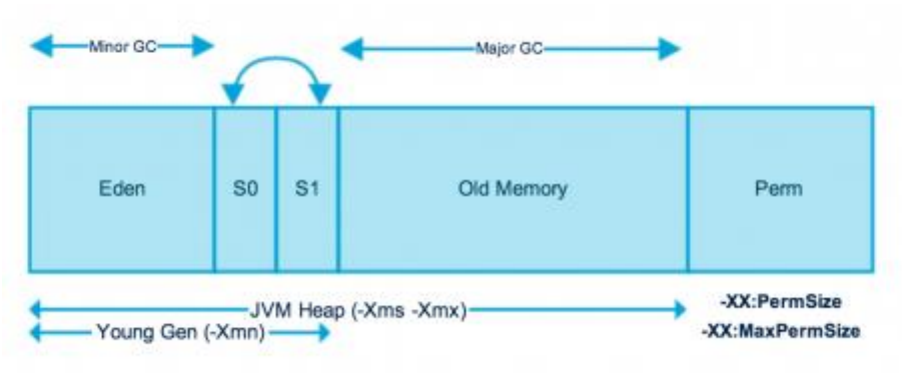
Java (JVM) Memory Model – Memory Management in Java

JUNE 11, 2016 BY PANKAJ 49 COMMENTS

http://www.journaldev.com/2856/java-jvm-memory-model-memory-management-in-java

Understanding **JVM Memory Model**, **Java Memory Management** are very important if you want to understand the working of **Java Garbage Collection**. Today we will look into memory management in java, different parts of JVM memory and how to monitor and perform garbage collection tuning.

Java (JVM) Memory Model



As you can see in the above image, JVM memory is divided into separate parts. At broad level, JVM Heap memory is physically divided into two parts – **Young Generation** and **Old Generation**.

Memory Management in Java – Young Generation

Young generation is the place where all the new objects are created. When young generation is filled, garbage collection is performed. This garbage collection is called **Minor GC**. Young Generation is divided into three parts – **Eden Memory** and two **Survivor Memory** spaces.

Important Points about Young Generation Spaces:

- Most of the newly created objects are located in the Eden memory space.
- When Eden space is filled with objects, Minor GC is performed and all the survivor objects are moved to one of the survivor spaces.
- Minor GC also checks the survivor objects and move them to the other survivor space. So at a time, one of the survivor space is always empty.
- Objects that are survived after many cycles of GC, are moved to the Old generation memory space. Usually it's done by setting a threshold for the age of the young generation objects before they become eligible to promote to Old generation.

Memory Management in Java – Old Generation

Old Generation memory contains the objects that are long lived and survived after many rounds of Minor GC. Usually garbage collection is performed in Old Generation memory when it's full. Old Generation Garbage Collection is called **Major GC** and usually takes longer time.

Stop the World Event

All the Garbage Collections are "Stop the World" events because all application threads are stopped until the operation completes.

Since Young generation keeps short-lived objects, Minor GC is very fast and the application doesn't get affected by this.

However Major GC takes longer time because it checks all the live objects. Major GC should be minimized because it will make your application unresponsive for the garbage collection duration. So if you have a responsive application and there are a lot of Major Garbage Collection happening, you will notice timeout errors.

The duration taken by garbage collector depends on the strategy used for garbage collection. That’s why it’s necessary to monitor and tune the garbage collector to avoid timeouts in the highly responsive applications.

Java Memory Model – Permanent Generation

Permanent Generation or “Perm Gen” contains the application metadata required by the JVM to describe the classes and methods used in the application. Note that Perm Gen is not part of Java Heap memory.

Perm Gen is populated by JVM at runtime based on the classes used by the application. Perm Gen also contains Java SE library classes and methods. Perm Gen objects are garbage collected in a full garbage collection.

Java Memory Model – Method Area

Method Area is part of space in the Perm Gen and used to store class structure (runtime constants and static variables) and code for methods and constructors.

Java Memory Model – Memory Pool

Memory Pools are created by JVM memory managers to create a pool of immutable objects, if implementation supports it. String Pool is a good example of this kind of memory pool. Memory Pool can belong to Heap or Perm Gen, depending on the JVM memory manager implementation.

Java Memory Model – Runtime Constant Pool

Runtime constant pool is per-class runtime representation of constant pool in a class. It contains class runtime constants and static methods. Runtime constant pool is the part of method area.

Java Memory Model – Java Stack Memory

Java Stack memory is used for execution of a thread. They contain method specific values that are short-lived and references to other objects in the heap that are getting referred from the method. You should readDifference between Stack and Heap Memory.

Memory Management in Java – Java Heap Memory Switches

Java provides a lot of memory switches that we can use to set the memory sizes and their ratios. Some of the commonly used memory switches are:

VM SWITCH	VM SWITCH DESCRIPTION
-Xms	For setting the initial heap size when JVM starts
-Xmx	For setting the maximum heap size.
-Xmn	For setting the size of the Young Generation, rest of the space goes for Old Generation.
-XX:PermGen	For setting the initial size of the Permanent Generation memory

-XX:MaxPermGen	For setting the maximum size of Perm Gen
-XX:SurvivorRatio	For providing ratio of Eden space and Survivor Space, for example if Young Generation size is 10m and VM switch is -XX:SurvivorRatio=2 then 5m will be reserved for Eden Space and 2.5m each for both the Survivor spaces. The default value is 8.
-XX:NewRatio	For providing ratio of old/new generation sizes. The default value is 2.

Most of the times, above options are sufficient, but if you want to check out other options too then please check JVM Options Official Page.

Memory Management in Java – Java Garbage Collection

Java Garbage Collection is the process to identify and remove the unused objects from the memory and free space to be allocated to objects created in the future processing. One of the best feature of java programming language is the **automatic garbage collection**, unlike other programming languages such as C where memory allocation and deallocation is a manual process.

Garbage Collector is the program running in the background that looks into all the objects in the memory and find out objects that are not referenced by any part of the program. All these unreferenced objects are deleted and space is reclaimed for allocation to other objects.

One of the basic way of garbage collection involves three steps:

1. **Marking**: This is the first step where garbage collector identifies which objects are in use and which ones are not in use.
2. **Normal Deletion**: Garbage Collector removes the unused objects and reclaim the free space to be allocated to other objects.
3. **Deletion with Compacting**: For better performance, after deleting unused objects, all the survived objects can be moved to be together. This will increase the performance of allocation of memory to newer objects.

There are two problems with simple mark and delete approach.

1. First one is that it's not efficient because most of the newly created objects will become unused
2. Secondly objects that are in-use for multiple garbage collection cycle are most likely to be in-use for future cycles too.

The above shortcomings with the simple approach is the reason that **Java Garbage Collection is Generational** and we have **Young Generation** and **Old Generation** spaces in the heap memory. I have already explained above how objects are scanned and moved from one generational space to another based on the Minor GC and Major GC.

Memory Management in Java – Java Garbage Collection Types

There are five types of garbage collection types that we can use in our applications. We just need to use JVM switch to enable the garbage collection strategy for the application. Let's look at each of them one by one.

1. **Serial GC (-XX:+UseSerialGC)**: Serial GC uses the simple **mark-sweep-compact** approach for young and old generations garbage collection i.e Minor and Major GC.

Serial GC is useful in client-machines such as our simple stand alone applications and machines with smaller CPU. It is good for small applications with low memory footprint.

2. **Parallel GC (-XX:+UseParallelGC)**: Parallel GC is same as Serial GC except that is spawns N threads for young generation garbage collection where N is the number of CPU cores in the system. We can control the number of threads using `-XX:ParallelGCThreads=n` JVM option.

   Parallel Garbage Collector is also called throughput collector because it uses multiple CPUs to speed up the GC performance. Parallel GC uses single thread for Old Generation garbage collection.

3. **Parallel Old GC (-XX:+UseParallelOldGC)**: This is same as Parallel GC except that it uses multiple threads for both Young Generation and Old Generation garbage collection.
4. **Concurrent Mark Sweep (CMS) Collector (-XX:+UseConcMarkSweepGC)**: CMS Collector is also referred as concurrent low pause collector. It does the garbage collection for Old generation. CMS collector tries to minimize the pauses due to garbage collection by doing most of the garbage collection work concurrently with the application threads.

   CMS collector on young generation uses the same algorithm as that of the parallel collector. This garbage collector is suitable for responsive applications where we can't afford longer pause times. We can limit the number of threads in CMS collector using `-XX:ParallelCMSThreads=n` JVM option.

5. **G1 Garbage Collector (-XX:+UseG1GC)**: The Garbage First or G1 garbage collector is available from Java 7 and it's long term goal is to replace the CMS collector. The G1 collector is a parallel, concurrent, and incrementally compacting low-pause garbage collector.

   Garbage First Collector doesn't work like other collectors and there is no concept of Young and Old generation space. It divides the heap space into multiple equalsized heap regions. When a garbage collection is invoked, it first collects the region with lesser live data, hence “Garbage First”. You can find more details about it at Garbage-First Collector Oracle Documentation.

Memory Management in Java – Java Garbage Collection Monitoring

We can use Java command line as well as UI tools for monitoring garbage collection activities of an application. For my example, I am using one of the demo application provided by Java SE downloads.

If you want to use the same application, go to Java SE Downloads page and download **JDK 7 and JavaFX Demos and Samples**. The sample application I am using is **Java2Demo.jar** and it’s present in`jdk1.7.0_55/demo/jfc/Java2D` directory. However this is an optional step and you can run the GC monitoring commands for any java application.

Command used by me to start the demo application is:

```
pankaj@Pankaj:~/Downloads/jdk1.7.0_55/demo/jfc/Java2D$ java -
Xmx120m -Xms30m -Xmn10m -XX:PermSize=20m -XX:MaxPermSize=20m -
XX:+UseSerialGC -jar Java2Demo.jar
```

jsat

We can use `jstat` command line tool to monitor the JVM memory and garbage collection activities. It ships with standard JDK, so you don’t need to do anything else to get it.

For executing `jstat` you need to know the process id of the application, you can get it easily using `ps -eaf | grep java` command.

```
pankaj@Pankaj:~$ ps -eaf | grep Java2Demo.jar
```

```
  501 9582  11579   0  9:48PM ttys000    0:21.66 /usr/bin/java
-Xmx120m -Xms30m -Xmn10m -XX:PermSize=20m -XX:MaxPermSize=20m -
XX:+UseG1GC -jar Java2Demo.jar

  501 14073 14045   0  9:48PM ttys002    0:00.00 grep
Java2Demo.jar
```

So the process id for my java application is 9582. Now we can run **jstat** command as shown below.

```
pankaj@Pankaj:~$ jstat -gc 9582 1000

 S0C    S1C    S0U    S1U      EC       EU        OC         OU
PC     PU    YGC     YGCT    FGC    FGCT     GCT

1024.0 1024.0  0.0    0.0    8192.0   7933.3   42108.0
23401.3   20480.0 19990.9    157    0.274  40      1.381
1.654

1024.0 1024.0  0.0    0.0    8192.0   8026.5   42108.0
23401.3   20480.0 19990.9    157    0.274  40      1.381
1.654

1024.0 1024.0  0.0    0.0    8192.0   8030.0   42108.0
23401.3   20480.0 19990.9    157    0.274  40      1.381
1.654

1024.0 1024.0  0.0    0.0    8192.0   8122.2   42108.0
23401.3   20480.0 19990.9    157    0.274  40      1.381
1.654

1024.0 1024.0  0.0    0.0    8192.0   8171.2   42108.0
23401.3   20480.0 19990.9    157    0.274  40      1.381
1.654
```

```
1024.0 1024.0  48.7   0.0    8192.0   106.7    42108.0
23401.3   20480.0 19990.9    158    0.275  40      1.381
1.656

1024.0 1024.0  48.7   0.0    8192.0   145.8    42108.0
23401.3   20480.0 19990.9    158    0.275  40      1.381
1.656
```

The last argument for jstat is the time interval between each output, so it will print memory and garbage collection data every 1 second.

Let's go through each of the columns one by one.

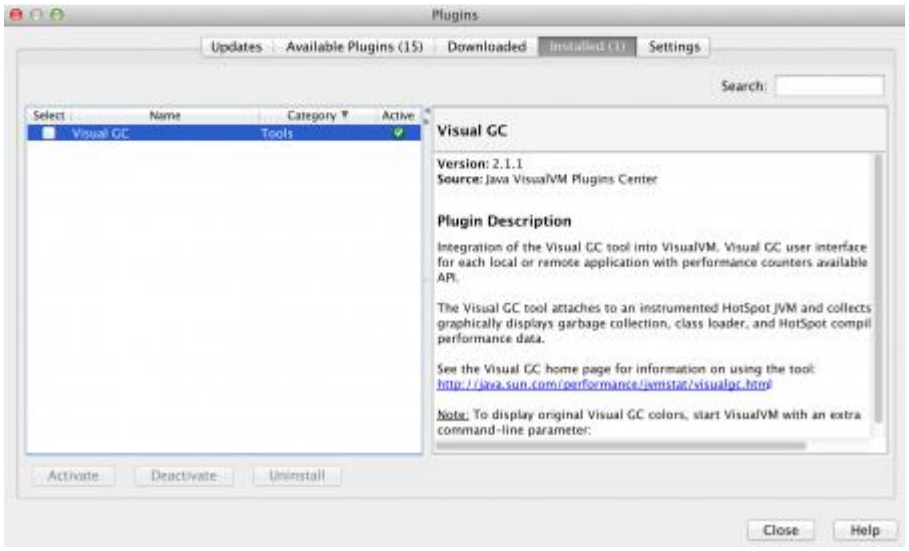
- **S0C and S1C**: This column shows the current size of the Survivor0 and Survivor1 areas in KB.
- **S0U and S1U**: This column shows the current usage of the Survivor0 and Survivor1 areas in KB. Notice that one of the survivor areas are empty all the time.
- **EC and EU**: These columns show the current size and usage of Eden space in KB. Note that EU size is increasing and as soon as it crosses the EC, Minor GC is called and EU size is decreased.
- **OC and OU**: These columns show the current size and current usage of Old generation in KB.
- **PC and PU**: These columns show the current size and current usage of Perm Gen in KB.
- **YGC and YGCT**: YGC column displays the number of GC event occurred in young generation. YGCT column displays the accumulated time for GC operations for Young generation. Notice that both of them are increased in the same row where EU value is dropped because of minor GC.
- **FGC and FGCT**: FGC column displays the number of Full GC event occurred. FGCT column displays the accumulated time for Full GC operations. Notice that Full GC time is too high when compared to young generation GC timings.
- **GCT**: This column displays the total accumulated time for GC operations. Notice that it's sum of YGCT and FGCT column values.

The advantage of **jstat** is that it can be executed in remote servers too where we don't have GUI. Notice that sum of S0C, S1C and EC is 10m as specified through `–Xmn10m` JVM option.

Java VisualVM with Visual GC

If you want to see memory and GC operations in GUI, then you can use `jvisualvm` tool. Java VisualVM is also part of JDK, so you don't need to download it separately.

Just run `jvisualvm` command in the terminal to launch the Java VisualVM application. Once launched, you need to install **Visual GC** plugin from Tools -< Plugins option, as shown in below image.



Visual GC

Version: 2.1.1
Source: Java VisualVM Plugins Center

Plugin Description

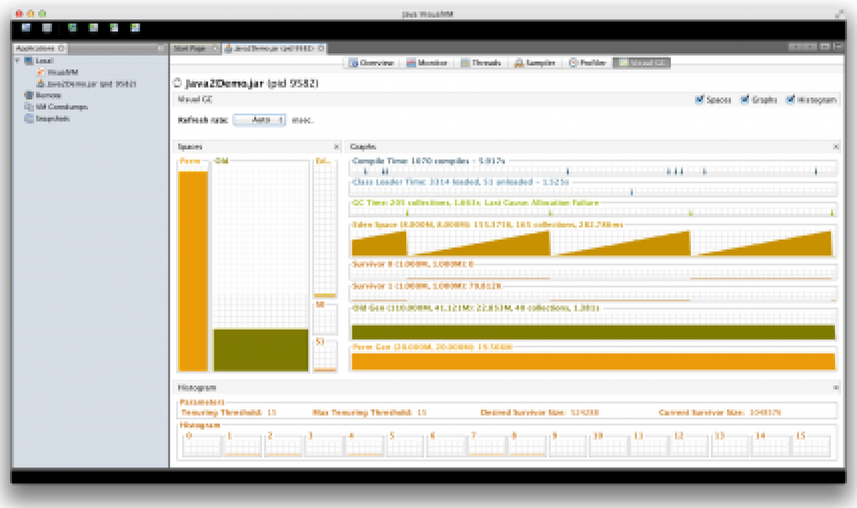
Integration of the Visual GC tool into VisualVM. Visual GC user interface for each local or remote application with performance counters available API.

The Visual GC tool attaches to an instrumented HotSpot JVM and collects graphically displays garbage collection, class loader, and HotSpot compil performance data.

See the Visual GC home page for information on using the tool: http://java.sun.com/performance/jvmstat/visualgc.html

Note: To display original Visual GC colors, start VisualVM with an extra command-line parameter:

After installing **Visual GC**, just open the application from the left side column and head over to **Visual GC**section. You will get an image of JVM memory and garbage collection details as shown in below image.



Java Garbage Collection Tuning

Java Garbage Collection Tuning should be the last option you should use for increasing the throughput of your application and only when you see drop in performance because of longer GC timings causing application timeout.

If you see `java.lang.OutOfMemoryError: PermGen space` errors in logs, then try to monitor and increase the Perm Gen memory space using -XX:PermGen and -XX:MaxPermGen JVM options. You might also try using `-XX:+CMSClassUnloadingEnabled` and check how it's performing with CMS Garbage collector.

If you are see a lot of Full GC operations, then you should try increasing Old generation memory space.

Overall garbage collection tuning takes a lot of effort and time and there is no hard and fast rule for that. You would need to try different options and compare them to find out the best one suitable for your application.

That's all for Java Memory Model, Memory Management in Java and Garbage Collection, I hope it helps you in understanding JVM memory and garbage collection process.