JVM Memory Model / Structure and Components

August 28, 2014 by Lokesh Gupta

Whenever you execute a java program, a separate memory area is reserved for storing various parts of your application code which you roughly call **JVM memory**. Though no necessary, but having some knowledge about structuring of this memory area is quire beneficial. It becomes more important when you start working on deeper areas like performance tuning. Without having good understanding of how JVM actually consume the memory and how garbage collector uses different parts of this memory, you may miss some important considerations for better memory management; thus better performance.

In this tutorial, I am discussing the various parts inside **JVM memory**, you should be aware of, and then in one of my future post we will discuss about how to use this information for performance tuning of your application.

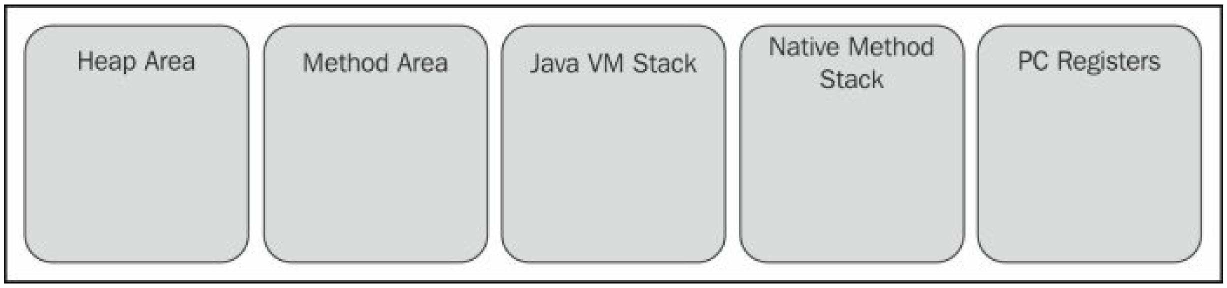
Table of Contents

JVM Memory Model / Structure

The Java Virtual Machine defines various **run-time data areas** that are used during execution of a program. Some of these data areas are created on Java Virtual Machine start-up and are destroyed only when the Java

Virtual Machine exits. Other data areas are per thread. Per-thread data areas are created when a thread is created and destroyed when the thread exits.

Let's look at the most basic categorization of various parts inside runtime memory.



JVM Memory Area Parts

Let's have a quick look at each of these components according to what is mentioned in the JVM specifications.

Heap area

The heap area represents the runtime data area, from which the memory is allocated for all class instances and arrays, and is created during the virtual machine startup.

The heap storage for objects is reclaimed by an automatic storage management system. The heap may be of a fixed or dynamic size (based on system's configuration), and the memory allocated for the heap area does not need to be contiguous.

A Java Virtual Machine implementation may provide the programmer or the user control over the initial size of the heap, as well as, if the heap can be dynamically expanded or contracted, control over the maximum and minimum heap size.

> If a computation requires more heap than can be made available by the automatic storage management system, the Java Virtual Machine throws an `OutOfMemoryError` .

Method area and runtime constant pool

Method area stores per-class structures such as the runtime constant pool; field and method data; the code for methods and constructors, including the special methods used in class, instance, and interface initialization.

The method area is created on the virtual machine startup. Although it is logically a part of the heap but it can or cannot be garbage collected, whereas we already read that garbage collection in heap is not optional; it's mandatory. The method area may be of a fixed size or may be expanded as required by the computation and may be contracted if a larger method area becomes unnecessary. The memory for the method area does not need to be contiguous.

> If memory in the method area cannot be made available to satisfy an allocation request, the Java Virtual Machine throws an `OutOfMemoryError`.

JVM Stacks

Each of the JVM threads has a private stack created at the same time as that of the thread. The stack stores frames. A frame is used to store data and partial results and to perform dynamic linking, return values for methods, and dispatch exceptions.

It holds local variables and partial results and plays a part in the method invocation and return. Because this stack is never manipulated directly, except to push and pop frames, the frames may be heap allocated. Similar to the heap, the memory for this stack does not need to be contiguous.

This specification permits that stacks can be either of a fixed or dynamic size. If it is of a fixed size, the size of each stack may be chosen independently when that stack is created.

> If the computation in a thread requires a larger Java Virtual Machine stack than is permitted, the Java Virtual Machine throws a `StackOverflowError`.

> If Java Virtual Machine stacks can be dynamically expanded, and expansion is attempted but insufficient memory can be made available to effect the expansion, or if insufficient memory can be made available to create the initial Java Virtual Machine stack for a new thread, the Java Virtual Machine throws an `OutOfMemoryError`.

Native method stacks

Native method stacks is called C stacks; it support native methods (methods written in a language other than the Java programming language), typically allocated per each thread when each thread is created. Java Virtual Machine implementations that cannot load native methods and that do not themselves rely on conventional stacks need not supply native method stacks.

The size of native method stacks can be either fixed or dynamic.

> If the computation in a thread requires a larger native method stack than is permitted, the Java Virtual Machine throws a `StackOverflowError`.

> If native method stacks can be dynamically expanded and native method stack expansion is attempted but insufficient memory can be made available, or if insufficient memory can be made available to create the initial native method stack for a new thread, the Java Virtual Machine throws an `OutOfMemoryError`.

PC registers

Each of the JVM threads has its own program counter (pc) register. At any point, each of the JVM threads is executing the code of a single method, namely the current method for that thread.

As the Java applications can contain some native code (for example, using native libraries), we have two different ways for native and non-native methods. If the method is not native (that is, a Java code), the PC register contains the address of the JVM instruction currently being executed. If the method is native, the value of the JVM's PC register is undefined.

The Java Virtual Machine's pc register is wide enough to hold a return address or a native pointer on the specific platform.

That's all for now related to memory area structuring inside JVM. I will come up with ideas to use this information for performance tuning in coming posts.

Happy Learning !!

Reference: http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html