Understanding Java Garbage Collection | CUBRID Blog

What are the benefits of knowing how garbage collection (GC) works in Java? Satisfying the intellectual curiosity as a software engineer would be a valid cause, but also, understanding how GC works can help you write much better Java applications.

This is a very personal and subjective opinion of mine, but I believe that a person well versed in GC tends to be a better Java developer. If you are interested in the GC process, that means you have experience in developing applications of certain size. If you have thought carefully about choosing the right GC algorithm, that means you completely understand the features of the application you have developed. Of course, this may not be common standards for a good developer. However, few would object when I say that understanding GC is a requirement for being a great Java developer.

This is the first of a series of "*Become a Java GC Expert*" articles. I will cover the *GC introduction* this time, and in the next article, I will talk about analyzing GC status and GC tuning examples from NHN.

The purpose of this article is to introduce GC to you in an easy way. I hope this article proves to be very helpful. Actually, my colleagues have already published a few great articles on Java Internals which became quite popular on Twitter. You may refer to them as well.

Returning back to Garbage Collection, there is a term that you should know before learning about GC. The term is "**stop-the-world**." Stop-the-world will occur no matter which GC algorithm you choose. *Stop-the-world* means that the JVM is stopping the application from running to execute a GC. When stop-the-world occurs, every thread except for the threads needed for the GC will stop their tasks. The interrupted tasks will resume only after the GC task has completed. GC tuning often means reducing this stop-the-world time.

Generational Garbage Collection

Java does not explicitly specify a memory and remove it in the program code. Some people sets the relevant object to null or use System.gc() method to remove the memory explicitly. Setting it to null is not a big deal, but calling System.gc() method will affect the system performance drastically, and must not be carried out. (Thankfully, I have not yet seen any developer in NHN calling this method.)

In Java, as the developer does not explicitly remove the memory in the program code, the garbage collector finds the unnecessary (garbage) objects and removes them. This garbage collector was created based on the following two hypotheses. (It is more correct to call them suppositions or preconditions, rather than hypotheses.)

- Most objects soon become unreachable.
- References from old objects to young objects only exist in small numbers.

These hypotheses are called the **weak generational hypothesis**. So in order to preserve the strengths of this hypothesis, it is physically divided into two - **young generation** and **old generation** - in HotSpot VM.

Young generation: Most of the newly created objects are located here. Since most objects soon become unreachable, many objects are created in the young generation, then disappear. When objects disappear from this area, we say a "**minor GC**" has occurred.

Old generation: The objects that did not become unreachable and survived from the young generation are copied here. It is generally larger than the young generation. As it is bigger in size, the GC occurs less frequently than in the young generation. When objects disappear from the old generation, we say a "**major GC**" (or a "**full GC**") has occurred.

Let's look at this in a chart.

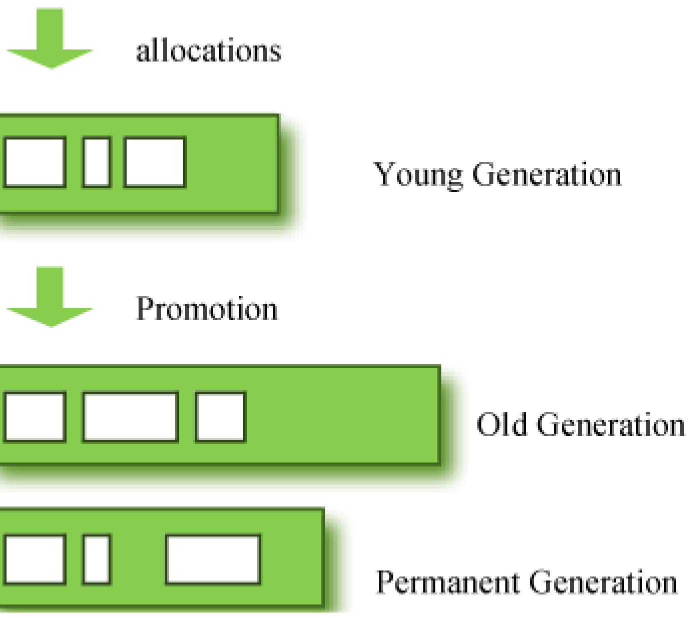


Figure 1: GC Area & Data Flow.

The **permanent generation** from the chart above is also called the "**method area,**" and it stores classes or interned character strings. So, this area is definitely not for objects that survived from the old generation to stay permanently. A GC may occur in this area. The GC that took place here is still counted as a major GC.

Some people may wonder:

> **What if an object in the old generation need to reference an object in the young generation?**

To handle these cases, there is something called the a "**card table**" in the old generation, which is a *512 byte chunk*. Whenever an object in the old generation references an object in the young generation, it is recorded in this table. When a GC is executed for the young generation, only this card table is searched to determine whether or not it is subject for GC, instead of checking the reference of all the objects in the old generation. This card table is managed with **write barrier**. This *write barrier* is a device that allows a faster performance for minor GC. Though a bit of overhead occurs because of this, the overall GC time is reduced.

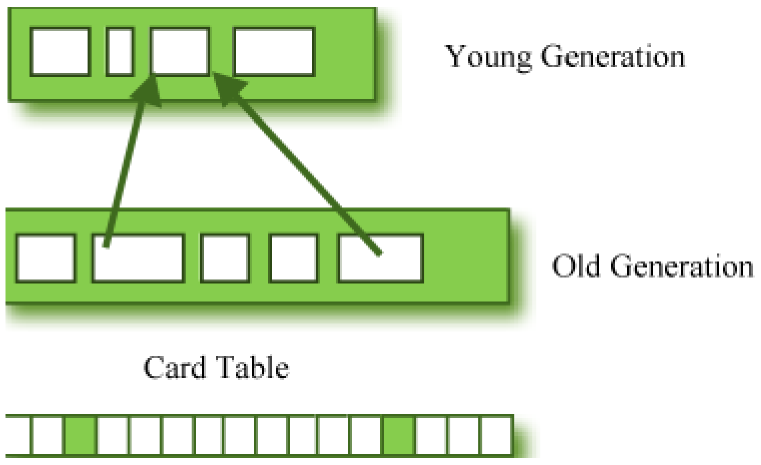


Figure 2: Card Table Structure.

Composition of the Young Generation

In order to understand GC, let's learn about the young generation, where the objects are created for the first time. The young generation is divided into 3 spaces.

- One **Eden** space
- Two **Survivor** spaces

There are 3 spaces in total, two of which are Survivor spaces. The order of execution process of each space is as below:

1. The majority of newly created objects are located in the Eden space.
2. After one GC in the Eden space, the surviving objects are moved to one of the Survivor spaces.
3. After a GC in the Eden space, the objects are piled up into the Survivor space, where other surviving objects already exist.
4. Once a Survivor space is full, surviving objects are moved to the other Survivor space. Then, the Survivor space that is full will be changed to a state where there is no data at all.
5. The objects that survived these steps that have been repeated a number of times are moved to the old generation.

As you can see by checking these steps, one of the Survivor spaces must remain empty. If *data exists in both Survivor spaces, or the usage is 0 for both spaces*, then take that as a sign that **something is wrong with your system**.

The process of data piling up into the old generation through minor GCs can be shown as in the below chart:

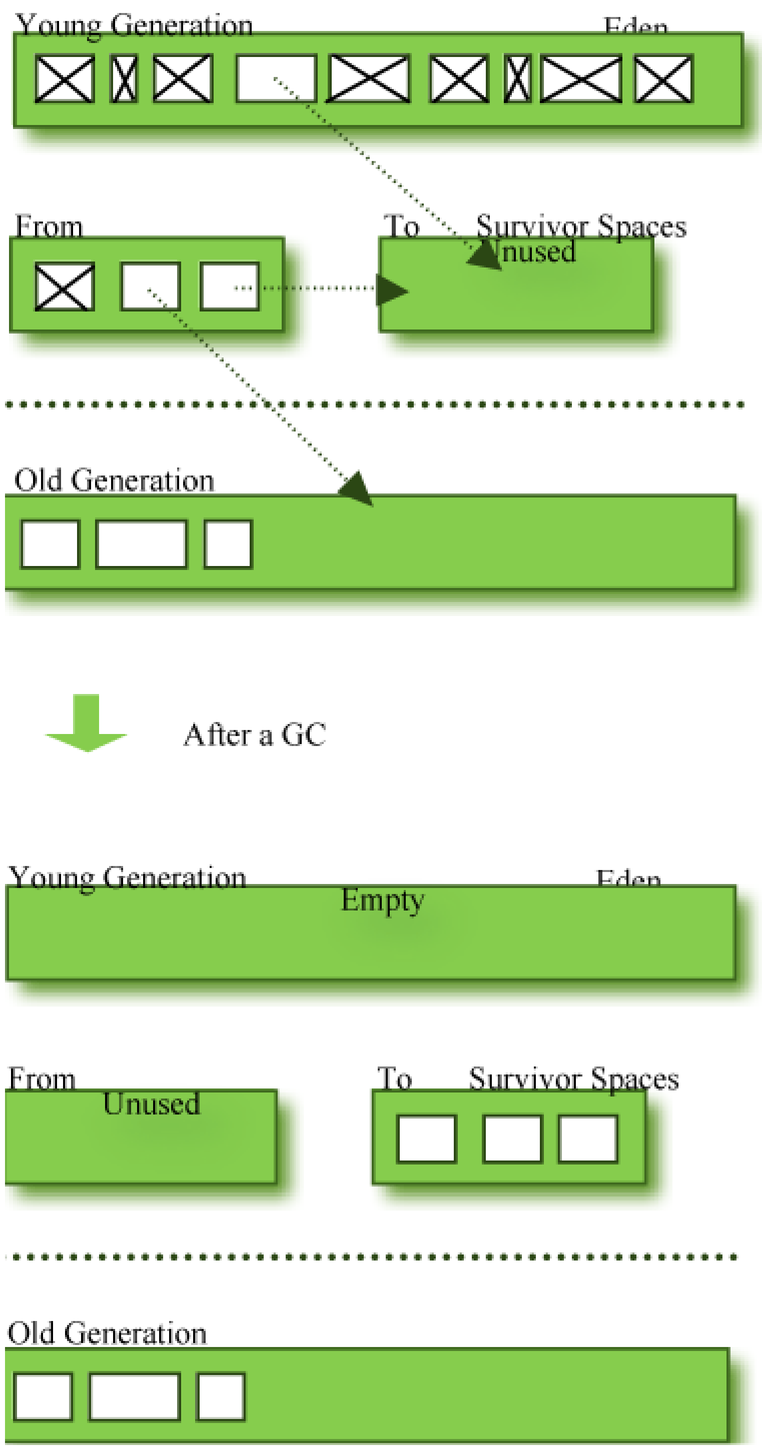


Figure 3: Before & After a GC.
Note that in HotSpot VM, two techniques are used for faster memory allocations. One is called "**bump-the-pointer**," and the other is called "**TLABs (Thread-Local Allocation Buffers)**."

Bump-the-pointer technique tracks the last object allocated to the Eden space. That object will be located on top of the Eden space. And if there is an object created afterwards, it checks only if the size of the object is suitable for the Eden space. If the said object seems right, it will be placed in the Eden space, and the new object goes on top. So, when new objects are created, only the lastly added object needs to be checked, which allows much faster memory allocations. However, it is a different story if we consider a multithreaded environment. To save objects used by multiple threads in the Eden space for Thread-Safe, an inevitable lock will occur and the performance will drop due to the lock-contention. **TLABs** is the solution to this problem in HotSpot VM. This allows each thread to have a small portion of its Eden space that corresponds to its own share. As each thread can only access to their own TLAB, even the bump-the-pointer technique will allow memory allocations without a lock.

This has been a quick overview of the GC in the young generation. You do not necessarily have to remember the two techniques that I have just mentioned. You will not go to jail for not knowing them. But please remember that after the objects are first created in the Eden space, and the long-surviving objects are moved to the old generation through the Survivor space.

GC for the Old Generation

The old generation basically performs a GC when the data is full. The execution procedure varies by the GC type, so it would be easier to understand if you know different types of GC.

According to JDK 7, there are 5 GC types.

1. Serial GC
2. Parallel GC
3. Parallel Old GC (Parallel Compacting GC)
4. Concurrent Mark & Sweep GC (or "CMS")
5. Garbage First (G1) GC

Among these, the **serial GC must not be used on an operating server**. This GC type was created when there was only one CPU core on desktop computers. Using this serial GC will drop the application performance significantly.

Now let's learn about each GC type.

Serial GC (-XX:+UseSerialGC)

The GC in the young generation uses the type we explained in the previous paragraph. The GC in the old generation uses an algorithm called "**mark-sweep-compact**."

1. The first step of this algorithm is to mark the surviving objects in the old generation.
2. Then, it checks the heap from the front and leaves only the surviving ones behind (sweep).
3. In the last step, it fills up the heap from the front with the objects so that the objects are piled up consecutively, and divides the heap into two parts: one with objects and one without objects (compact).

The serial GC is suitable for a small memory and a small number of CPU cores.

Parallel GC (-XX:+UseParallelGC)

Figure 4: Difference between the Serial GC and Parallel GC.
From the picture, you can easily see the difference between the serial GC and parallel GC. While the serial GC uses only one thread to process a GC, the parallel GC uses several threads to process a GC, and therefore, faster. This GC is useful when there is enough memory and a large number of cores. It is also called the "**throughput GC**."

Parallel Old GC(-XX:+UseParallelOldGC)

Parallel Old GC was supported since JDK 5 update. Compared to the parallel GC, the only difference is the GC algorithm for the old generation. It goes through three steps: *mark – summary – compaction*. The summary step identifies the surviving objects separately for the areas that the GC have previously performed, and thus different from the sweep step of the mark-sweep-compact algorithm. It goes through a little more complicated steps.

CMS GC (-XX:+UseConcMarkSweepGC)

Figure 5: Serial GC & CMS GC.
As you can see from the picture, the Concurrent Mark-Sweep GC is much more complicated than any other GC types that I have explained so far. The early *initial mark* step is simple. The surviving objects among the objects the closest to the classloader are searched. So, the pausing time is very short. In the *concurrent mark* step, the objects referenced by the surviving objects that have just been confirmed are tracked and checked. The difference of this step is that it proceeds while other threads are processed at the same time. In the *remark* step, the objects that were newly added or stopped being referenced in the concurrent mark step are checked. Lastly, in the *concurrent sweep* step, the garbage collection procedure takes place. The garbage collection is carried out while other threads are still being processed. Since this GC type is performed in this manner, the pausing time for GC is very short. The CMS GC is also called the low latency GC, and is **used when the response time from all applications is crucial**.

While this GC type has the advantage of short stop-the-world time, it also has the following disadvantages.

- It uses more memory and CPU than other GC types.
- The compaction step is not provided by default.

You need to carefully review before using this type. Also, if the compaction task needs to be carried out because of the many memory fragments, the stop-the-world time can be longer than any other GC types. You need to check how often and how long the compaction task is carried out.

G1 GC

Finally, let's learn about the garbage first (G1) GC.

Figure 6: Layout of G1 GC.
If you want to understand G1 GC, forget everything you know about the young generation and the old generation. As you can see in the picture, one object is allocated to each grid, and then a GC is executed. Then, once one area is full, the objects are allocated to another area, and then a GC is executed. The steps where the data moves from the three spaces of the young generation to the old generation cannot be found in this GC type. This type was created to replace the CMS GC, which has causes a lot of issues and complaints in the long term.

The biggest advantage of the G1 GC is its **performance**. It is faster than any other GC types that we have discussed so far. But in JDK 6, this is called an *early access* and can be used only for a test. It is officially included in JDK 7. In my personal opinion, we need to go through a long test period (at least 1 year) before NHN can use JDK7 in actual services, so you probably should wait a while. Also, I heard a few times that a JVM crash occurred after applying the G1 in JDK 6. Please wait until it is more stable.

I will talk about the **GC tuning** in the next issue, but I would like to ask you one thing in advance. If the size and the type of all objects created in the application are identical, all the GC options for WAS used in our company can be the same. But the size and the lifespan of the objects created by WAS vary depending on the service, and the type of equipment varies as well. In other words, just because a certain service uses the GC option "A," it does not mean that the same option will bring the best results for a different service. It is necessary to find the best values for the WAS threads, WAS instances for each equipment and each GC option by constant tuning and monitoring. This did not come from my personal experience, but from the discussion of the engineers making Oracle JVM for JavaOne 2010.

In this issue, we have only glanced at the GC for Java. Please look forward to our next issue, where I will talk about **how to monitor the Java GC status and tune GC**.

I would like to note that I referred to a new book released in December 2011 called "*Java Performance*" (Amazon, it can also be viewed from safari online, if the company provides an account), as well as “*Memory Management in the Java HotSpotTM Virtual Machine*,” a white paper provided by the Oracle website. (The book is different from "*Java Performance Tuning*.")

By Sangmin Lee, Senior Engineer at Performance Engineering Lab, NHN Corporation.

How to Monitor Java Garbage Collection

This is the second article in the series of "*Become a Java GC Expert*". In the first issue Understanding Java Garbage Collection we have learned about the processes for different GC algorithms, about how GC works, what Young and Old Generation is, what you should know about the 5 types of GC in the new JDK 7, and what the performance implications are for each of these GC types.

In this article, I will explain **how JVM is actually running Garbage Collection in the real time**.

What is GC Monitoring?

Garbage Collection Monitoring refers to the *process of figuring out how JVM is running GC*. For example, we can find out:

1. when an object in young has moved to old and by how much,
2. or when stop-the-world has occurred and for how long.

GC monitoring is carried out *to see if JVM is running GC efficiently*, and *to check if additional GC tuning is necessary*. Based on this information, the application can be edited or GC method can be changed (**GC tuning**).

How to Monitor GC?

There are different ways to monitor GC, but the only difference is how the GC operation information is shown. GC is done by JVM, and since the GC monitoring tools disclose the GC information provided by JVM, you will get the same results no matter how you monitor GC. Therefore, you do not need to learn all methods to monitor GC, but since it only requires a little amount of time to learn each GC monitoring method, knowing a few of them can help you use the right one for different situations and environments.

The tools or JVM options listed below cannot be used universally regardless of the HVM vendor. This is because there is no need for a "standard" for disclosing GC information. In this example we will use **HotSpot JVM** (Oracle JVM). Since NHN is using Oracle (Sun) JVM, there should be no difficulties in applying the tools or JVM options that we are explaining here.

First, the GC monitoring methods can be separated into **CUI** and **GUI** depending on the access interface. The typical CUI GC monitoring method involves using a separate CUI application called "**jstat**", or selecting a JVM option called "**verbosegc**" when running JVM.

GUI GC monitoring is done by using a separate GUI application, and three most commonly used applications would be "jconsole", "jvisualvm" and "Visual GC".

Let's learn more about each method.

jstat

jstat is a monitoring tool in HotSpot JVM. Other monitoring tools for HotSpot JVM are **jps** and **jstatd**. Sometimes, you need all three tools to monitor a Java application.

jstat does not provide only the GC operation information display. It also provides class loader operation information or Just-in-Time compiler operation information. Among all the information jstat can provide, in this article we will only cover its functionality to *monitor* GC operating information.

jstat is located in $JDK_HOME/bin, so if *java* or *javac* can run without setting a separate directory from the command line, so can jstat.

You can try running the following in the command line.

```
$> jstat -gc  $<vmid$> 1000S0C       S1C        S0U    S1U       EC          EU
OC          OU          PC          PU          YGC      YGCT     FGC       FGCT
GCT3008.0   3072.0    0.0      1511.1   343360.0   46383.0      699072.0   283690.2
75392.0    41064.3    2540     18.454     4       1.133     19.5883008.0   3072.0    0.0
1511.1   343360.0   47530.9      699072.0   283690.2   75392.0     41064.3    2540
18.454    4      1.133     19.5883008.0   3072.0    0.0      1511.1   343360.0
47793.0     699072.0   283690.2   75392.0     41064.3    2540    18.454    4      1.133
19.588$>
```

Just like in the example, the real type data will be output along with the following columns: **S0C S1C S0U S1U EC EU OC OU PC**.

vmid (Virtual Machine ID), as its name implies, is the **ID** for the VM. Java applications running either on a local machine or on a remote machine can be specified using vmid. The vmid for Java application running on a local machine is called **lvmid** (Local vmid), and usually is PID. To find out the lvmid, you can write the PID value using a **ps** command or Windows task manager, but we suggest **jps** because PID and lvmid does not always match. **jps** stands for Java PS. jps shows *vmids* and main method information. Just like ps shows PIDs and process names.

Find out the vmid of the Java application that you want to monitor by using jps, then use it as a parameter in jstat. If you use jps alone, only bootstrap information will show when several WAS instances are running in one equipment. We suggest that you use **ps -ef | grep java** command along with **jps**.

GC performance data needs constant observation, therefore when running jstat, try to output the GC monitoring information on a regular basis.

For example, running "**jstat –gc <vmid> 1000**" (or 1s) will display the GC monitoring data on the console every 1 second. "**jstat –gc <vmid> 1000 10**" will display the GC monitoring information once every 1 second for 10 times in total.

There are many options other than **-gc**, among which GC related ones are listed below.

Option Name	Description
gc	It shows the current size for each heap area and its current usage (Ede, survivor, old, etc.), total number of GC performed, and the accumulated time for GC operations.
gccapactiy	It shows the minimum size (ms) and maximum size (mx) of each heap area, current size, and the number of GC performed for each area. (Does not show current usage and accumulated time for GC operations.)
gccause	It shows the "information provided by -gcutil" + reason for the last GC and the reason for the current GC.
gcnew	Shows the GC performance data for the new area.
gcnewcapacity	Shows statistics for the size of new area.
gcold	Shows the GC performance data for the old area.
gcoldcapacity	Shows statistics for the size of old area.
gcpermcapacity	Shows statistics for the permanent area.
gcutil	Shows the usage for each heap area in percentage. Also shows the total number of GC performed and the accumulated time for GC operations.

Only looking at frequency, you will probably use **-gcutil** (or -gccause), **-gc** and **-gccapacity** the most in that order.

- **-gcutil** is used to check the usage of heap areas, the number of GC performed, and the total accumulated time for GC operations,
- while **-gccapacity** option and others can be used to check the actual size allocated.

You can see the following output by using the **-gc** option:

```
S0C      S1C    …   GCT
1248.0   896.0  …   1.246
1248.0   896.0  …   1.246
…        …
…        …
```

Different jstat options show different types of columns, which are listed below. Each column information will be displayed when you use the "jstat option" listed on the right.

Column	Description	Jstat Option
		-gc

S0C	Displays the current size of Survivor0 area in KB	-gccapacity -gcnew -gcnewcapacity
S1C	Displays the current size of Survivor1 area in KB	-gc -gccapacity -gcnew -gcnewcapacity
S0U	Displays the current usage of Survivor0 area in KB	-gc -gcnew
S1U	Displays the current usage of Survivor1 area in KB	-gc -gcnew
EC	Displays the current size of Eden area in KB	-gc -gccapacity -gcnew -gcnewcapacity
EU	Displays the current usage of Eden area in KB	-gc -gcnew
OC	Displays the current size of old area in KB	-gc -gccapacity -gcold -gcoldcapacity
OU	Displays the current usage of old area in KB	-gc -gcold
PC	Displays the current size of permanent area in KB	-gc -gccapacity -gcold -gcoldcapacity - gcpermcapacity
PU	Displays the current usage of permanent area in KB	-gc -gcold
YGC	The number of GC event occurred in young area	-gc -gccapacity -gcnew -gcnewcapacity -gcold -gcoldcapacity - gcpermcapacity

		-gcutil -gccause
YGCT	The accumulated time for GC operations for Yong area	-gc -gcnew -gcutil -gccause
FGC	The number of full GC event occurred	-gc -gccapacity -gcnew -gcnewcapacity -gcold -gcoldcapacity - gcpermcapacity -gcutil -gccause
FGCT	The accumulated time for full GC operations	-gc -gcold -gcoldcapacity - gcpermcapacity -gcutil -gccause
GCT	The total accumulated time for GC operations	-gc -gcold -gcoldcapacity - gcpermcapacity -gcutil -gccause
NGCMN	The minimum size of new area in KB	-gccapacity -gcnewcapacity
NGCMX	The maximum size of max area in KB	-gccapacity -gcnewcapacity
NGC	The current size of new area in KB	-gccapacity -gcnewcapacity
OGCMN	The minimum size of old area in KB	-gccapacity -gcoldcapacity
OGCMX	The maximum size of old area in KB	-gccapacity

		-gcoldcapacity
OGC	The current size of old area in KB	-gccapacity -gcoldcapacity
PGCMN	The minimum size of permanent area in KB	-gccapacity - gcpermcapacity
PGCMX	The maximum size of permanent area in KB	-gccapacity - gcpermcapacity
PGC	The current size of permanent generation area in KB	-gccapacity - gcpermcapacity
PC	The current size of permanent area in KB	-gccapacity - gcpermcapacity
PU	The current usage of permanent area in KB	-gc -gcold
LGCC	The cause for the last GC occurrence	-gccause
GCC	The cause for the current GC occurrence	-gccause
TT	Tenuring threshold. If copied this amount of times in young area (S0 ->S1, S1->S0), they are then moved to old area.	-gcnew
MTT	Maximum Tenuring threshold. If copied this amount of times inside young arae, then they are moved to old area.	-gcnew
DSS	Adequate size of survivor in KB	-gcnew

The advantage of **jstat** is that it can always monitor the GC operation data of Java applications running on local/remote machine, as long as a console can be used. From these items, the following result is output when **–gcutil** is used. At the time of GC tuning, pay careful attention to **YGC, YGCT, FGC, FGCT** and **GCT**.

```
S0       S1        E         O         P         YGC      YGCT      FGC      FGCT       GCT 0.00
66.44     54.12     10.58     86.63     217     0.928     2     0.067     0.995 0.00
66.44     54.12     10.58     86.63     217     0.928     2     0.067     0.995 0.00
66.44     54.12     10.58     86.63     217     0.928     2     0.067     0.995
```

These items are important because they show how much time was spent in running GC.

In this example, **YGC** is 217 and **YGCT** is 0.928. So, after calculating the arithmetical average, you can see that it required about *4 ms* (0.004 seconds) for each young GC. Likewise, the average full GC time us *33ms*.

But the arithmetical average often does not help analyzing the actual GC problem. This is due to the severe deviations in GC operation time. (In other words, if the average time is *0.067 seconds* for a full GC, one GC may have lasted 1 ms while the other one lasted *57 ms*.) In order to check the individual GC time instead of the arithmetical average time, it is better to use **-verbosegc**.

-verbosegc

-verbosegc is one of the JVM options specified when running a Java application. While *jstat* can monitor any JVM application that has not specified any options, **-verbosegc** needs to be specified in the beginning, so it could be seen as an unnecessary option (since jstat can be used instead). However, as **-verbosegc** displays easy to understand output results whenever a GC occurs, it is very helpful for monitoring rough GC information.

	jstat	-verbosegc
Monitoring Target	Java application running on a machine that can log in to a terminal, or a remote Java application that can connect to the network by using jstatd	Only when -verbogc was specified as a JVM starting option
Output information	Heap status (usage, maximum size, number of times for GC/time, etc.)	Size of ew and old area before/after GC, and GC operation time
Output Time	Every designated time	Whenever GC occurs
Whenever useful	When trying to observe the changes of the size of heap area	When trying to see the effect of a single GC

The followings are other options that can be used with **-verbosegc**.

- -XX:+PrintGCDetails
- -XX:+PrintGCTimeStamps
- -XX:+PrintHeapAtGC
- -XX:+PrintGCDateStamps (from JDK 6 update 4)

If only **-verbosegc** is used, then **-XX:+PrintGCDetails** is applied by default. Additional options for **–verbosgc** are not exclusive and can be mixed and used together.

When using **-verbosegc**, you can see the results in the following format whenever a minor GC occurs.

```
[GC [<collector>: <starting occupancy1> -> <ending occupancy1>, <pause time1> secs]
<starting occupancy3> -> <ending occupancy3>, <pause time3> secs]
```

Collector	Name of Collector Used for minor gc

starting occupancy1	The size of young area before GC
ending occupancy1	The size of young area after GC
pause time1	The time when the Java application stopped running for minor GC
starting occupancy3	The total size of heap area before GC
ending occupancy3	The total size of heap area after GC
pause time3	The time when the Java application stopped running for overall heap GC, including major GC

This is an example of **-verbosegc** output for **minor GC**:

```
S0     S1      E       O       P          YGC     YGCT     FGC     FGCT      GCT0.00   66.44
54.12  10.58  86.63     217     0.928       2     0.067     0.9950.00   66.44   54.12  10.58
86.63     217     0.928       2     0.067     0.9950.00   66.44   54.12  10.58   86.63     217
0.928       2     0.067     0.995
```

This is the example of output results after an **Full GC** occurred.

```
[Full GC [Tenured: 3485K->4095K(4096K), 0.1745373 secs] 61244K->7418K(63104K), [Perm :
10756K->10756K(12288K)], 0.1762129 secs] [Times: user=0.19 sys=0.00, real=0.19 secs]
```

If a CMS collector is used, then the following CMS information can be provided as well.

As **-verbosegc** option outputs a log every time a GC event occurs, it is easy to see the changes of the heap usage rates caused by GC operation.

(Java) VisualVM + Visual GC

Java Visual VM is a GUI profiling/monitoring tool provided by Oracle JDK.

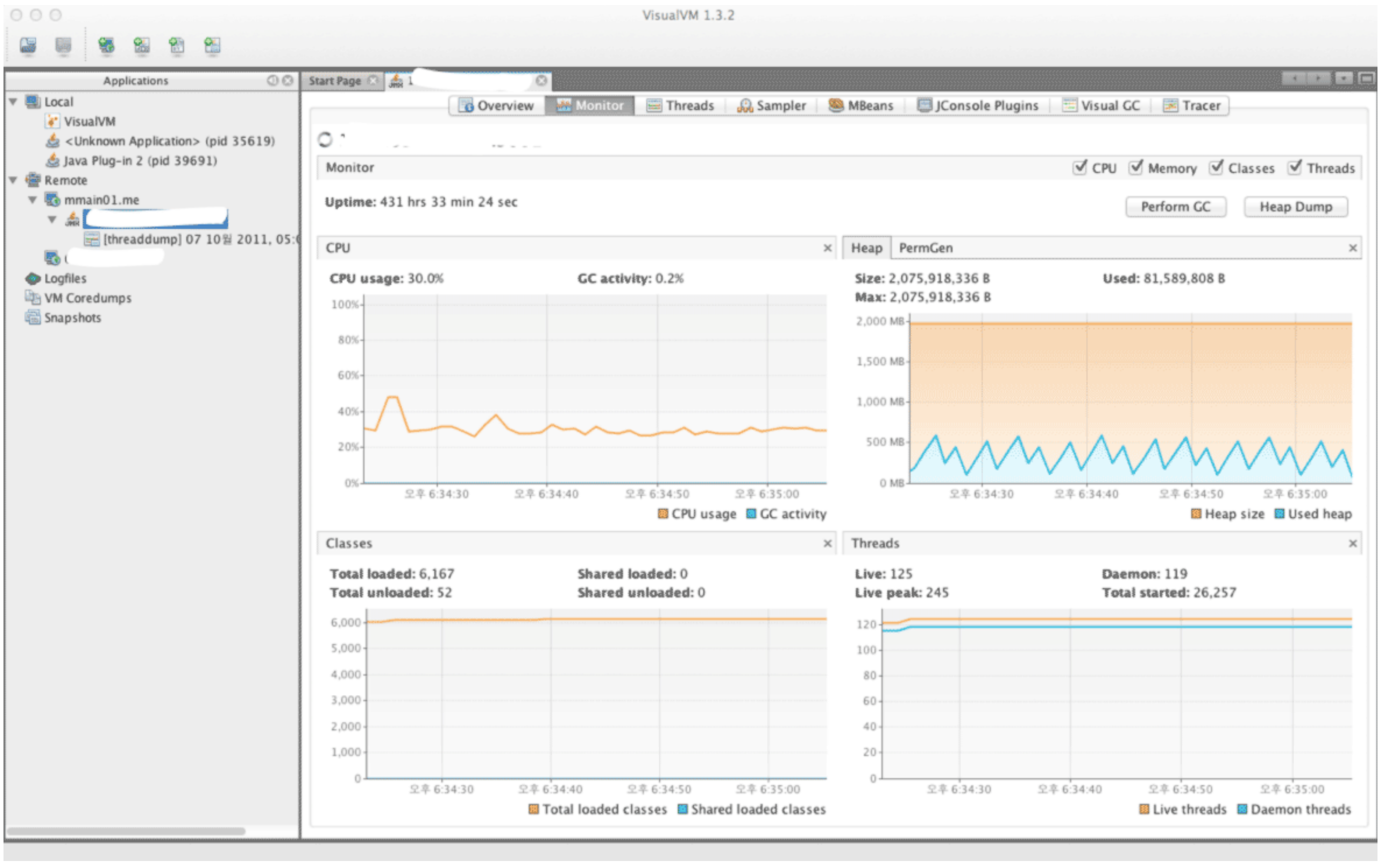


Figure 1: VisualVM Screenshot.

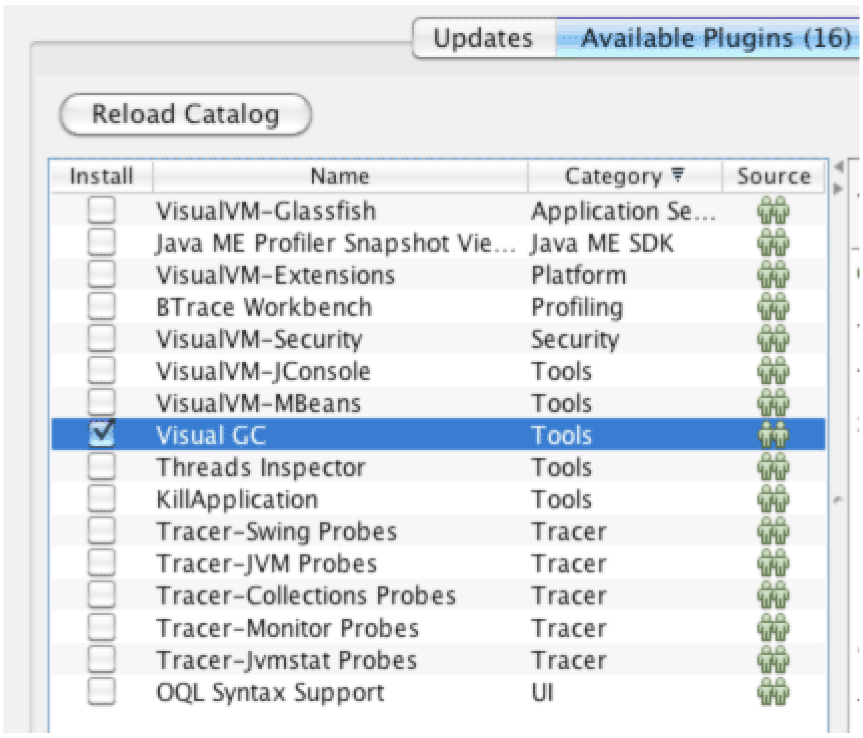
Instead of the version that is included with JDK, you can download Visual VM directly from its website. For the sake of convenience, the version included with JDK will be referred to as Java VisualVM (jvisualvm), and the version available from the website will be referred to as Visual VM (visualvm). The features of the two are not exactly identical, as there are slight differences, such as when installing plug-ins. Personally, I prefer the Visual VM version, which can be downloaded from the website.

After running Visual VM, if you select the application that you wish to monitor from the window on the left side, you can find the "*Monitoring*" tab there. You can get the basic information about GC and Heap from this Monitoring tab.

Though the basic GC status is also available through the basic features of VisualVM, you cannot access detailed information that is available from either **jstat** or **-verbosegc** option.

If you want the detailed information provided by jstat, then it is recommended to install the Visual GC plug-in.

Visual GC can be accessed in real time from the *Tools* menu.

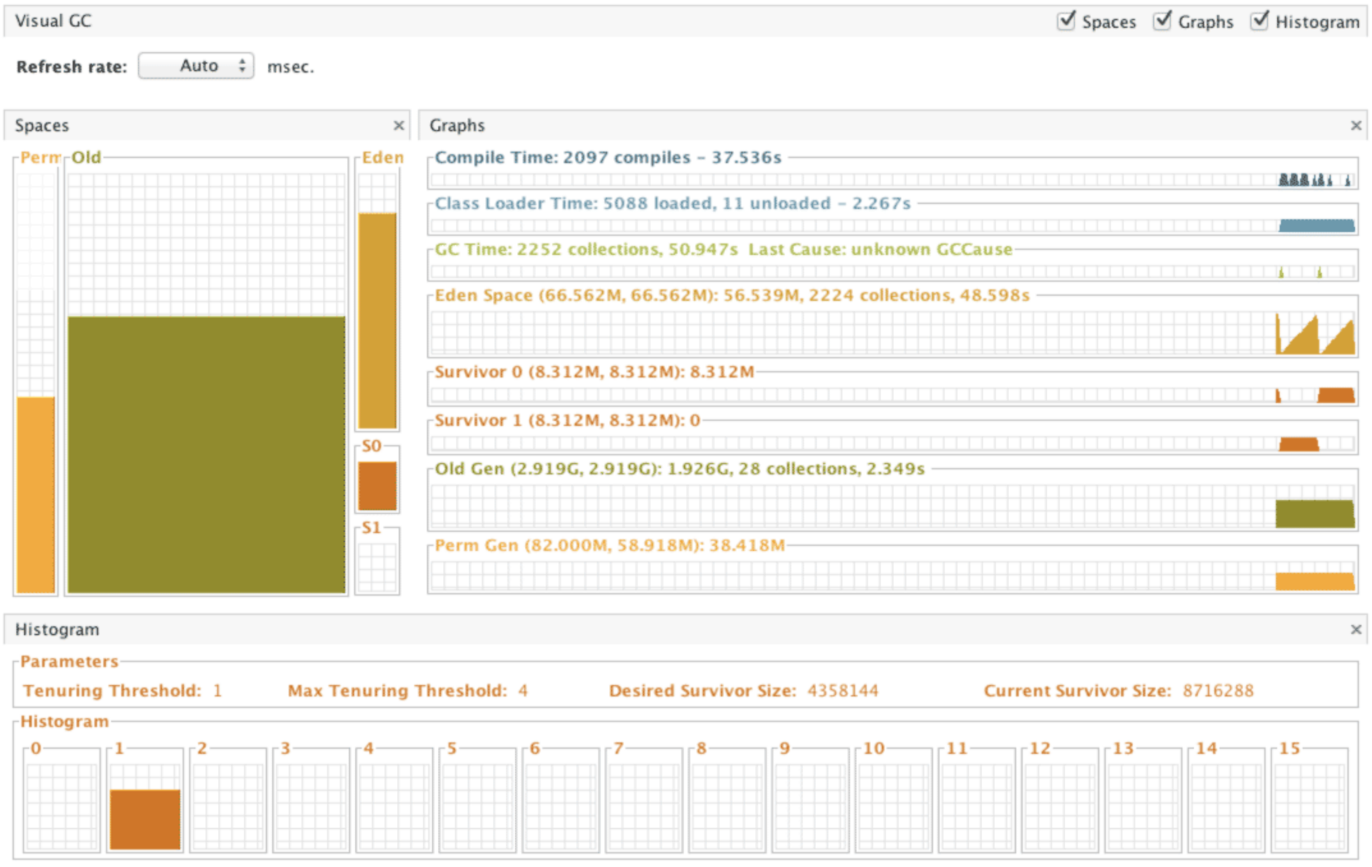

Updates | Available Plugins (16)

Reload Catalog

Install	Name	Category	Source
☐	VisualVM-Glassfish	Application Se...	
☐	Java ME Profiler Snapshot Vie...	Java ME SDK	
☐	VisualVM-Extensions	Platform	
☐	BTrace Workbench	Profiling	
☐	VisualVM-Security	Security	
☐	VisualVM-JConsole	Tools	
☐	VisualVM-MBeans	Tools	
☑	Visual GC	Tools	
☐	Threads Inspector	Tools	
☐	KillApplication	Tools	
☐	Tracer-Swing Probes	Tracer	
☐	Tracer-JVM Probes	Tracer	
☐	Tracer-Collections Probes	Tracer	
☐	Tracer-Monitor Probes	Tracer	
☐	Tracer-Jvmstat Probes	Tracer	
☐	OQL Syntax Support	UI	

Figure 2: Viusal GC Installation Screenshot.

By using Visual GC, you can see the information provided by running **jstatd** in a more intuitive way.


Visual GC ✓ Spaces ✓ Graphs ✓ Histogram

Refresh rate: Auto msec.

Spaces: Perm, Old, Eden, S0, S1

Graphs

Compile Time: 2097 compiles - 37.536s

Class Loader Time: 5088 loaded, 11 unloaded - 2.267s

GC Time: 2252 collections, 50.947s Last Cause: unknown GCCause

Eden Space (66.562M, 66.562M): 56.539M, 2224 collections, 48.598s

Survivor 0 (8.312M, 8.312M): 8.312M

Survivor 1 (8.312M, 8.312M): 0

Old Gen (2.919G, 2.919G): 1.926G, 28 collections, 2.349s

Perm Gen (82.000M, 58.918M): 38.418M

Histogram

Parameters

Tenuring Threshold: 1 Max Tenuring Threshold: 4 Desired Survivor Size: 4358144 Current Survivor Size: 8716288

Histogram

Figure 3: Visual GC execution screenshot.

HPJMeter

HPJMeter is convenient for analyzing **-verbosegc** output results. If Visual GC can be considered as the GUI equivalent of *jstat*, then HPJMeter would be the GUI equivalent of *-verbosgc*. Of course, GC analysis is just one of the many features provided by HPJMeter. HPJMeter is a performance monitoring tool developed by HP. It can be used in HP-UX, as well as Linux and MS Windows.

Originally, a tool called **HPTune** used to provide the GUI analysis feature for **-verbosegc**. However, since the HPTune feature has been integrated into HPJMeter since version 3.0, there is no need to download HPTune separately.

When executing an application, the **-verbosegc** output results will be redirected to a separate file.

You can open the redirected file with HPJMeter, which allows faster and easier GC performance data analysis through the intuitive GUI.

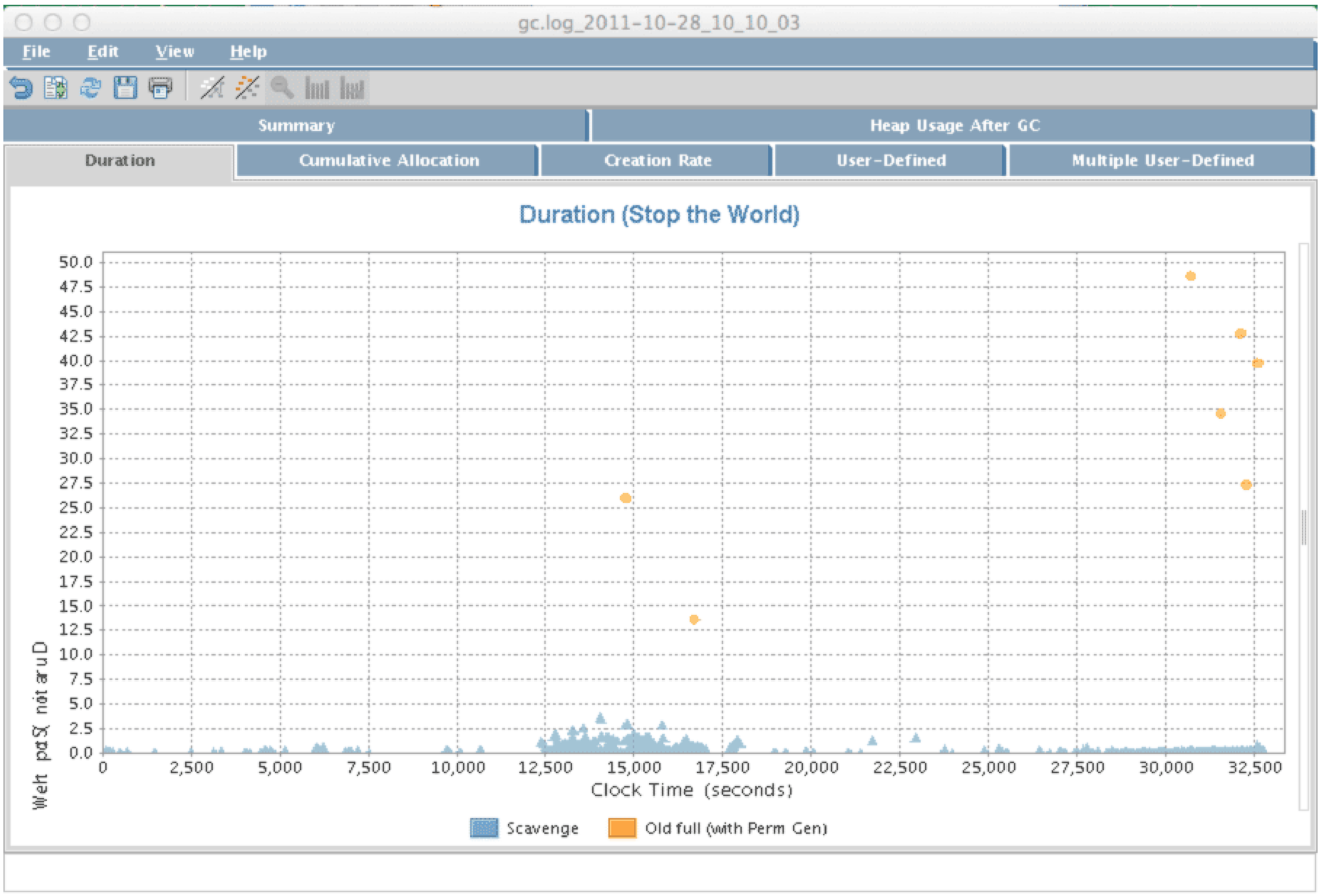


Figure 4: HPJMeter.

What is the Next Article About?

In this article I focused on *how to monitor GC operation information*, as the preparation stage for GC tuning. From my personal experience, I suggest using **jstat** to monitor GC operation, and if you feel that it takes too lmuch time to execute GC, then try **-verbosegc** option to analyze GC. The general GC tuning process is *to analyze the results after applying the changed GC options* after the **-verbosegc** option has been applied based on the analysis. In the next article, we will see the best options for executing GC tuning by using real cases as our examples.

By Sangmin Lee, Senior Engineer at Performance Engineering Lab, NHN Corporation.

How to Tune Java Garbage Collection

This is the *third* article in the series of "*Become a Java GC Expert*". In the first issue Understanding Java Garbage Collection we have learned about the processes for different GC algorithms, about how GC works, what Young and Old Generation is, what you should know about the 5 types of GC in the new JDK 7, and what the performance implications are for each of these GC types.

In the second article How to Monitor Java Garbage Collection I have explained how JVM actually runs the Garbage Collection in the real time, how we can monitor GC, and which tools we can use to make this process faster and more effective.

In this third article based on real cases as our examples **I will show some of the best options you can use for GC tuning**. I have written this article under the assumption that you have already understood the previous articles in this series. Therefore, for your further understanding, if you haven't already read the two previous articles, please do so before reading this one.

Is GC Tuning Required?

Or more precisely **is GC tuning required for Java-based services**? I should say GC tuning is *not* always required for all Java-based services. This means a Java-based system in operation has the following options and actions:

- The memory size has been specified using `-Xms` and `-Xmx` options.
- The `-server` option is included.
- Logs such as *Timeout log* are not left in the system.

In other words, if you have not set the memory size and too many Timeout logs are printed, you need to perform GC tuning on your system.
But, there is one thing to keep in mind: **GC tuning is the last task to be done.**

Think about the fundamental cause of GC tuning. The Garbage Collector clears an object created in Java. The number of objects necessary to be cleared by the garbage collector as well as the number of GCs to be executed depend on the number of objects which have been created. Therefore, to control the GC performed by your system, you should, first, **decrease the number of objects created**.

There is a saying, "many a little makes a mickle." We need to take care of small things, or they will add up and become something big which is difficult to manage.

- We need to use and make `StringBuilder` or `StringBuffer` a way of life instead of `String`.
- And it is better to accumulate as few logs as possible.

However, we know that there are some cases we cannot help. We have seen that XML and JSON parsing use the most memory. Even though we use `String` as little as possible and process logs as well as we can, a huge temporary memory is used for parsing XML or JSON, some 10-100 MB. However, it is difficult not to use XML and JSON. Just understand that it takes too much memory.

If application memory usage improves after repeated tunings, you can start GC tuning. I classify the purposes of GC tuning into two.

1. One is to **minimize the number of objects passed to the old area**;
2. and the other is to **decrease Full GC execution time**.

Minimizing Number of Objects Passed to Old Area

Generational GC is the GC provided by Oracle JVM, excluding the G1 GC which can be used from JDK 7 and higher versions. In other words, an object is created in the Eden area and transferred from and to the Survivor area. After that, the objects left are sent to the Old area. Some objects are created in the Eden area and directly passed to the Old area because of their large size. GC in the Old area takes relatively more time than the GC in the New area. Therefore, decreasing the number of objects passed to the Old area can decrease the full GC in frequency. Decreasing the number of objects passed to the Old area may be misunderstood as choosing to leave the object in the New area. However, this is impossible. Instead, you can **adjust the size of the New area**.

Decreasing Full GC Time

The execution time of Full GC is relatively longer than that of Minor GC. Therefore, if it takes too much time to execute Full GC (1 second or more), timeout may occur in several connected parts.

- If you try to decrease the Old area size to decrease Full GC execution time, `OutOfMemoryError` may occur or the number of Full GCs may increase.
- Alternatively, if you try to decrease the number of Full GC by increasing the Old area size, the execution time will be increased.

Therefore, you need to **set the Old area size to a "proper" value**.

Options Affecting the GC Performance

As I have mentioned at the end of Understanding Java Garbage Collection, do not think that "**Somebody's got a great performance when he used GC options. Why don't we use that option as he did?**" The reason is that **the size of objects created and their lifetime is different from one Web service to another**.

Simply consider, if a task is performed under the conditions of A, B, C, D and E, and the same task is performed under the conditions of only A and B, then which one will be done quicker? From a common-sense standpoint, the answer would be *the task which is performed under conditions of A and B*.

Java GC options are the same. Setting several options does not enhance the speed of executing GC. Rather, it *may* make it slower. The **basic principle of GC tuning** is to **apply the different GC options to two or more servers and compare them**, and then add those options to the server for which the server has demonstrated enhanced performance or better GC time. Keep this in mind.

The following table shows options related to memory size among the GC options that can affect performance.

Table 1: JVM Options to Be Checked for GC Tuning.

Classification	Option	Description
Heap area size	`-Xms`	Heap area size when starting JVM
	`-Xmx`	Maximum heap area size
New area size	`-XX:NewRatio`	Ratio of New area and Old area
	`-XX:NewSize`	New area size
	`-XX:SurvivorRatio`	Ratio of Eden area and Survivor area

I frequently use `-Xms`, `-Xmx`, and `-XX:NewRatio` options for GC tuning. `-Xms` and `-Xmx` option are particularly required. How you set the `NewRatio` option makes a significant difference on GC performance.

Some people ask **how to set the Perm area size**? You can set the Perm area size with the `-XX:PermSize` and `-XX:MaxPermSize` options but only when `OutOfMemoryError` occurs and the cause is the Perm area size.

Another option that may affect the GC performance is the GC type. The following table shows available options by GC type (based on JDK 6.0).

Table 2: Available Options by GC Type.

Classification	Option	Remarks
Serial GC	-XX:+UseSerialGC	
Parallel GC	-XX:+UseParallelGC -XX:ParallelGCThreads=value	
Parallel Compacting GC	-XX:+UseParallelOldGC	
CMS GC	-XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:+CMSParallelRemarkEnabled -XX:CMSInitiatingOccupancyFraction=value -XX:+UseCMSInitiatingOccupancyOnly	
G1	-XX:+UnlockExperimentalVMOptions -XX:+UseG1GC	In JDK 6, these two options must be used together.

Except G1 GC, the GC type is changed by setting the option at the first line of each GC type. The most general GC type that does not intrude is Serial GC. It is optimized for client systems.

There are a lot of options that affect GC performance. But you can get significant effect by setting the options mentioned above. Remember that setting too many options does not promise enhanced GC execution time.

Procedure of GC Tuning

The procedure of GC tuning is similar to the general performance improvement procedure. The following is the GC tuning procedure that I use.

1. Monitoring GC status

You need to monitor the GC status to check the GC status of the system in operation. Please see various GC monitoring methods in How to Monitor Java Garbage Collection.

2. Deciding whether to tune GC after analyzing the monitoring result

After checking the GC status, you should analyze the monitoring result and decide whether to tune GC or not. If the analysis shows that the time taken to execute GC is just 0.1-0.3 seconds. you don't need to waste your time on tuning the GC. However, **if the GC execution time is 1-3 seconds, or more than 10 seconds, GC tuning is necessary**.

But, if you have allocated about 10GB Java memory and it is impossible to decrease the memory size, there is no way to tune GC. Before tuning GC, you need to think about why you need to allocate large memory size. If you have allocated the memory of 1 GB or 2 GB and `OutOfMemoryError` occurs, you should execute heap dump to verify and remove the cause.

> **Note:**
> Heap dump is a file of the memory that is used to check the objects and data in the Java memory. This file can be created by using the **jmap** command included in the JDK. While creating the file, the Java process stops. Therefore, do not create this file while the system is operating.
>
> Search on the Internet the detailed description on heap dump. For Korean readers, see my book I published last year: The story of troubleshooting for Java developers and system operators (Sangmin Lee, Hanbit Media, 2011, 416 pages).

3. Setting GC type/memory size

If you have decided on GC tuning, select the GC type and set the memory size. At this time, if you have several servers, it is important to check the difference of each GC option by setting different GC options for each server.

4. Analyzing results

Start analyzing the results after collecting data for at least 24 hours after setting GC options. If you are lucky, you will find the most suitable GC options for the system. If you are not, you should analyze the logs and check how the memory has been allocated. Then you need to find the optimum options for the system by changing the GC type/memory size.

5. If the result is satisfactory, apply the option to all servers and terminate GC tuning.

If the GC tuning result is satisfactory, apply the option to all the servers and terminate GC tuning.

In the following section, you will see the tasks to be done in each stage.

Monitoring GC Status and Analyzing Results

The best way to check the GC status of the Web Application Server (WAS) in operation is to use the **jstat** command. I have explained the jstat command in How To Monitor Java Garbage Collection, so I will describe the data to check in this article.

The following example shows a JVM for which GC tuning has not been done (however, it is not the operation server).

```
$ jstat -gcutil 21719 1s
S0    S1    E     O     P     YGC    YGCT     FGC    FGCT    GCT
48.66 0.00 48.10 49.70 77.45 3428 172.623 3 59.050 231.673
48.66 0.00 48.10 49.70 77.45 3428 172.623 3 59.050 231.673
```

Here, check the values of YGC and YGCT. Divide YGCT by YGC. Then you get 0.050 seconds (50 ms). It means that it takes average 50 ms to execute GC in the Young area. With that result, you don't need to care about GC for the Young area.

And now, check the values of FGCT and FGC. Divide FGCT by FGC. Then you get 19.68 seconds. It means that it takes average 19.68 seconds to execute GC. It may take 19.68 seconds to execute GC three times. Otherwise, it takes 1 second to execute GC two times and 58 seconds for once. In both cases, GC tuning is required.

You can easily check GC status by using the **jstat** command; however, the best way to analyze GC is by generating logs with the `-verbosegc` option. For a detailed description on how to generate and tools to analyze logs, I have explained it the previous article. **HPJMeter** is my favorite among tools that are used to analyze the `-verbosegc` log. It is easy to use and analyze. With HPJmeter you can easily check the distribution of GC execution times and the frequency of GC occurrence.

If the GC execution time meets all of the following conditions, GC tuning is not required.

- Minor GC is processed quickly (within 50 ms).
- Minor GC is not frequently executed (about 10 seconds).
- Full GC is processed quickly (within 1 second).
- Full GC is not frequently executed (once per 10 minutes).

The values in parentheses are not the absolute values; they vary according to the service status. Some services may be satisfied with 0.9 seconds of Full GC processing speed, but some may not. Therefore, check the values and decide whether to execute GC tuning or not by considering each service.

There is one thing you should be careful of when you check the GC status; do not check the time of Minor GC and Full GC only. You must **check the number of GC executions**, as well. If the New area size is too small, Minor GC will be too frequently executed (sometimes once or more per 1 second). In addition, the number of objects passed to the Old area increases, causing increased Full GC executions. Therefore, apply the `-gccapacity` option in the stat command to check how much the area is occupied.

Setting GC Type/Memory Size

Setting GC Type

There are five GC types for Oracle JVM. However, if not JDK 7, one among Parallel GC, Parallel Compacting GC and CMS GC should be selected. There is no principle or rule to decide which one to select.

If so, **how can we select one?** The most recommended way is to apply all three. However, one thing is clear - CMS GC is faster than other Parallel GCs. At this time, if so, just apply CMS GC. However, CMS GC is not always faster. Generally, Full GC of CMS GC is fast, however, when concurrent mode failure occurs, it is slower than other Parallel GCs.

Concurrent mode failure

Let's take a deeper look into the concurrent mode failure.

The biggest difference between Parallel GC and CMS GC is the compaction task. The compaction task is to remove memory fragmentation by compacting memory in order to remove the empty space between allocated memory areas.

In the Parallel GC type, the compaction is executed whenever Full GC is executed, taking too much time. However, after executing Full GC, memory can be allocated in a faster way since the next memory can be allocated sequentially.

On the contrary, CMS GC does not accompany compaction. Therefore, the CMS GC is executed faster. However, when compaction is not executed, some empty spaces are generated in the memory as before executing Disk Defragmenter. Therefore, there may be no space for large objects. For example, 300 MB is left in the Old area, but some 10 MB objects cannot be sequentially saved in the area. In this case, "Concurrent mode failure" warning occurs and compaction is executed. However, if CMS GC is used, it takes a longer time to execute compaction than other Parallel GCs. And, it may cause another problem. For a more detailed description on concurrent mode failure, see Understanding CMS GC Logs, written by Oracle engineers.

In conclusion, you should find the best GC type for your system.

Each system requires its proper GC type, so you need to find the best GC type for your system. If you are running six servers, I recommend you to set the same options for each of two servers, add the `-verbosegc` option, and then analyze the result.

Setting Memory Size

The following shows the relationship between the memory size, the number of GC execution, and the GC execution time.

- Large memory size
  - decreases the number of GC executions.

    - increases the GC execution time.

- Small memory size
    - decreases the GC execution time.
    - increases the number of GC executions.

There is no "right" answer to set the memory size to small or large. 10 GB is OK if the server resource is good and Full GC can be completed within 1 second even when the memory has been set to 10 GB. But most servers are not in the status. When the memory is set to 10 GB, it takes about 10 ~ 30 seconds to execute Full GC. Of course, the time may vary according the object size.

If so, **how we should set the memory size?** Generally, I recommend 500 MB. But note that it does not mean that you should set the WAS memory with the `-Xms500m` and `-Xmx500m` options. Based on the current status before GC tuning, check the memory size left after Full GC. If there is about 300 MB left after Full GC, it is good to set the memory to 1 GB (300 MB (for default usage) + 500 MB (minimum for the Old area) + 200 MB (for free memory)). That means you should set the memory space with more than 500 MB for the Old area. Therefore, if you have three operation servers, set one server to 1 GB, one to 1.5 GB, and one to 2 GB, and then check the result.

Theoretically, GC will be done fast in the order of 1 GB > 1.5 GB > 2 GB, so 1 GB will be the fastest to execute GC. However, it cannot be guaranteed that it takes 1 second to execute Full GC with 1 GB and 2 seconds with 2 GB. The time depends on the server performance and the object size. Therefore, the best way to create the measurement data is to set as many as possible and monitor them.

You should set one more thing for setting the memory size: `NewRatio`. `NewRatio` is the ratio of the New area and the Old area. If `XX:NewRatio=1`, New area:Old area is 1:1. For 1 GB, New area:Old area is 500MB: 500MB. If `NewRatio` is 2, New area:Old area is 1:2. Therefore, as the value gets larger, the Old area size gets larger and the New area size gets smaller.

It may not be an important thing, but `NewRatio` value significantly affects the entire GC performance. If the New area size is small, much memory is passed to the Old area, causing frequent Full GC and taking a long time to handle it.

You may simply think that `NewRatio 1` would be the best; however, it may not be so. When `NewRatio` is set to 2 or 3, the entire GC status may be better. And I have seen such cases.

What is the fastest way to complete GC tuning? Comparing the results from performance tests is the fastest way to get the result. To set different options for each server and monitor the status, it is recommended to check the data after at least one or two days. However, you should prepare for giving the same load with the operation situation when you execute GC tuning through performance test. And the request ratio such as the URL that gives the load must be identical to that of the operation situation. However, giving accurate load is not easy for the professional performance tester and takes too long time for preparing. Therefore, it is more convenient and easier to apply the options to operation and wait for the result even though it takes a longer time.

Analyzing GC Tuning Results

After applying the GC option and setting the `-verbosegc` option, check whether the logs are accumulated as desired with the `tail` command. If the option is not exactly set and no log is accumulated, you will waste your time. If logs are accumulated as desired, check the result after collecting data for one or two days. The easiest way is to move logs to the local PC and analyze the data by using **HPJMeter**.

In the analysis, focus on the following. The priority is determined by me. The most important item to decide the GC option is Full GC execution time.

- Full GC execution time
- Minor GC execution time
- Full GC execution interval
- Minor GC execution interval
- Entire Full GC execution time
- Entire Minor GC execution time
- Entire GC execution time
- Full GC execution times
- Minor GC execution timesl

It is a very lucky case to find the most appropriate GC option, and in most cases, it's not. Be careful when executing GC tuning because `OutOfMemoryError` may occur if you try to complete GC tuning all at once.

Examples of Tuning

So far, we have theoretically discussed GC tuning without any examples. Now we will take a look at the examples of GC tuning.

Example 1

The following example is GC tuning for **Service S**. For the newly developed Service S, it took too much time to execute Full GC.

See the result of `jstat -gcutil`.

```
S0 S1 E O P YGC YGCT FGC FGCT GCT
12.16 0.00 5.18 63.78 20.32 54 2.047 5 6.946 8.993
```

Information to the left **Perm** area is not important for the initial GC tuning. At this time, the values from the right YGC are important.

The average value taken to execute Minor GC and Full GC once is calculated as below.

Table 3: Average Time Taken to Execute Minor GC and Full GC for Service S.

GC Type	GC Execution Times	GC Execution Time	Average
Minor GC	54	2.047	37 s
Full GC	5	6.946	1,389 ms

37 ms is not bad for Minor GC. However, **1.389 seconds** for Full GC means that timeout may frequently occur when GC occurs in the system of which DB Timeout is set to 1 second. In this case, the system requires GC tuning.

First, you should check how the memory is used before starting GC tuning. Use the `jstat -gccapacity` option to check the memory usage. The result checked from this server is as follows.

```
NGCMN NGCMX NGC S0C S1C EC OGCMN OGCMX OGC OC PGCMN PGCMX PGC PC YGC FGC
212992.0 212992.0 212992.0 21248.0 21248.0 170496.0 1884160.0 1884160.0 1884160.0 1884160.0 262144.0 262144.0 262144.0 262144.0 54 5
```

The key values are as follows.

- New area usage size: 212,992 KB
- Old area usage size: 1,884,160 KB

Therefore, the totally allocated memory size is 2 GB, excluding the Perm area, and New area:Old area is 1:9. To check the status in a more detailed way than **jstat**, the `-verbosegc` log has been added and three options were set for the three instances as shown below. No other option has been added.

- NewRatio=2
- NewRatio=3
- NewRatio=4

After one day, the GC log of the system has been checked. Fortunately, no Full GC has occurred in this system after `NewRatio` has been set.

Why? The reason is that most of the objects created from the system are destroyed soon, so the objects are not passed to the Old area but destroyed in the New area.

In this status, it is not necessary to change other options. Just select the best value for `NewRatio`. So, **how can we determine the best value?** To get it, analyze the average response time of Minor GC for each `NewRatio`.

The average response time of Minor GC for each option is as follows:

- NewRatio=2: 45 ms
- NewRatio=3: 34 ms
- NewRatio=4: 30 ms

We have concluded that NewRatio=4 is the best option since the GC time is the shortest even though the New area size is the smallest. After applying the GC option, the server has no Full GC.

For your information, the following is the result of executing `jstat -gcutil` some days after the JVM of the service had started.

```
S0 S1 E O P YGC YGCT FGC FGCT GCT
8.61 0.00 30.67 24.62 22.38 2424 30.219 0 0.000 30.219
```

You many think that GC has not frequently occurred since the server has few requests. However, Full GC has not been executed while Minor GC has been executed 2,424 times.

Example 2

This example is for Service A. We found that the JVM had not operated for a long time (8 seconds or more) periodically in the Application Performance Manager (APM) in the company. So we executed GC tuning. We were searching for the reason and found that it took a long time to execute Full GC, so we decided to execute GC tuning.

As the starting stage of GC tuning, we added the `-verbosegc` option and the result is as follows.

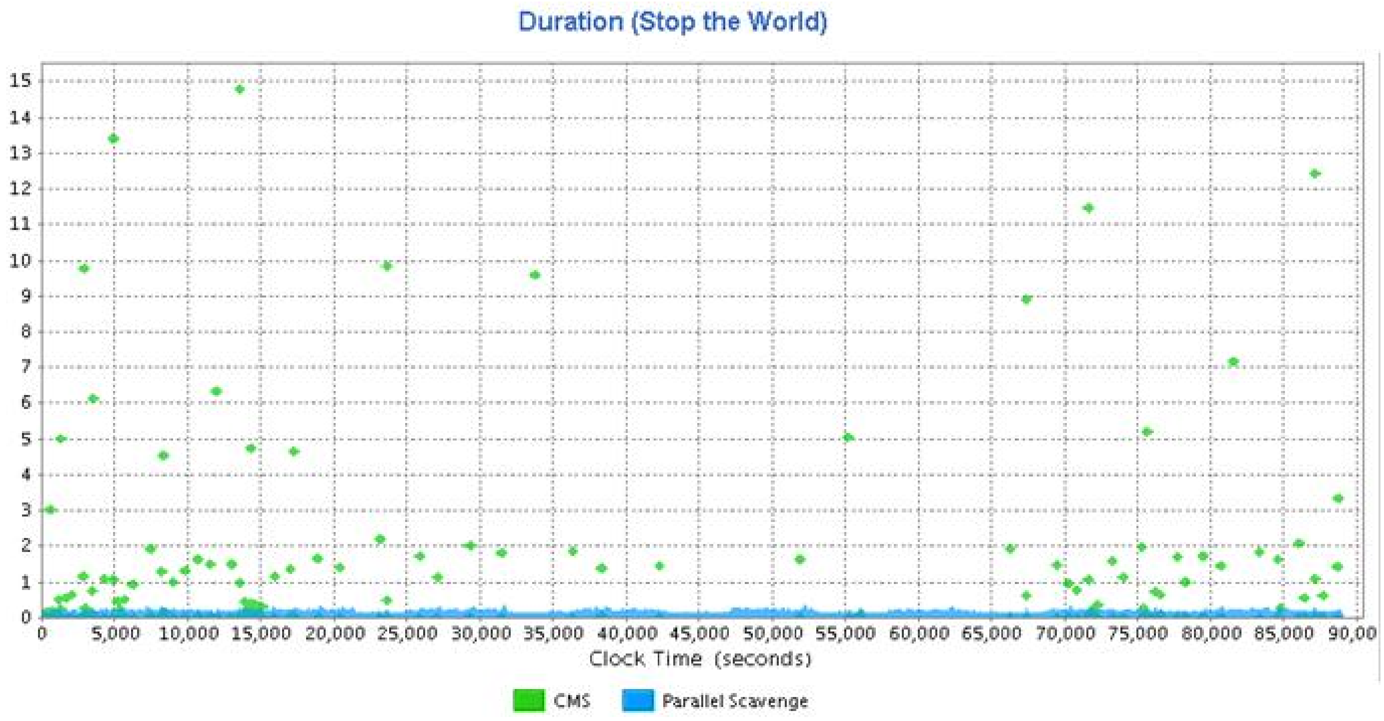


Figure 1: Duration Graph before GC Tuning.
The above graph, which shows the duration, is one of the graphs that the HPJMeter automatically provides after analysis. The **X-axis** shows the time after the JVM has started and the **Y-axis** shows the response time of each GC. The green dots, the CMS, indicates the Full GC result, and the blue bots, Parallel Scavenge, indicates the Minor GC result.

Previous I said that CMS GC would be the fastest. But the above result show that there were some cases which took up to 15 seconds. **What has caused such result?** Please remember what I said before: CMS gets slower when compaction is executed. In addition, the memory of the service has been set by using `-Xms1g` and `-Xmx4g` and the memory allocated was 4 GB.

So I changed the GC type from CMS GC to Parallel GC. I changed the memory size to 2 GB and then set the `NewRatio` to 3. The result of `jstat -gcutil` after a few hours is as follows.

```
S0 S1 E O P YGC YGCT FGC FGCT GCT
0.00 30.48 3.31 26.54 37.01 226 11.131 4 11.758 22.890
```

The Full GC time was faster, 3 seconds per one time, compared to 15 seconds for 4 GB. However, 3 seconds is still not so fast. So I created six cases as follows.

- Case 1: `-XX:+UseParallelGC -Xms1536m -Xmx1536m -XX:NewRatio=2`
- Case 2: `-XX:+UseParallelGC -Xms1536m -Xmx1536m -XX:NewRatio=3`
- Case 3: `-XX:+UseParallelGC -Xms1g -Xmx1g -XX:NewRatio=3`
- Case 4: `-XX:+UseParallelOldGC -Xms1536m -Xmx1536m -XX:NewRatio=2`
- Case 5: `-XX:+UseParallelOldGC -Xms1536m -Xmx1536m -XX:NewRatio=3`
- Case 6: `-XX:+UseParallelOldGC -Xms1g -Xmx1g -XX:NewRatio=3`

Which one would be the fastest? The result showed that the smaller the memory size was, the better the result was. The following figure shows the duration graph of Case 6, which showed the highest GC improvement. The slowest response time was 1.7 seconds and the average had been changed to within 1 second, showing the improved result.

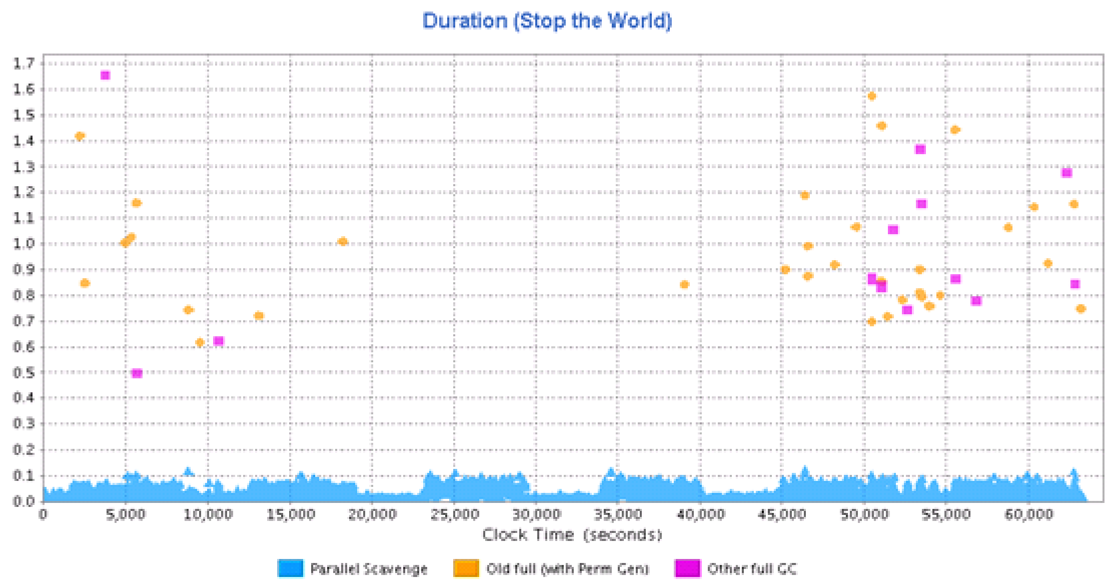


Figure 2: Duration Graph after Applying Case 6.

With the result, I changed all GC options of the service to Case 6. However, this change causes `OutOfMemoryError` at night each day. It is difficult to detail the reason here, but in short, batch data processing made a lack of JVM memory. The related problems are being cleared now.

It is very dangerous to analyze the GC logs accumulated for a short time and to apply the result to all servers as executing GC tuning. Keep in mind that GC tuning can be executed without failure **only** when you analyze the service operation as well as the GC logs.

We have reviewed two GC tuning examples to see how GC tuning is executed. As I mentioned, the GC option set in the examples can be identically set for the server which has the same CPU, OS version and JDK version with the service that executes the same functions. However, do not apply the option I did to your services in operation, since they may not work for you.

Conclusion

I execute GC tuning based on my experiences without executing heap dump and analyzing the memory in detail. Precise memory status analysis may draw the better GC tuning results. However, that kind of analysis may be helpful when the memory is used in the constant and routine pattern. But, if the service is heavily used and there are a lot of memory usage patterns, GC tuning based on reliable previous experience may be recommendable.

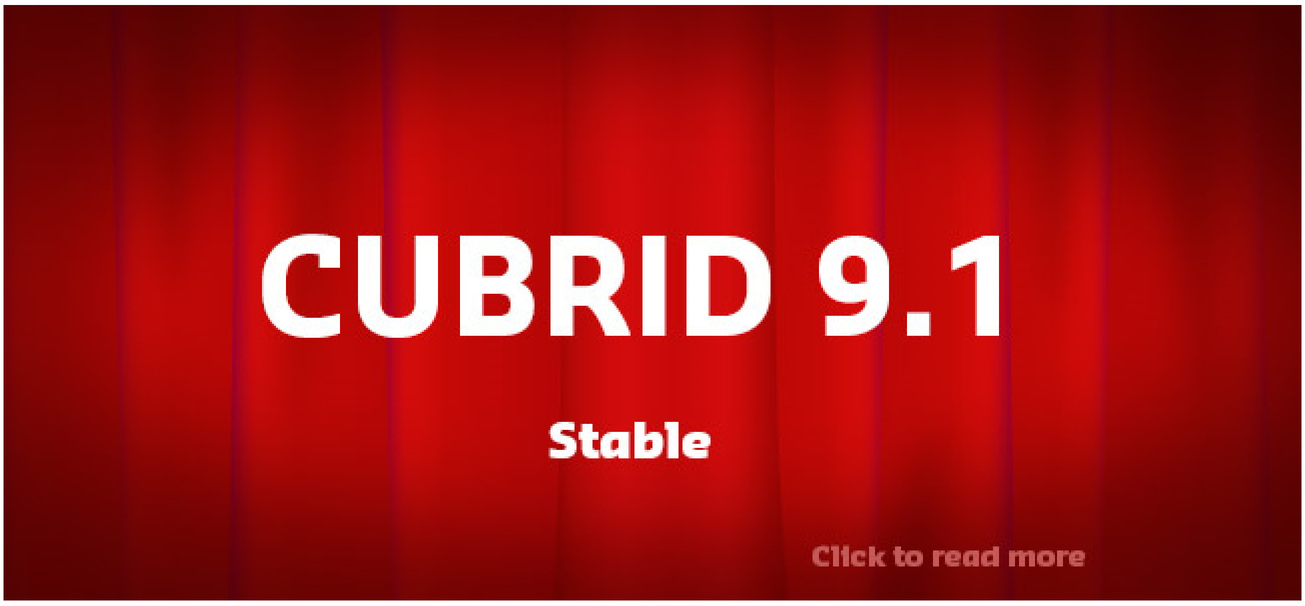
I have executed the performance test by setting the G1 GC option to some servers, but have not applied to any operation server yet. The G1 GC option shows a faster result than any other GC types. However, it requires to upgrade to JDK 7. In addition, stability is still not guaranteed. Nobody knows if there is any critical bug or not. So the time is not yet ripe for applying the option.

After JDK 7 is stabilized (this does not mean that it is not stable) and WAS is optimized for JDK 7, enabling stable application of G1 GC *may* finally work as expected and some day we *may* not need the GC tuning.

For more detail on GC tuning, search on Slideshare.com for related materials. The most recommendable material is Everything I Ever Learned About JVM Performance Tuning @Twitter, written by Attila Szegedi, a Twitter engineer. Please take the time to read it.

By Sangmin Lee, NHN Performance Engineering Lab.

> **About the author:**
> Joined NHN in 2009, Sangmin Lee works for support fault diagnosis, in-house lecture, and APM technical support and operating websites: tuning-java.com and GodOfJava.com. He has written several books on Java. He has written in his previous company "The story of custom coding which affects the Java performance and tuning", and "The story of testing that Java developers can learn easily with fun" and "The story of troubleshooting for Java developers and system operators" while commuting in a bus. Now, he is revising "Java standard".



MaxClients in Apache and its effect on Tomcat during Full GC

This is the *fourth* article in the series of "*Become a Java GC Expert*". In the first issue Understanding Java Garbage Collection we have learned about the processes for different GC algorithms, about how GC works, what Young and Old Generation is, what you should know about the 5 types of GC in the new JDK 7, and what the performance implications are for each of these GC types.

In the second article How to Monitor Java Garbage Collection we have explained how JVM actually runs the Garbage Collection in the real time, how we can monitor GC, and which tools we can use to make this process faster and more effective.

In the third article How to Tune Java Garbage Collection we have shown some of the best options based on real cases as our examples that you can use for GC tuning. Also we have explained how to minimize the number of objects passed to Old Area, decreasing Full GC time, as well as how to set GC type and the memory size.

In this fourth article I will explain the importance of `MaxClients` parameter in Apache that significantly affects the overall system performance when GC occurs. I will provide several examples through which you will understand the problem `MaxClients` value causes. I will also explain how to reliably set the proper value for `MaxClients` depending on the available system memory.

The effect of MaxClients on the system

The operation environment of NHN services has a variety of Throttle valve-type options. These options are important for reliable service operation. Let's see how the `MaxClients` option in Apache affects the system when Full GC has occurred in Tomcat.

Most developers know that "stop the world (STW) phenomenon" occurs when GC has occurred in Java (*for more refer to Understanding Java Garbage Collection*). In particular, Java developers at NHN may have experienced faults caused by GC-related issues in Tomcat. Because Java Virtual Machine (JVM) manages the memory, Java-based systems cannot be free of the STW phenomenon caused by GC.

Several times a day, GC occurs in services you have developed and currently operate. In this situation, even if TTS caused by faults does not occur, services may return unexpected 503 errors to users.

Service Operation Environment

For their structural characteristics, Web services are suitable for scale-out rather than scale-up. So, generally, physical equipment is configured with Apache * 1 + Tomcat * n according to equipment performance. However, this article assumes an environment where Apache * 1 + Tomcat * 1 are installed on one host as shown in **Figure 1** below for a convenient description.

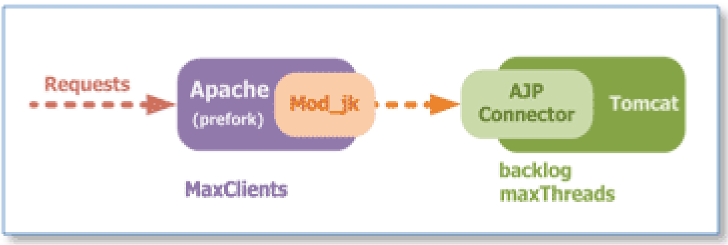


Figure 1: Service Operation Environment Assumed for the Article.
For reference, this article describes options in Apache 2.2.21 (prefork MPM), Tomcat 6.0.35, jdk 1.6.0_24 on CentOS 4.72 (32-bit) environment.

The total system memory is 2 GB and the Garbage Collector uses ParallelOldGC. The `AdaptiveSizePolicy` option is set to **true** by default and the heap size is set to 600m.

STW and HTTP 503

Let's assume that requests are flowing into Apache at 200 req/s and more than 10 httpd processes are running for service, even though this situation may depend on response time for requests. In this situation, assuming that the pause time at full GC is 1 second, **what will happen if full GC occurs in Tomcat?**

The first thing that hits your mind is that Tomcat will be paused by full GC without responding to all requests being processed. In this case, **what will happen to Apache while Tomcat is paused and requests are not processed?**

While Tomcat is paused, requests will continuously flow into Apache at 200 req/s. In general, before full GC occurs, responses to requests can be sent quickly by the service with only 10 or more httpd processes. However, because Tomcat is paused now, new httpd processes will continuously be created for new inflowing requests within the range allowed by the `MaxClients` parameter value of the **httpd.conf** file. As the default value is 256, it will not care that the requests are inflowing at 200 req/s.

At this time, **how about the newly created httpd processes?**

Httpd processes forwards requests to Tomcat by using the idle connections in the AJP connection pool managed by the **mod_jk** module. If there is no idle connection, it will request to create new connections. However, because Tomcat is paused, the request to create new connections will be rejected. Therefore, these requests will be queued in the backlog queue as many as the size of the backlog queue, set in the AJP Connector of the **server.xml** file, allows.

If the number of queued requests exceed the size of the backlog queue, a **Connection is Refused** error will be returned to Apache and Apache will return the **HTTP 503** error to users.

In the assumed situation, the default size of backlogs is 100 and the requests are flowing in at 200 req/s. Therefore, more than 100 requests will receive the 503 error for 1 second of the pause time caused by full GC.

In this situation, if full GC is over, sockets in the backlog queue are retrieved by Tomcat's acceptance and assigned to worker threads within the range allowed by `MaxThreads` (defaults to 200) in order to process requests.

MaxClients and backlog

In this situation, **which option should be set in order to prevent the 503 error to users?**

First, we need to understand that the backlog value should be enough to accept requests flowing to Tomcat during the pause time in full GC. In other words, it should be set to at least 200 or greater.

Now, **is there a problem in such configuration?**

Let's repeat the above situation under the assumption that the backlog setting value has been increased to 200. This result is more serious as shown below.

The system memory usage is typically 50%. However, it rapidly increases to almost 100% when full GC occurs, causing a rapid increase of swap memory usage. Moreover, because the pause time of full GC increases from 1 second to 4 or more seconds, the system is down for that time and cannot respond to any requests.

In the first situation, only 100 or more requests received the 503 error. However, after increasing the backlog size to 200, more than 500 requests will be hung for 3 or more seconds and cannot receive responses.

This situation is a good example that shows more serious situations which may occur when you do not precisely understand the organic relations between settings, i.e., their impact on the system.

Then, **why does this phenomenon occur?**

The reason is the characteristics of the `MaxClients` option.

Setting the `MaxClients` value to a generous value does not matter. The most important thing in setting the `MaxClients` option is that **the total memory usage should be calculated not to exceed 80%** even though httpd processes are created as much as the maximum `MaxClients` value.

The swappiness value of the system is set to 60 (default). As such, when memory usage exceeds 80%, swap will actively occur.

Let's see why this characteristic causes the more serious situation described above.

When requests are flowing in at 200 req/s and Tomcat is paused by full GC, the backlog setting value is 200. Approximately, an additional 100 httpd processes can be created in Apache above the first case. In this situation, when the total memory usage exceeds 80%, the OS will actively use the swap memory area, and objects for GC will be moved from the old area of JVM to the swap area since the OS considers them unused for a long period.

Finally, when the swap area is used in GC, the pause time will rapidly increase. So, the number of httpd processes will increase, causing 100% of memory usage and the situation previously described will occur.

The difference between the two cases is only the backlog setting values: 100 vs. 200. **Why did this situation occur only for 200?**

The reason for the difference is the number of httpd processes created in these configurations. When the setting value is set to 100 and, full GC occurs, 100 requests for creating new connections are created and then are queued in the backlog queue. The other requests receive the connection refused error message and return the 503 error. Therefore, the number of total httpd processes will be slightly more than 100.

When the value is set to 200, then 200 requests for creating new connections can be accepted. Therefore, the number of total httpd processes will be more than 200 and the value exceeds the threshold that determines the occurrence of memory swap.

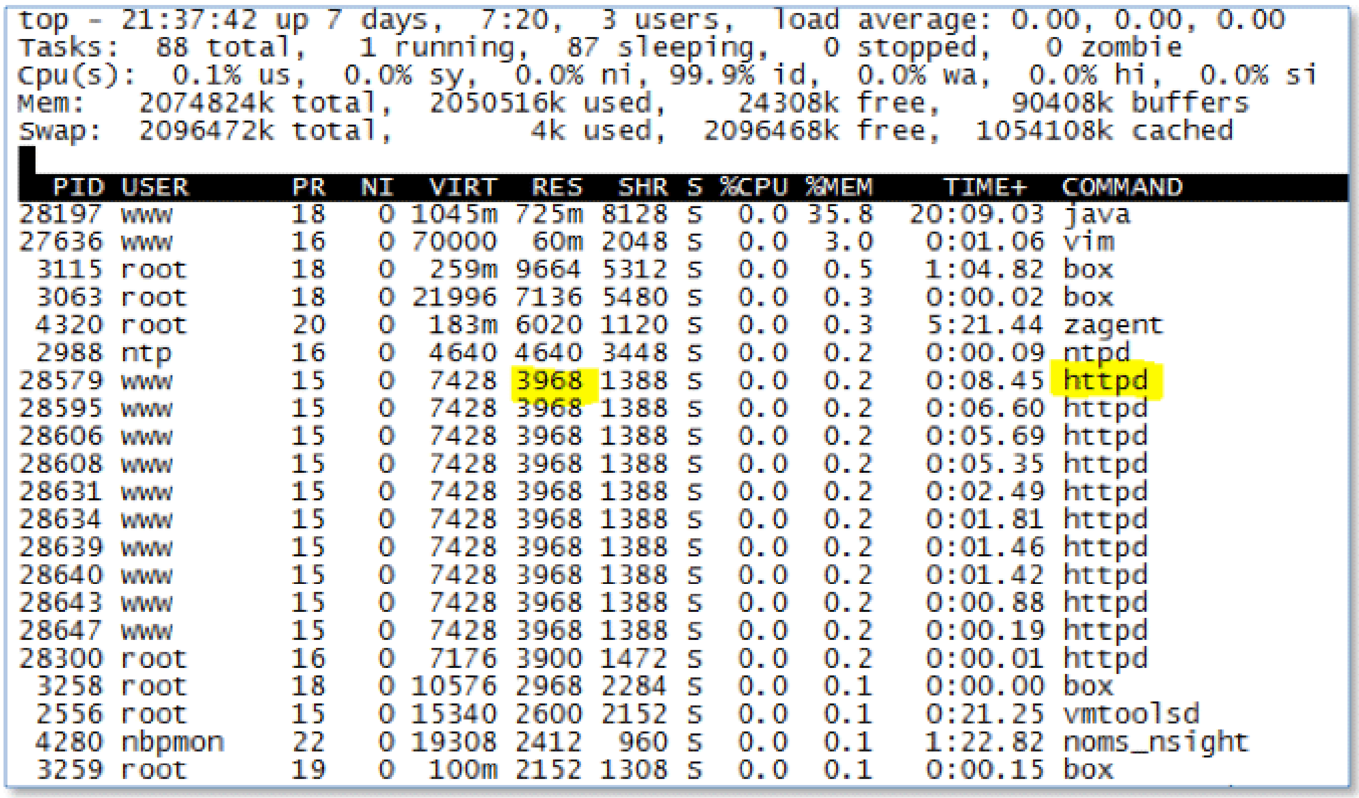
Then, by setting the `MaxClients` option without considering the memory usage, the number of httpd processes rapidly increases with full GC, causing swap and degradation of the system performance.

If so, **how can we determine the `MaxClients` value, what is the threshold value for the current system situation?**

Calculation Method of MaxClients Setting

As the total memory of the system is 2 GB, the `MaxClients` value should be set to use no more than 80% of the memory (1.6 GB) in any situation in order to prevent performance degradation caused by the memory swap. In other words, the 1.6 GB memory should be shared and allocated to Apache, Tomcat, and agent-type programs, which are installed by default.

Let's assume that the agent-type programs, which are installed in the system by default, occupy the memory at about 200 m. For Tomcat, the heap size set to `-Xmx` is 600m. Therefore, Tomcat will always occupy 725m (Perm Gen + Native Heap Area) based on the top RES (see the figure below). Finally, Apache can use 700m of the memory.



```
top - 21:37:42 up 7 days,  7:20,  3 users,  load average: 0.00, 0.00, 0.00
Tasks:  88 total,   1 running,  87 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.1% us,  0.0% sy,  0.0% ni, 99.9% id,  0.0% wa,  0.0% hi,  0.0% si
Mem:   2074824k total,  2050516k used,    24308k free,    90408k buffers
Swap:  2096472k total,        4k used,  2096468k free,  1054108k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
28197 www       18   0 1045m 725m 8128 S  0.0 35.8  20:09.03 java
27636 www       16   0 70000  60m 2048 S  0.0  3.0   0:01.06 vim
 3115 root      18   0  259m 9664 5312 S  0.0  0.5   1:04.82 box
 3063 root      18   0 21996 7136 5480 S  0.0  0.3   0:00.02 box
 4320 root      20   0  183m 6020 1120 S  0.0  0.3   5:21.44 zagent
 2988 ntp       16   0  4640 4640 3448 S  0.0  0.2   0:00.09 ntpd
28579 www       15   0  7428 3968 1388 S  0.0  0.2   0:08.45 httpd
28595 www       15   0  7428 3968 1388 S  0.0  0.2   0:06.60 httpd
28606 www       15   0  7428 3968 1388 S  0.0  0.2   0:05.69 httpd
28608 www       15   0  7428 3968 1388 S  0.0  0.2   0:05.35 httpd
28631 www       15   0  7428 3968 1388 S  0.0  0.2   0:02.49 httpd
28634 www       15   0  7428 3968 1388 S  0.0  0.2   0:01.81 httpd
28639 www       15   0  7428 3968 1388 S  0.0  0.2   0:01.46 httpd
28640 www       15   0  7428 3968 1388 S  0.0  0.2   0:01.42 httpd
28643 www       15   0  7428 3968 1388 S  0.0  0.2   0:00.88 httpd
28647 www       15   0  7428 3968 1388 S  0.0  0.2   0:00.19 httpd
28300 root      16   0  7176 3900 1472 S  0.0  0.2   0:00.01 httpd
 3258 root      18   0 10576 2968 2284 S  0.0  0.1   0:00.00 box
 2556 root      15   0 15340 2600 2152 S  0.0  0.1   0:21.25 vmtoolsd
 4280 nbpmon    22   0 19308 2412  960 S  0.0  0.1   1:22.82 noms_nsight
 3259 root      19   0  100m 2152 1308 S  0.0  0.1   0:00.15 box
```

Figure 2: Top Screen of Test System.

If so, **what should the value of `MaxClients` be with a memory of 700m?**

It will be different according to the type and the number of loaded modules. However, for NHN Web services, which use Apache as a simple proxy, 4m (based on top RES) will be enough for one httpd process (see

Figure 2). Therefore, the maximum `MaxClients` value for 700m should be 175.

Conclusion

Reliable service configuration should decrease the system downtime under overload and send successful responses to requests within the allowable range. For Java-based Web services, you must check whether the service has been configured to reliably respond to the STW under full GC.

If the `MaxClients` option is set to a large value, to respond to simple increase of user requests and against DDoS attacks, without considering the system memory usage, it loses its functionality as a throttle valve, causing bigger faults.

In this example, the best way to solve the problem is to expand the memory or the server, or set the `MaxClients` option to 175 (in the above case) so that Apache returns the 503 error to requests that only exceed 175.

> The situation in this article occurs within 3 to 5 seconds, so it cannot be checked by most monitoring tools which run at regular sampling intervals.

By Dongsoon Choi, Senior Engineer at Game Service Technical Support Team, NHN Corporation.

The Principles of Java Application Performance Tuning

This is the fifth article in the series of "Become a Java GC Expert". In the first issue Understanding Java Garbage Collection we have learned about the processes for different GC algorithms, about how GC works, what Young and Old Generation is, what you should know about the 5 types of GC in the new JDK 7, and what the performance implications are for each of these GC types.

In the second article How to Monitor Java Garbage Collection we have explained how JVM actually runs the Garbage Collection in the real time, how we can monitor GC, and which tools we can use to make this process faster and more effective.

In the third article How to Tune Java Garbage Collection we have shown some of the best options based on real cases as our examples that you can use for GC tuning. Also we have explained how to minimize the number of objects passed to Old Area, decreasing Full GC time, as well as how to set GC type and the memory size.

In the fourth article MaxClients in Apache and its effect on Tomcat during Full GC we have explained the importance of `MaxClients` parameter in Apache that significantly affects the overall system performance when GC occurs.

In this fifth article I will explain about the principles of Java application performance tuning. Specifically, I will explain what is required in order to tune the performance of Java application, the steps you need to perform to identify whether your application needs tuning. I will also explain the problems you may encounter during performance tuning. The article will be finalized with the recommendations you need to follow to make better decisions when tuning Java applications.

Overview

Not every application requires tuning. If an application performs as well as expected, you don't need to exert additional efforts to enhance its performance. However, it would be difficult to expect an application would reach its target performance as soon as it finishes debugging. This is when tuning is required. Regardless of the implementation language, tuning an application requires high expertise and concentration. Also, you may not use the same method for tuning a certain application to tune another application. This is because each application has its unique action and a different type of resource usage. For this reason, tuning an application requires more basic knowledge compared to the knowledge required to write an application. For example, you need knowledge on virtual machines, operating systems and computer architectures. When you focus on an application domain based on such knowledge, you can successfully tune an application.

Sometimes Java application tuning requires only changing JVM options, such as Garbage Collector, but sometimes it requires changing the application source code. Whichever method you choose, you need to monitor the process of executing the Java application first. For this reason, the issues this article will deal with are as follows:

- **How can I monitor a Java application?**
- **What JVM options should I give?**

- **How can I know if modifying source codes is required or not?**

Knowledge Required to Tune the Performance of Java Applications

Java applications operate inside Java Virtual Machine (JVM). Therefore, to tune a Java application, you need to understand the JVM operation process. I have previously blogged about Understanding JVM Internals where you can find great insights about JVM.

The knowledge regarding the process of the operation of JVM in this article mainly refers to the knowledge of Garbage Collection (GC) and Hotspot. Although you may not be able to tune the performance of all kinds of Java applications only with the knowledge on GC or Hotspot, these two factors influence the performance of Java applications in most cases.

It is noted that from the perspective of an operating system JVM is also an application process. To make an environment in which a JVM can operate well, you should understand how an OS allocates resources to processes. This means, to tune the performance of Java applications, you should have an understanding of OS or hardware as well as JVM itself.

Another aspect is that knowledge of Java language domain is also important. It is also important to understand lock or concurrency and to be familiar with class loading or object creation.

When you carry out Java application performance tuning, you should approach it by integrating all this knowledge.

The Process of Java Application Performance Tuning

Figure 1 shows a flow chart from the book <Java Performance> co-authored by Charlie Hunt and Binu John. This chart shows the process of Java application performance tuning.

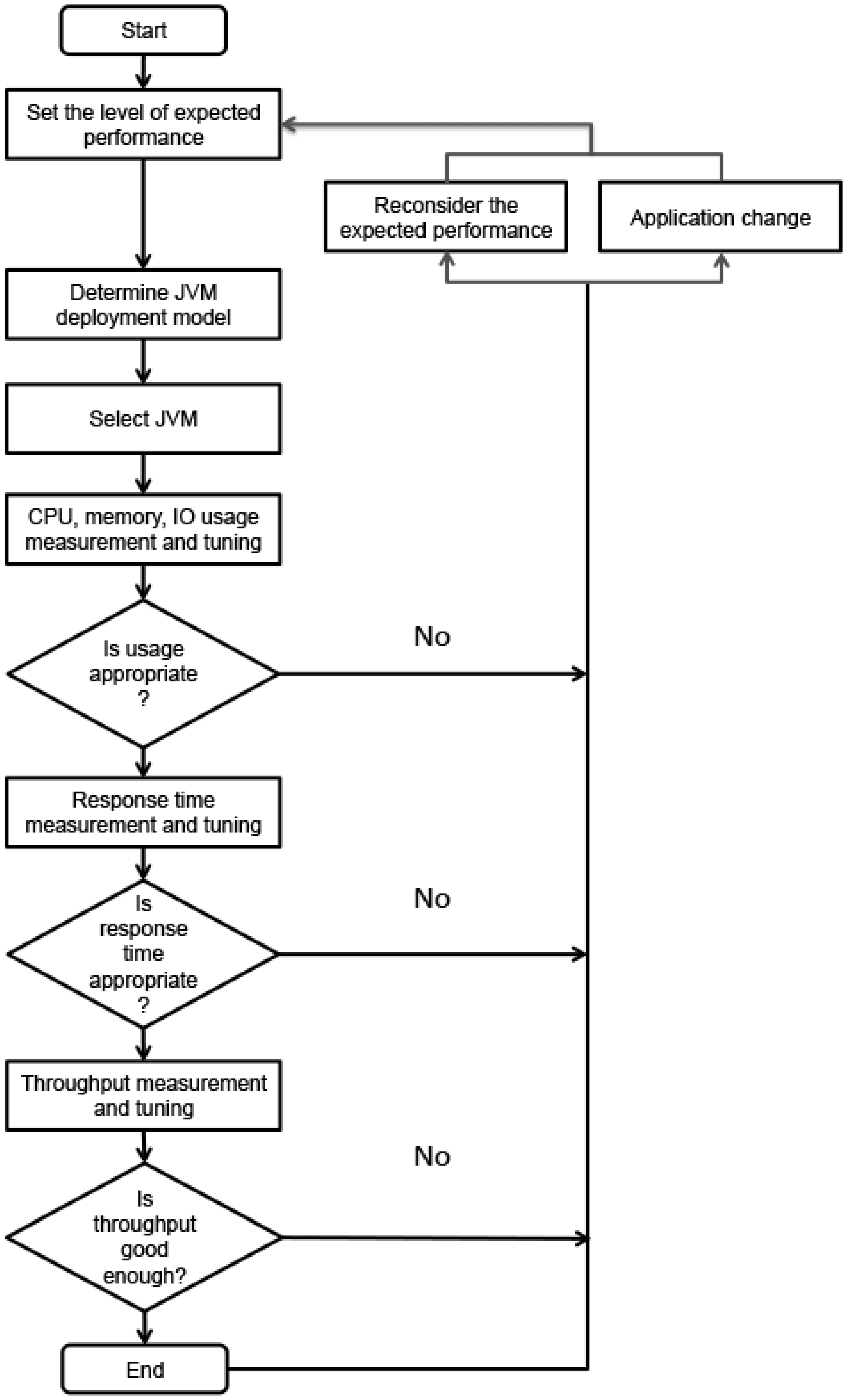

Start
Set the level of expected performance
Reconsider the expected performance
Application change
Determine JVM deployment model
Select JVM
CPU, memory, IO usage measurement and tuning
Is usage appropriate ?
No
Response time measurement and tuning
Is response time appropriate ?
No
Throughput measurement and tuning
No
Is throughput good enough?
End

Figure 1: The Process of Tuning the Performance of Java Applications.
The above process is not a one-time process. You may need to repeat it until the tuning is completed. This also applies to determining an expected performance value. In the process of tuning, sometimes you should lower the expected performance value, and sometimes raise it.

JVM distribution model

A **JVM distribution model** is related with making a decision on whether to operate Java applications on a single JVM or to operate them on multiple JVMs. You can decide it according to its availability, responsiveness and maintainability. When operating JVM on multiple servers, you can also decide whether to run multiple JVMs on a single server or to run a single JVM per server. For example, for each server, you can decide whether to run a single JVM using a heap of 8 GB, or to use four JVMs each using a heap of 2 GB. Of course, you can decide the number of JVMs running on a single server depending on the number of cores and the characteristics of the application. When comparing the two settings in terms of responsiveness, it might be more advantageous to use a heap of 2 GB rather than 8 GB for the same application, for it takes shorter to perform a full garbage collection when using a heap of 2 GB. If you use a heap of 8 GB, however, you can reduce the frequency of full GCs. You can also improve responsiveness by increasing the hit rate if the application uses internal cache. Therefore, you can choose a suitable distribution model by taking into account the characteristics of the application and the method to overcome the disadvantage of the model you chose for some advantages.

JVM architecture

Selecting a JVM means whether to use a **32-bit JVM** or a **64-bit JVM**. Under the same conditions, you had better choose a 32-bit JVM. This is because a 32-bit JVM performs better than a 64-bit JVM. However, the maximum logical heap size of a 32-bit JVM is 4 GB. (However, actual allocatable size for both 32-bit OS and 64-bit OS is 2-3 GB.) It is appropriate to use a 64-bit JVM when a heap size larger than this is required.

Table 1: Performance Comparison (source).

Benchmark	Time (sec)	Factor
C++ Opt	23	1.0x
C++ Dbg	197	8.6x
Java 64-bit	134	5.8x
Java 32-bit	290	12.6x
Java 32-bit GC*	106	4.6x
Java 32-bit SPEC GC*	89	3.7x
Scala	82	3.6x
Scala low-level*	67	2.9x
Scala low-level GC*	58	2.5x
Go 6g	161	7.0x
Go Pro*	126	5.5x

The next step is to run the application and to measure its performance. This process includes tuning GC, changing OS settings and modifying codes. For these tasks, you can use a system monitoring tool or a profiling tool.

It should be noted that tuning for responsiveness and tuning for throughput could be different approaches. Responsiveness will be reduced if stop-the-world occurs from time to time, for example, for a full garbage collection despite a large amount of throughput per unit time. You also need to consider that a trade-off could occur. Such trade-off could occur not only between responsiveness and throughput. You may need to use more CPU resources to reduce memory usage or put up with reduction in responsiveness or throughput. As opposite cases could likewise occur, you need to approach it according to the priority.

The flow chart of **Figure 1** above shows the performance tuning approach for almost all kinds of Java applications, including Swing applications. However, this chart is somewhat unsuitable for writing a server application for Internet service as our company NHN does. The flow chart in **Figure 2** below is a simpler procedure designed based on **Figure 1** to be more suitable for NHN.

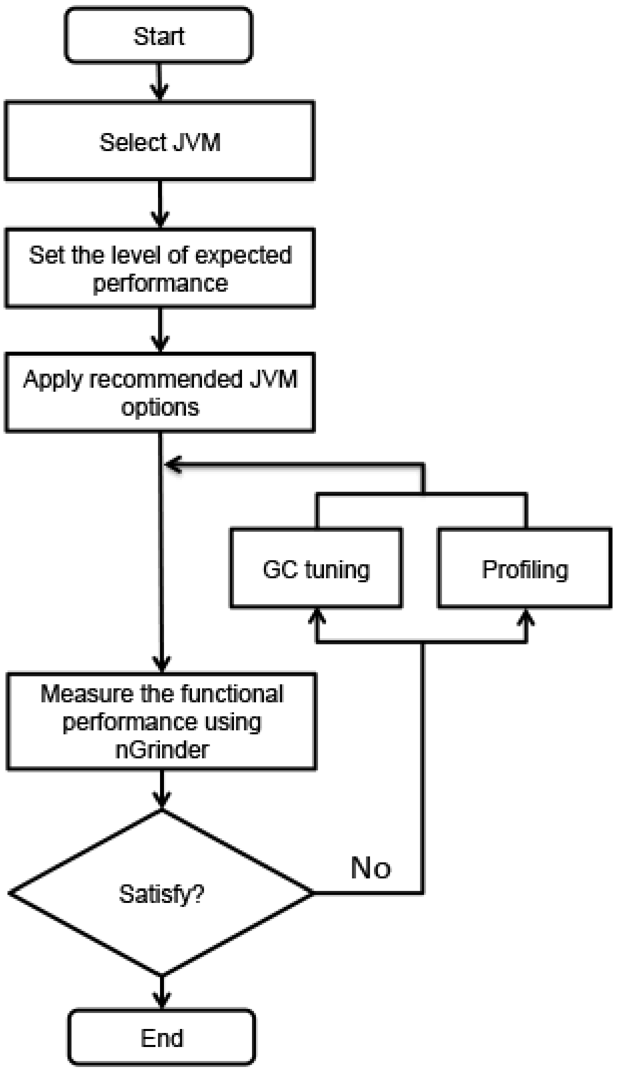

Start
Select JVM
Set the level of expected performance
Apply recommended JVM options
GC tuning
Profiling
Measure the functional performance using nGrinder
Satisfy?
No
End

Figure 2: A Recommended Procedure for Tuning NHN's Java Applications.

Select JVM in the above flow chart means using a 32-bit JVM as much as possible except when you need to use a 64-bit JVM to maintain cache of several GB.

Now, based on the flow chart in **Figure 2**, you will learn about things to do to execute each of the steps.q

JVM Options

I will explain how to specify suitable JVM options mainly for a web application server. Despite not being applied to every case, the **best GC algorithm**, especially for web server applications, is the Concurrent Mark Sweep GC. This is because what matters is **low latency**. Of course, when using the Concurrent Mark Sweep, sometimes a very long stop-the-world phenomenon could take place due to fractions. Nevertheless, this problem is likely to be resolved by adjusting the new area size or the fraction ratio.

Specifying the **new area size** is as important as specifying the **entire heap size**. You had better specify the ratio of the new area size to the entire heap size by using `-XX:NewRatio` or specify the desired new area size by using the `-XX:NewSize` option. Specifying a new area size is important because most objects cannot survive long. In web applications, most objects, except cache data, are generated when `HttpResponse` to `HttpRequest` is created. This time hardly exceeds a second. This means the life of objects does not exceed a second, either. If the new area size is not large, it should be moved to the old area to make space for newly created objects. The cost for GC for the old area is much bigger than that for the new area; therefore, it is good to set the size of the new area sufficiently.

If the new area size exceeds a certain level, however, responsiveness will be reduced. This is because the garbage collection for the new area is basically to copy data from one survivor area to another survivor area. Also, the stop-the-world phenomenon will occur even when performing GC for the new area as well as the old area. If the new area becomes bigger, the survivor area size will increase, and thus the size of the data to copy will increase as well. Given such characteristics, it is good to set a suitable new area size by referring to the `NewRatio` of HotSpot JVM by OS.

Table 2: NewRatio by OS and option.

OS and option	Default -XX:NewRatio
Sparc -server	2
Sparc -client	8
x86 -server	8
x86 -client	12

If the `NewRatio` is specified, `1/(NewRatio +1)` of the entire heap size becomes the new area size. You will find the `NewRatio` of **Sparc -server** is very small. This is because the Sparc system was used for more high-end use than x86 when default values were specified. Now it is common to use the x86 server and its performance has also been improved. Thus it is better to specify 2 or 3, which is the value similar to that of the **Sparc -server**.

You can also specify `NewSize` and `MaxNewSize` instead of `NewRatio`. The new area is created as much as the value specified for `NewSize` and the size increments as much as the value specified for `MaxNewSize`. The Eden or Survivor area also increases according to the (specified or default) ratio. As you specify the same size for `-Xs` and `-Xmx`, it is a very good choice to specify the same size for `MaxSize` and `MaxNewSize`.

If you have specified both `NewRatio` and `NewSize`, you should use the bigger one. Therefore, when a heap has been created, you can express the initial New area size as follows:

```
min(MaxNewSize, max(NewSize, heap/(NewRatio+1)))
```

However, it is impossible to determine the appropriate entire heap size and New area size in a single attempt. Based on my experience running Web server applications at NHN, I recommend to run Java applications with the following JVM options. After monitoring the performance of the application with these options, you can use a more suitable GC algorithm or options.

Table 3: Recommended JVM options.

Type	Option
Operation mode	`-sever`
Entire heap size	Specify the same value for `-Xms` and `-Xmx`.
New area size	`-XX:NewRatio`: value of 2 to 4 `-XX:NewSize=?` `–XX:MaxNewSize=?`. Also good to specify `NewSize` instead of `NewRatio`.
Perm size	`-XX:PermSize=256 m` `-XX:MaxPermSize=256 m`. Specify the value to an extent not to cause any trouble in the operation because it does not affect the performance.
GC log	`-Xloggc:$CATALINA_BASE/logs/gc.log` `-XX:+PrintGCDetails` `-XX:+PrintGCDateStamps`. Leaving a GC log does not particularly affect the performance of Java applications. You are recommended to leave a GC log as much as possible.
GC algorithm	`-XX:+UseParNewGC` `-XX:+CMSParallelRemarkEnabled` `-XX:+UseConcMarkSweepGC` `-XX:CMSInitiatingOccupancyFraction=75`.This is only a generally recommendable configuration. Other choices could be better depending on the characteristics of the application.
Creating a heap dump when an OOM error occurs	`-XX:+HeapDumpOnOutOfMemoryError` `-XX:HeapDumpPath=$CATALINA_BASE/logs`
Actions after an OOM occurs	`-XX:OnOutOfMemoryError=$CATALINA_HOME/bin/stop.sh` or `-XX:OnOutOfMemoryError=$CATALINA_HOME/bin/restart.sh`. After leaving a heap dump, take a proper operation according to a management policy.

Measuring the Performance of Applications

The information to acquire to grasp the performance of an application is as follows:

- **TPS (OPS):** The information required to understand the performance of an application conceptually.
- **Request Per Second (RPS):** Strictly speaking, RPS is different from responsiveness, but you can understand it as responsiveness. Through RPS, you can check the time it takes for the user to see the result.
- **RPS Standard Deviation:** It is necessary to induce even RPS if possible. If a deviation occurs, you need to check GC tuning or interworking systems.

To obtain a more accurate performance result, you should measure it after warming up the application sufficiently. This is because byte code is expected to be compiled by HotSpot JIT. In general, you can measure actual performance values after applying load to a certain feature for at least 10 minutes by using nGrinder load testing tool.

Tuning in Earnest

You don't need to tune the performance of an application if the result of the execution of nGrinder meets the expectation. If the performance does not meet the expectation, you need to carry out tuning to resolve problems. Now you will see the approach by case.

In the event the Stop-the-World takes long

Long **stop-the-world** time could result from inappropriate GC options or incorrect implementation. You can decide the cause according to the result of a profiler or a heap dump. This means you can judge the cause after checking the type and number of objects of a heap. If you find many unnecessary objects, you had better modify source codes. If you find no particular problem in the process of creating objects, you had better simply change GC options.

To adjust GC options appropriately, you need to have GC log secured for a sufficient period of time. You need to understand in which situation the stop-the-world takes a long time. For more information on the selection of appropriate GC options, read my colleague's blog about How to Monitor Java Garbage Collection.

In the event CPU usage rate is low

When blocking time occurs, both TPS and CPU usage rate will decrease. This might result from the problem of interworking systems or concurrency. To analyze this, you can use an analysis on the result of thread dump or a profiler. For more information on thread dump analysis, read How to Analyze Java Thread Dumps.

You can conduct a very accurate lock analysis by using a commercial profiler. In most cases, however, you can obtain a satisfactory result with only the CPU analyzer in **jvisualvm**.

In the event CPU usage rate is high

If TPS is low but CPU usage rate is high, this is likely to result from inefficient implementation. In this case, you should find out the location of bottlenecks by using a profiler. You can analyze this by using **jvisuavm**, **TPTP** of Eclipse or **JProbe**.

Approach for Tuning

You are advised to use the following approach to tune applications.

First, you should check whether performance tuning is necessary. The process of performance measuring is not easy work. You are also not guaranteed to obtain a satisfactory result all the time. Therefore, if the application already meets its target performance, you don't need to invest additionally in performance.

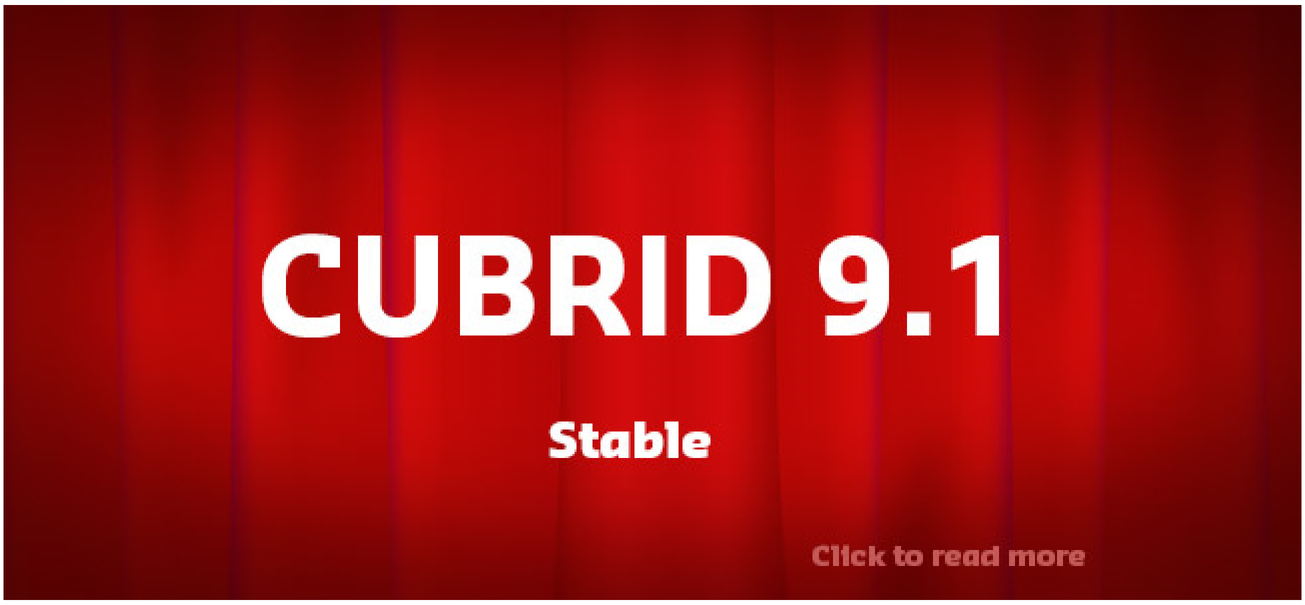
The problem lies in only a single place. All you have to do is to fix it. The Pareto principle applies to performance tuning as well. This does not mean to emphasize that the low performance of a certain feature results necessarily from a single problem. Rather, this emphasizes that we should focus on one factor that has the biggest influence on the performance when approaching performance tuning. Thus, you could handle another problem after fixing the most important one. You are advised to try to fix just one problem at a time.

You should consider the balloon effect. You should decide what to give up to get something. You can improve responsiveness by applying cache but if the cache size increases, the time it takes to carry out a full GC will increase as well. In general, if you want a small amount of memory usage, throughput or responsiveness could be deteriorated. Thus, you need to consider what is most important and what is less important.

So far, you have read the method for Java application performance tuning. To introduce a concrete procedure for performance measurement, I had to omit some details. Nevertheless, I think this could satisfy most of the cases for tuning Java web server applications.

Good luck with performance tuning!

By Se Hoon Park, Senior Software Engineer at Web Platform Development Lab, NHN Corporation.



Understanding Java Garbage Collection | CUBRID Blog

What are the benefits of knowing how garbage collection (GC) works in Java? Satisfying the intellectual curiosity as a software engineer would be a valid cause, but also, understanding how GC works can help you write much better Java applications.

This is a very personal and subjective opinion of mine, but I believe that a person well versed in GC tends to be a better Java developer. If you are interested in the GC process, that means you have experience in developing applications of certain size. If you have thought carefully about choosing the right GC algorithm, that means you completely understand the features of the application you have developed. Of course, this may not be common standards for a good developer. However, few would object when I say that understanding GC is a requirement for being a great Java developer.

This is the first of a series of "*Become a Java GC Expert*" articles. I will cover the *GC introduction* this time, and in the next article, I will talk about analyzing GC status and GC tuning examples from NHN.

The purpose of this article is to introduce GC to you in an easy way. I hope this article proves to be very helpful. Actually, my colleagues have already published a few great articles on Java Internals which became quite popular on Twitter. You may refer to them as well.

Returning back to Garbage Collection, there is a term that you should know before learning about GC. The term is "**stop-the-world**." Stop-the-world will occur no matter which GC algorithm you choose. *Stop-the-world* means that the JVM is stopping the application from running to execute a GC. When stop-the-world occurs, every thread except for the threads needed for the GC will stop their tasks. The interrupted tasks will resume only after the GC task has completed. GC tuning often means reducing this stop-the-world time.

Generational Garbage Collection

Java does not explicitly specify a memory and remove it in the program code. Some people sets the relevant object to null or use System.gc() method to remove the memory explicitly. Setting it to null is not a big deal, but calling System.gc() method will affect the system performance drastically, and must not be carried out. (Thankfully, I have not yet seen any developer in NHN calling this method.)

In Java, as the developer does not explicitly remove the memory in the program code, the garbage collector finds the unnecessary (garbage) objects and removes them. This garbage collector was created based on the following two hypotheses. (It is more correct to call them suppositions or preconditions, rather than hypotheses.)

- Most objects soon become unreachable.
- References from old objects to young objects only exist in small numbers.

These hypotheses are called the **weak generational hypothesis**. So in order to preserve the strengths of this hypothesis, it is physically divided into two - **young generation** and **old generation** - in HotSpot VM.

Young generation: Most of the newly created objects are located here. Since most objects soon become unreachable, many objects are created in the young generation, then disappear. When objects disappear from this area, we say a "**minor GC**" has occurred.

Old generation: The objects that did not become unreachable and survived from the young generation are copied here. It is generally larger than the young generation. As it is bigger in size, the GC occurs less frequently than in the young generation. When objects disappear from the old generation, we say a "**major GC**" (or a "**full GC**") has occurred.

Let's look at this in a chart.

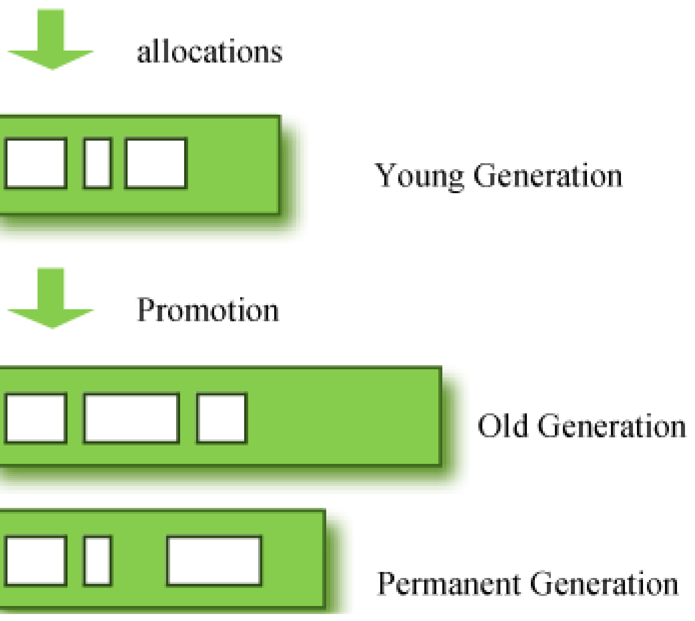


Figure 1: GC Area & Data Flow.

The **permanent generation** from the chart above is also called the "**method area,**" and it stores classes or interned character strings. So, this area is definitely not for objects that survived from the old generation to stay permanently. A GC may occur in this area. The GC that took place here is still counted as a major GC.

Some people may wonder:

> **What if an object in the old generation need to reference an object in the young generation?**

To handle these cases, there is something called the a "**card table**" in the old generation, which is a *512 byte chunk*. Whenever an object in the old generation references an object in the young generation, it is recorded in this table. When a GC is executed for the young generation, only this card table is searched to determine whether or not it is subject for GC, instead of checking the reference of all the objects in the old generation. This card table is managed with **write barrier**. This *write barrier* is a device that allows a faster performance for minor GC. Though a bit of overhead occurs because of this, the overall GC time is reduced.

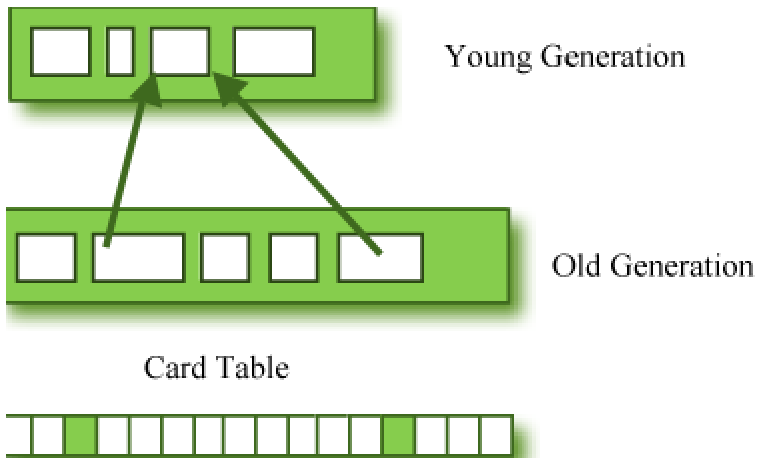


Figure 2: Card Table Structure.

Composition of the Young Generation

In order to understand GC, let's learn about the young generation, where the objects are created for the first time. The young generation is divided into 3 spaces.

- One **Eden** space
- Two **Survivor** spaces

There are 3 spaces in total, two of which are Survivor spaces. The order of execution process of each space is as below:

1. The majority of newly created objects are located in the Eden space.
2. After one GC in the Eden space, the surviving objects are moved to one of the Survivor spaces.
3. After a GC in the Eden space, the objects are piled up into the Survivor space, where other surviving objects already exist.
4. Once a Survivor space is full, surviving objects are moved to the other Survivor space. Then, the Survivor space that is full will be changed to a state where there is no data at all.
5. The objects that survived these steps that have been repeated a number of times are moved to the old generation.

As you can see by checking these steps, one of the Survivor spaces must remain empty. If *data exists in both Survivor spaces, or the usage is 0 for both spaces*, then take that as a sign that **something is wrong with your system**.

The process of data piling up into the old generation through minor GCs can be shown as in the below chart:

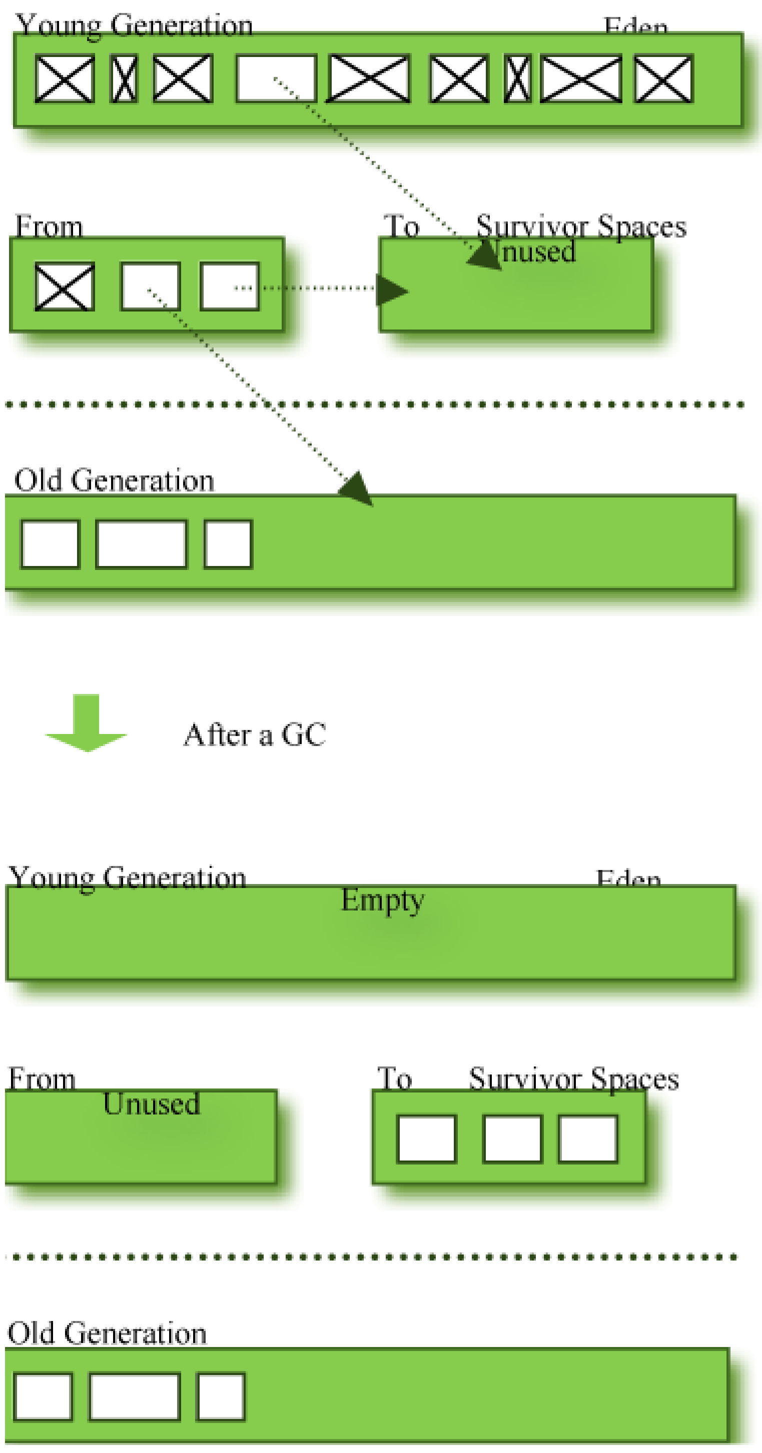


Figure 3: Before & After a GC.

Note that in HotSpot VM, two techniques are used for faster memory allocations. One is called "**bump-the-pointer**," and the other is called "**TLABs (Thread-Local Allocation Buffers)**."

Bump-the-pointer technique tracks the last object allocated to the Eden space. That object will be located on top of the Eden space. And if there is an object created afterwards, it checks only if the size of the object is suitable for the Eden space. If the said object seems right, it will be placed in the Eden space, and the new object goes on top. So, when new objects are created, only the lastly added object needs to be checked, which allows much faster memory allocations. However, it is a different story if we consider a multithreaded environment. To save objects used by multiple threads in the Eden space for Thread-Safe, an inevitable lock will occur and the performance will drop due to the lock-contention. **TLABs** is the solution to this problem in HotSpot VM. This allows each thread to have a small portion of its Eden space that corresponds to its own share. As each thread can only access to their own TLAB, even the bump-the-pointer technique will allow memory allocations without a lock.

This has been a quick overview of the GC in the young generation. You do not necessarily have to remember the two techniques that I have just mentioned. You will not go to jail for not knowing them. But please remember that after the objects are first created in the Eden space, and the long-surviving objects are moved to the old generation through the Survivor space.

GC for the Old Generation

The old generation basically performs a GC when the data is full. The execution procedure varies by the GC type, so it would be easier to understand if you know different types of GC.

According to JDK 7, there are 5 GC types.

1. Serial GC
2. Parallel GC
3. Parallel Old GC (Parallel Compacting GC)
4. Concurrent Mark & Sweep GC (or "CMS")
5. Garbage First (G1) GC

Among these, the **serial GC must not be used on an operating server**. This GC type was created when there was only one CPU core on desktop computers. Using this serial GC will drop the application performance significantly.

Now let's learn about each GC type.

Serial GC (-XX:+UseSerialGC)

The GC in the young generation uses the type we explained in the previous paragraph. The GC in the old generation uses an algorithm called "**mark-sweep-compact**."

1. The first step of this algorithm is to mark the surviving objects in the old generation.
2. Then, it checks the heap from the front and leaves only the surviving ones behind (sweep).
3. In the last step, it fills up the heap from the front with the objects so that the objects are piled up consecutively, and divides the heap into two parts: one with objects and one without objects (compact).

The serial GC is suitable for a small memory and a small number of CPU cores.

Parallel GC (-XX:+UseParallelGC)

Figure 4: Difference between the Serial GC and Parallel GC.
From the picture, you can easily see the difference between the serial GC and parallel GC. While the serial GC uses only one thread to process a GC, the parallel GC uses several threads to process a GC, and therefore, faster. This GC is useful when there is enough memory and a large number of cores. It is also called the "**throughput GC**."

Parallel Old GC(-XX:+UseParallelOldGC)

Parallel Old GC was supported since JDK 5 update. Compared to the parallel GC, the only difference is the GC algorithm for the old generation. It goes through three steps: *mark – summary – compaction*. The summary step identifies the surviving objects separately for the areas that the GC have previously performed, and thus different from the sweep step of the mark-sweep-compact algorithm. It goes through a little more complicated steps.

CMS GC (-XX:+UseConcMarkSweepGC)

Figure 5: Serial GC & CMS GC.
As you can see from the picture, the Concurrent Mark-Sweep GC is much more complicated than any other GC types that I have explained so far. The early *initial mark* step is simple. The surviving objects among the objects the closest to the classloader are searched. So, the pausing time is very short. In the *concurrent mark* step, the objects referenced by the surviving objects that have just been confirmed are tracked and checked. The difference of this step is that it proceeds while other threads are processed at the same time. In the *remark* step, the objects that were newly added or stopped being referenced in the concurrent mark step are checked. Lastly, in the *concurrent sweep* step, the garbage collection procedure takes place. The garbage collection is carried out while other threads are still being processed. Since this GC type is performed in this manner, the pausing time for GC is very short. The CMS GC is also called the low latency GC, and is **used when the response time from all applications is crucial**.

While this GC type has the advantage of short stop-the-world time, it also has the following disadvantages.

- It uses more memory and CPU than other GC types.
- The compaction step is not provided by default.

You need to carefully review before using this type. Also, if the compaction task needs to be carried out because of the many memory fragments, the stop-the-world time can be longer than any other GC types. You need to check how often and how long the compaction task is carried out.

G1 GC

Finally, let's learn about the garbage first (G1) GC.

Figure 6: Layout of G1 GC.
If you want to understand G1 GC, forget everything you know about the young generation and the old generation. As you can see in the picture, one object is allocated to each grid, and then a GC is executed. Then, once one area is full, the objects are allocated to another area, and then a GC is executed. The steps where the data moves from the three spaces of the young generation to the old generation cannot be found in this GC type. This type was created to replace the CMS GC, which has causes a lot of issues and complaints in the long term.

The biggest advantage of the G1 GC is its **performance**. It is faster than any other GC types that we have discussed so far. But in JDK 6, this is called an *early access* and can be used only for a test. It is officially included in JDK 7. In my personal opinion, we need to go through a long test period (at least 1 year) before NHN can use JDK7 in actual services, so you probably should wait a while. Also, I heard a few times that a JVM crash occurred after applying the G1 in JDK 6. Please wait until it is more stable.

I will talk about the **GC tuning** in the next issue, but I would like to ask you one thing in advance. If the size and the type of all objects created in the application are identical, all the GC options for WAS used in our company can be the same. But the size and the lifespan of the objects created by WAS vary depending on the service, and the type of equipment varies as well. In other words, just because a certain service uses the GC option "A," it does not mean that the same option will bring the best results for a different service. It is necessary to find the best values for the WAS threads, WAS instances for each equipment and each GC option by constant tuning and monitoring. This did not come from my personal experience, but from the discussion of the engineers making Oracle JVM for JavaOne 2010.

In this issue, we have only glanced at the GC for Java. Please look forward to our next issue, where I will talk about **how to monitor the Java GC status and tune GC**.

I would like to note that I referred to a new book released in December 2011 called "*Java Performance*" (Amazon, it can also be viewed from safari online, if the company provides an account), as well as “*Memory Management in the Java HotSpotTM Virtual Machine*,” a white paper provided by the Oracle website. (The book is different from "*Java Performance Tuning*.")

By Sangmin Lee, Senior Engineer at Performance Engineering Lab, NHN Corporation.