Java Performance - Memory and Runtime Analysis - Tutorial

Lars Vogel

Version 0.8

12.12.2010

Revision History		
Revision 0.1	28.12.2008	Lars Vogel
created		
Revision 0.2 -0.8	12.01.2009 - 12.12.2010	Lars Vogel
bug fixes and enhancements		

Abstract

This article will be a collection of Java performance measurement pointer.

It describes how memory works in general and how Java use the heap and the stack. The article describes how to set the available memory for Java. It discusses then how to get the runtime and the memory consumption of a Java application.

Table of Contents

1. Performance factors

Important influence factors to the performance of a Java program can be separated into two main parts:

- Memory Consumption of the Java program
- Total runtime of a program

In case the program is to some case a program which interacts with others also the response time is a very important fact of the performance of a program.

A average CPU can do approximately 1 billion (10^9) operations per second.

This article does not cover concurrency. If you want to read about concurrency / multithreading please seeConcurrency / Multithreading in Java

2. Memory

Java handles its memory in two areas. The heap and the stack. We will start with a short overview of memory in general on a computer. Then the Java heap and stack is explained.

2.1. Native Memory

Native memory is the memory which is available to a process, e.g. the Java process. Native memory is controlled by the operating system (OS) and based on physical memory and other physical devices, e.g. disks, flash memory, etc.

The processor (CPU) of the computer computes the instructions to execute and stores its computation results into registers. These registers are fast memory elements which stores the result of the CPU. The processor can access the normal memory over the memory bus. A amount of memory a CPU can access is based on the size of the physical address which the CPU uses to identify physical memory. A 16-bit address can access 2^16 (=65.536) memory locations. A 32-bit address can access 2^32 (=4.294.967.296) memory locations. If each memory area consists of 8 bytes then a 16-bit system can access 64KB of memory and the 32-bit system can access 4GB of memory.

An OS normally uses virtual memory to map the physical memory to memory which each process can see. The OS assigns then memory to each process in a virtual memory space for this process and maps access to this virtual memory to the real physical memory.

Current 32-bit systems uses an extension (Physical Address Extension (PAE)) which extends the physical space to 36-bits of the operation system. This allows the OS to access 64GB. The OS uses then virtual memory to allow the individual process 4 GB of memory. Even with PAE enabled a process can not access more then 4 GB of memory.

Of course with a 64-bit OS this 4GB limitation does not exists any more.

2.2. Memory in Java

Java manages the memory for use. New objects created and placed in the heap. Once your application have no reference anymore to an objects the Java garbage collector is allowed to delete this object and remove the memory so that your application can use this memory again.

2.3. Java Heap

In the heap the Java Virtual Machine (JVM) stores all objects created by the Java application, e.g. by using the "new" operator. The Java garbage collector (gc) can logically separate the heap into different areas, so that the gc can faster identify objects which can get removed

The memory for new objects is allocated on the heap at run time. Instance variables live inside the object in which they are declared.

2.4. Java Stack

Stack is where the method invocations and the local variables are stored. If a method is called then its stack frame is put onto the top of the call stack. The stack frame holds the state of the method including which line of code is executing and the values of all local variables. The method at the top of the stack is always the current running method for that stack. Threads have their own call stack.

2.5. Escape Analysis

As said earlier in Java objects are created in the heap. The programming language does not offer the possibility to let the programmer decide if an objects should be generated in the stack. But in certain cases it would be desirable to allocate an object on the stack, as the memory allocation on the stack is cheaper then the memory allocation in the heap, deallocation on the stack is free and the stack is efficiently managed by the runtime.

The JVM uses therefore internally escape analysis to check if an object is used only with a thread or method. If the JVM identify this it may decide to create the object on the stack, increasing performance of the Java program.

3. Garbage Collector

The JVM automatically re-collects the memory which is not used any more. The memory for objects which are not referred any more will be automatically released by the garbage collector.

To see then the garbage collector starts working add the command line argument "-verbose:gc" to your virtual machine.

An in-depth article about the garbage collector can be found here: Tuning Garbage Collection with the 5.0 Java Virtual Machine

4. Memory settings for Java virtual machine

The JVM runs with fixed available memory. Once this memory is exceed you will receive "java.lang.OutOfMemoryError". The JVM tries to make an intelligent choice about the available memory at startup (see Java settings for details) but you can overwrite the default with the following settings.

To turn performance you can use certain parameters in the JVM.

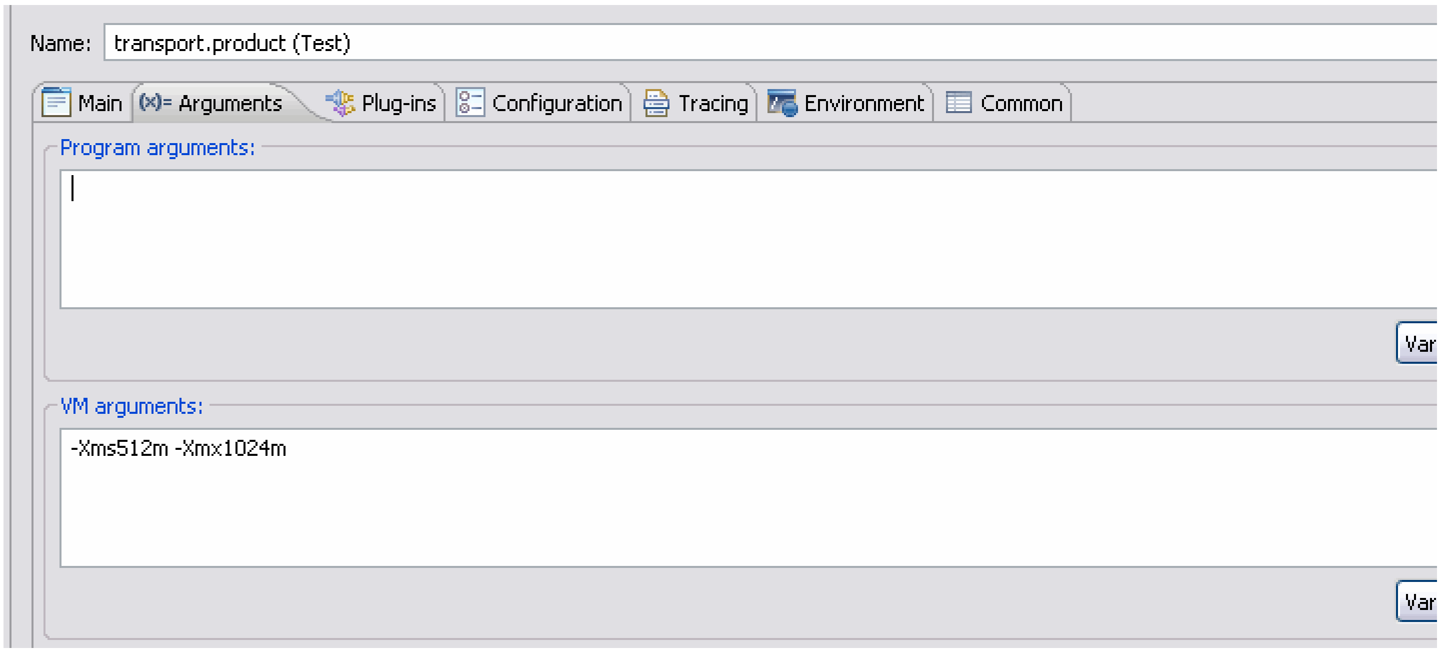
Table 1.

Parameter	Description
-Xms1024m	Set the minimum available memory for the JVM to 1024 Megabyte
-Xmx1800m	Set the maximum available memory for the JVM to 1800 Megabyte. The Java application cannot use more hea then defined via this parameter.

Increase the values in these parameters to avoid the following error: "Exception in thread java.lang.OutOfMemoryError: Java heap space". Note that you cannot allocate more memory then you have physically available.

If you start your Java program from the command line use for example the following setting: java -Xmx1024m YourProgram.

In Eclipse your can use the VM arguments in the run configuration.


Name: transport.product (Test)

```
-Xms512m -Xmx1024m
```

5. Memory Consumption and Runtime

In general a operation is considered as extensive if this operation has a long runtime or a high memory consumption.

5.1. Memory Consumption

The total used / free memory of an program can be obtained in the program via java.lang.Runtime.getRuntime();

The runtime has several method which relates to the memory. The following coding example demonstrate its usage.

```
package test;


import java.util.ArrayList;

import java.util.List;


public class PerformanceTest {

        private static final long MEGABYTE = 1024L * 1024L;


        public static long bytesToMegabytes(long bytes) {

                return bytes / MEGABYTE;
        }


        public static void main(String[] args) {

                // I assume you will know how to create a object Person
yourself...

                List<Person> list = new ArrayList<Person>();

                for (int i = 0; i <= 100000; i++) {

                        list.add(new Person("Jim", "Knopf"));

                }

                // Get the Java runtime

                Runtime runtime = Runtime.getRuntime();

                // Run the garbage collector

                runtime.gc();

                // Calculate the used memory

                long memory = runtime.totalMemory() - runtime.freeMemory();

                System.out.println("Used memory is bytes: " + memory);

                System.out.println("Used memory is megabytes: "

                                + bytesToMegabytes(memory));

        }

}
```

5.2. Runtime of a Java program

Use System.currentTimeMillis() to get the start time and the end time and calculate the difference.

```
package de.vogella.performance.test;


class TimeTest1 {
        public static void main(String[] args) {


                long startTime = System.currentTimeMillis();


                long total = 0;

                for (int i = 0; i < 10000000; i++) {

                        total += i;

                }


                long stopTime = System.currentTimeMillis();

                long elapsedTime = stopTime - startTime;

                System.out.println(elapsedTime);

        }

}
```

6. Lazy initialization

In case a variable is very expensive to create then sometimes it is good to defer the creation of this variable until the variable is needed. This is called lazy initialization.

In general lazy initialization should only be used if a analysis has proven that this is really a very expensive operations. This is based on the fact that lazy initialization makes it more difficult to read the code.

I use the project "de.vogella.performance.lazyinitialization" for the examples in this chapter. And a have a own field defined."

```
package de.vogella.performance.lazyinitialization;



public class MyField {



}
```

6.1. Concurrency - Overview

The simplest way is to use a synchronized block. As then field access is always synchronized in case on read access this variant is slow.

```
package de.vogella.performance.lazyinitialization;


public class SynchronizedTest {

        private MyField myField;


        public synchronized MyField getMyField() {

                if (myField == null) {

                        myField = new MyField();

                }

                return myField;

        }


}
```

6.2. Double-Check Item

Faster is to use the synchronized block only if the variable is still initial and to define the variable as volatile to propergate the change to other threads.

```
package de.vogella.performance.lazyinitialization;


public class DoubleCheckItem {

        private volatile MyField myField;
        public MyField getMyField() {
                if (myField == null) {
                        synchronized (this) {
                                if (myField == null) {
                                        myField = new MyField();
                                }
                        }
                }
                return myField;
        }
}
```

An improved version of this came from Joshua Bloch.

```
package de.vogella.performance.lazyinitialization;

public class DoubleCheckItemImproved {
        private volatile MyField myField;

        public MyField getMyField() {
                MyField tmp = myField;
                if (tmp == null) {
                        synchronized (this) {
                                tmp = myField;
                                if (tmp == null) {
                                        myField = tmp = new MyField();
                                }
                        }

                }

                return tmp; // Using tmp here instead of myField avoids an memory
update

        }

}
```

Avoid lazy initialization if you can, the code becomes difficult to read if you use it.

7. Just-in-time (JIT) compiler

The Java JIT compiler compiles Java byte-code to native executable code during the runtime of your program. This increases the runtime of your program significantly. The JIT compiler uses runtime information to identify part in your application which are runtime intensive. These so-called "hot spots" are then translated native code. This is the reason why the JIT compiler is also called "Hot-spot" compiler.

JIT is store the original byte-code and the native code in memory because JIT can also decide that a certain compilation steps must be revised.

8. Profiler

A profiler for Java is a program which allows you to trace a running java program and see the memory and CPU consumption for the Java application.

Visualvm is already part of the jdk distribution (as of Update 7 for jdk1.6). https://visualvm.dev.java.net/. To start visualvm just click on jvisualvm.exe in the jdk bin directory.

Netbeans seems to have an integrated profiler, but I have not used it so far.

A good commercial seems to be this one looks nice: http://www.yourkit.com

9. Load Test

A load test tool is a tool which emulates user and system actions on an application to measure the reaction and behavior of this system. A load test tool is commonly used for a web application to measure its behavior.

Popular tools for load testing are:

- Apache JMeter - See http://jakarta.apache.org/jmeter/
- Eclipse TPTP testing tool - See http://www.eclipse.org/tptp/
- Grinder - See http://grinder.sourceforge.net/

10. Thank you

Please help me to support this article:

11. Questions and Discussion

Before posting questions, please see the vogella FAQ. If you have questions or find an error in this article please use the www.vogella.de Google Group. I have created a short list how to create good questions which might also help you.

12. Links and Literature

12.1. Performance

https://www.ibm.com/developerworks/java/library/j-jtp09196/ Java theory and practice: Instrumenting applications with JMX

http://www.ibm.com/developerworks/java/library/j-jtp11137.html Java theory and practice: Stick a fork in it, Part 1

http://www.ibm.com/developerworks/java/library/j-jtp03048.html Java theory and practice: Stick a fork in it, Part 2