. Stack Memory

For every thread JVM will create a runtime stack at the time of thread creation. Each and every method call performed by the thread and corresponding local variables will be stored by in the stack,

For every method call a separate entry will be added to the stack and each entry is called **"Stack frame"** or **"activation record".**

After completing the method call the corresponding entry will be removed from the stack. After completing the all method calls the stack will become empty and that empty stack will be destroyed by the JVM just before terminating the thread.

The data stored in the stack is private to the corresponding thread.



Stack Frame Structure

Stack frame contains 3 parts

1. Local Variable Array
2. Operand Stack
3. Frame Data

i. Local Variable Array

* It contains all parameters and local variables of the method.

* Each slot in the array is of 4 bytes.

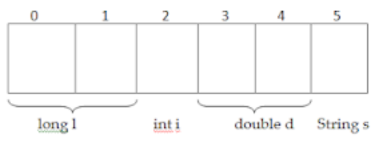
* Values of type int, float and reference occupied 1 slot in array.

* Values of type long, double occupied 2 consecutive entries in the array.

* Values of byte, short, char will be converted to int type before storing and occupy one slot.

* The way of storing boolean type is varied from JVM to JVM, but most of the JVM's follow one slot for boolean values.

```
Eg: public static void m1(long l, int i, double d, String s) {
                ..........................
                ..........................
    }
```



ii. Operand Stack

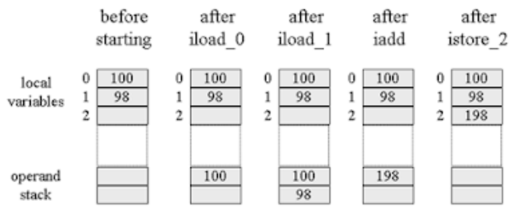
* JVM uses operand stack as work space.

* Some instructions can push the values to the operand stack and some instructions perform required operations and some instructions store results etc.

* The operand stack follows the last-in first-out (LIFO) methodology.

* For example, the iadd instruction adds two integers by popping two ints off the top of the operand stack, adding them, and pushing the int result. Here is how a Java virtual machine would add two local variables that contain ints and store the int result in a third local variable:

```
iload_0    // push the int in local variable 0
iload_1    // push the int in local variable 1
iadd       // pop two ints, add them, push result
istore_2   // pop int, store into local variable 2
```



iii. Frame Data

In addition to the local variables and operand stack, the Java stack frame includes data to support constant pool resolution, all symbolic references related to that method, normal method return, and exception dispatch.

This data is stored in the **frame data** portion of the Java stack frame. It also contains a referenced to exception table which contains corresponding catch block information in the case of exceptions. When a method throws an exception, the Java virtual machine uses the exception table referred to by the frame data to determine how to handle the exception.

Whenever the Java virtual machine encounters any of the instructions that refer to an entry in the constant pool, it uses the frame data's pointer to the constant pool to access that information.

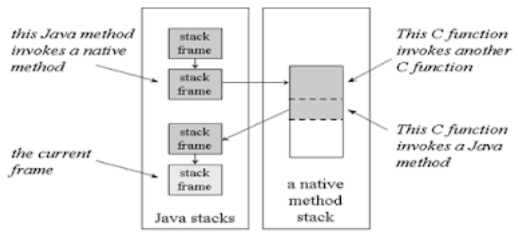
4. PC Registers (Program Counter Registers)

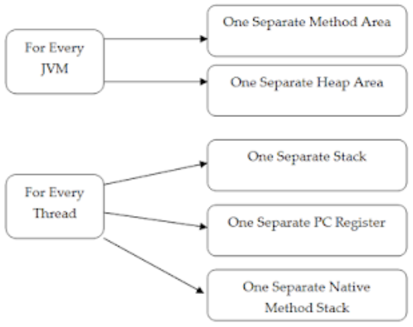
For every thread a separate PC register will be created at the time of thread creation. PC register contains address of current executing instruction. Once instruction execution completes automatically PC register will be incremented to hold address of next instruction. An "address" can be a native pointer or an offset from the beginning of a method's byte codes.

5. Native Method Stacks

Here also for every Thread a separate run time stack will be created. It contains all the native methods used in the application. Native method means methods written in a language other than the Java programming language. In other words, it is a stack used to execute C/C++ codes invoked through JNI (Java Native Interface). According to the language, a C stack or C++ stack is created.

When a thread invokes a Java method, the virtual machine creates a new frame and pushes it onto the Java stack. When a thread invokes a native method, however, that thread leaves the Java stack behind. Instead of pushing a new frame onto the thread's Java stack, the Java virtual machine will simply dynamically link to and directly invoke the native method.


this Java method invokes a native method
stack frame
stack frame
stack frame
stack frame
the current frame
Java stacks
This C function invokes another C function
This C function invokes a Java method
a native method stack



Next Tutorial JVM Tutorial Part 3

Previous Tutorial JVM Tutorial

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 JVM Tutorial Part 3

Older Article Operators And Assignments

RELATED POSTS:

JVM Tutorial Part 3 JVM Tutorial Part 2 JVM Tutorial

No comments:

Post a Comment