# [Understanding Java Garbage Collection](http://www.cubrid.org/blog/dev-platform/understanding-java-garbage-collection/)

What are the benefits of knowing how garbage collection (GC) works in [Java](http://www.cubrid.org/blog/tags/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*](http://www.cubrid.org/blog/tags/Garbage%20Collection/)" 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](http://www.cubrid.org/blog/tags/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](http://www.cubrid.org/blog/tags/Java/) 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](http://www.cubrid.org/blog/dev-platform/understanding-jvm-internals/) 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.

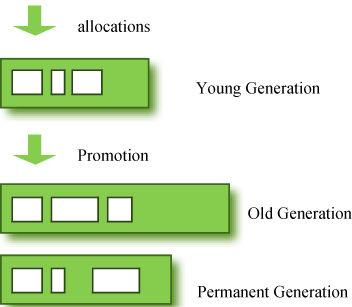
References from old objects to young objects only exist in small numbers.

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.



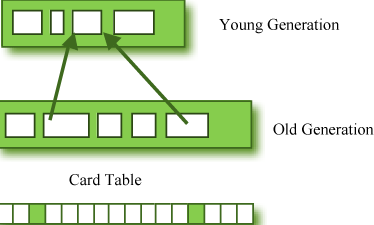
**Figure 1: GC Area & Data Flow.**

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

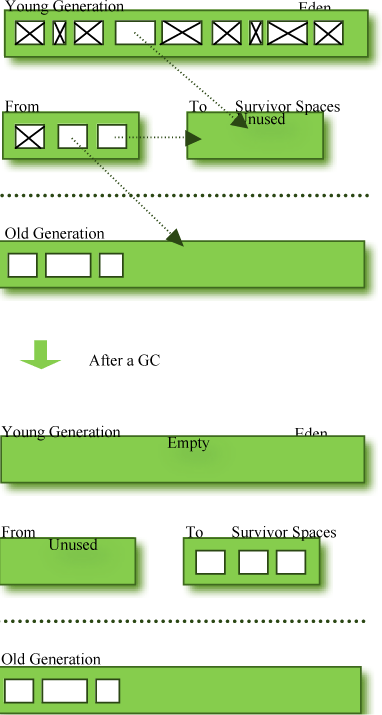
Two **Survivor** spaces

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.

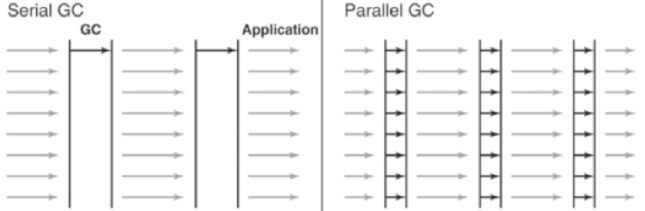
### Serial GC (-XX:+UseSerialGC)

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)



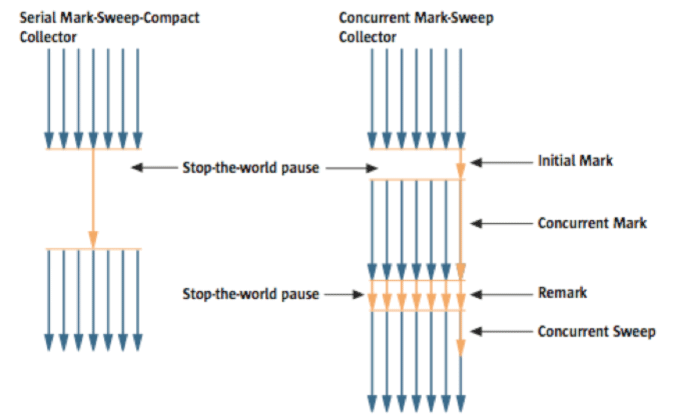
**Figure 4: Difference between the Serial GC and Parallel GC.**

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**.

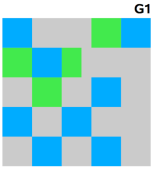
While this GC type has the advantage of short stop-the-world time, it also has the following disadvantages.

* 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**.

# [The Principles of Java Application Performance Tuning](http://www.cubrid.org/blog/dev-platform/the-principles-of-java-application-performance-tuning/)

This is the fifth article in the series of "[Become a Java GC Expert](http://www.cubrid.org/blog/tags/Garbage%20Collection/)". In the first issue [Understanding Java Garbage Collection](http://www.cubrid.org/blog/dev-platform/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](http://www.cubrid.org/blog/dev-platform/how-to-monitor-java-garbage-collection/) we have explained how [JVM](http://www.cubrid.org/blog/dev-platform/understanding-jvm-internals/) 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](http://www.cubrid.org/blog/dev-platform/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](http://www.cubrid.org/blog/dev-platform/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](http://www.cubrid.org/blog/dev-platform/how-to-tune-java-garbage-collection/), 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](http://www.cubrid.org/blog/dev-platform/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.

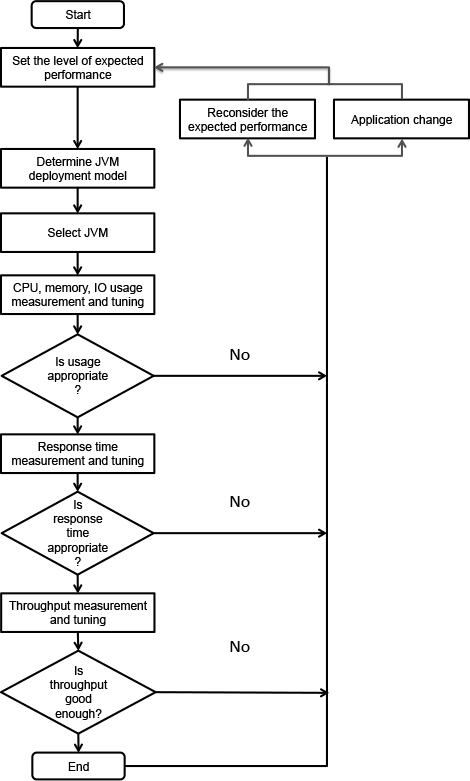
It is noted that from the perspective of an operating system JVM is also an application process. To make an environment in which a JVM can operate well, you should understand how an OS allocates resources to processes. This means, to tune the performance of Java applications, you should have an understanding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



**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

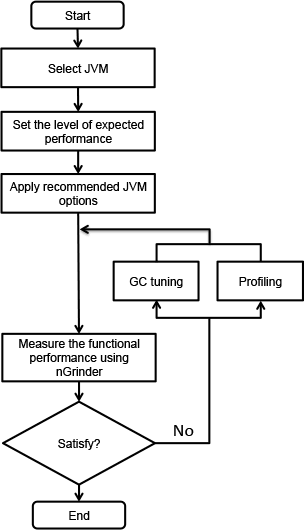
Selecting a JVM means whether to use a **32-bit JVM** or a **64-bit JVM**. Under the same conditions, you had better choose a 32-bit JVM. This is because a 32-bit JVM performs better than a 64-bit JVM. However, the maximum logical heap size of a 32-bit JVM is 4 GB. (However, actual allocatable size for 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](http://www.readwriteweb.com/hack/2011/06/cpp-go-java-scala-performance-benchmark.php)**).** | | |
| --- | --- | --- |
| 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](http://www.cubrid.org/blog/dev-platform/understanding-java-garbage-collection/) 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](http://www.cubrid.org/blog/tags/NHN/) does. The flow chart in **Figure 2** below is a simpler procedure designed based on **Figure 1** to be more suitable for NHN.



**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](http://www.cubrid.org/blog/dev-platform/understanding-java-garbage-collection/). 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 toHttpRequest 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 theNewRatio 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:

|  |  |
| --- | --- |
| 1 | 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](http://www.nhnopensource.org/ngrinder/" \t "_self)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](http://www.cubrid.org/blog/dev-platform/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](http://www.cubrid.org/blog/dev-platform/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](http://en.wikipedia.org/wiki/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](http://en.wikipedia.org/wiki/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.

# [How to Monitor Java Garbage Collection](http://www.cubrid.org/blog/dev-platform/how-to-monitor-java-garbage-collection/)

This is the second article in the series of "[*Become a Java GC Expert*](http://www.cubrid.org/blog/tags/Garbage%20Collection/)". In the first issue [Understanding Java Garbage Collection](http://www.cubrid.org/blog/dev-platform/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**](http://www.cubrid.org/blog/dev-platform/understanding-jvm-internals/)**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:

when an object in young has moved to old and by how much,

or when [stop-the-world](http://www.cubrid.org/blog/dev-platform/understanding-java-garbage-collection/#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](http://www.cubrid.org/blog/tags/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.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | $> jstat –gc  $<vmid$> 1000    S0C       S1C       S0U    S1U      EC         EU          OC         OU         PC         PU         YGC     YGCT    FGC      FGCT     GCT  3008.0   3072.0    0.0     1511.1   343360.0   46383.0     699072.0   283690.2   75392.0    41064.3    2540    18.454    4      1.133    19.588  3008.0   3072.0    0.0     1511.1   343360.0   47530.9     699072.0   283690.2   75392.0    41064.3    2540    18.454    4      1.133    19.588  3008.0   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:

|  |  |
| --- | --- |
| 1  2  3  4 | 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 |
| --- | --- | --- |
| S0C | Displays the current size of Survivor0 area in KB | -gc -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**.

|  |  |
| --- | --- |
| 1  2  3  4 | 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**:

|  |  |
| --- | --- |
| 1  2  3  4 | 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 |

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

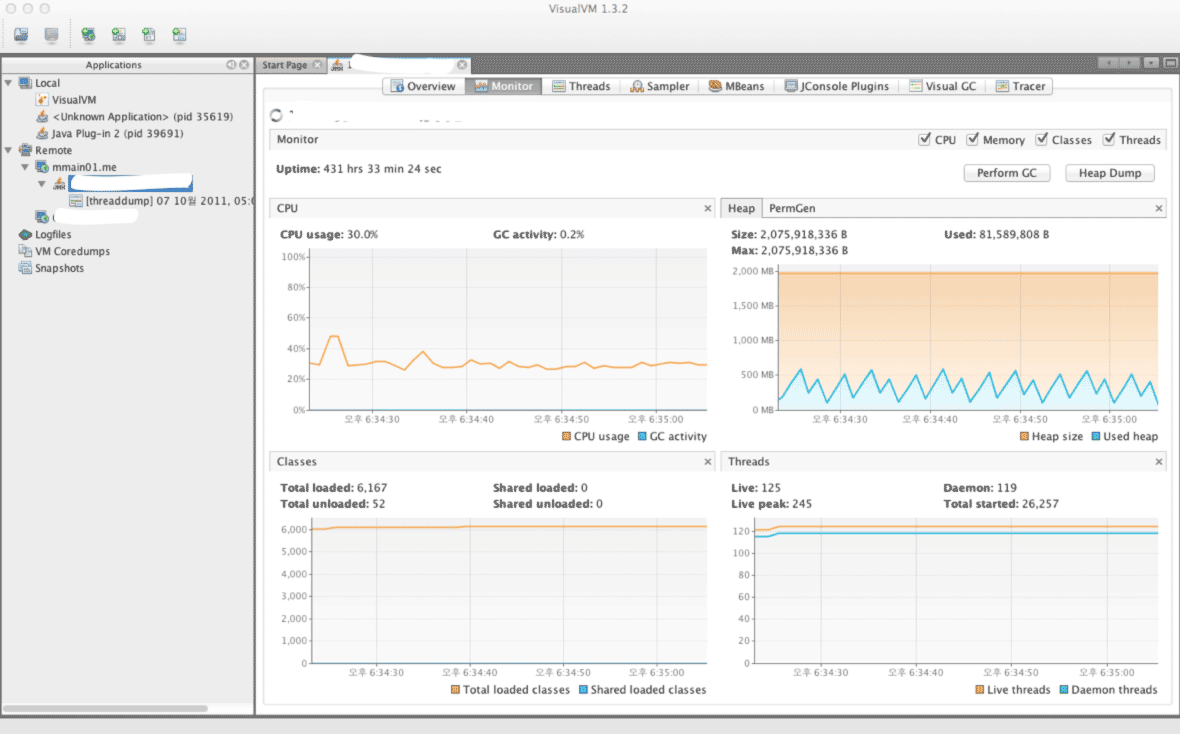
|  |  |
| --- | --- |
| 1 | [Full GC [Tenured: 3485K->4095K(4096K), 0.1745373 secs] 61244K->7418K(63104K), [Perm : 10756K->10756K(12288K)], 0.1762129 secs] [Times: user=0.19 sys=0.00, real=0.19 secs] |

If a [CMS collector](http://www.cubrid.org/blog/dev-platform/understanding-java-garbage-collection/#cms-gc) 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.

[](http://www.cubrid.org/files/attach/images/220547/126/316/visual-vm.png)

**Figure 1: VisualVM Screenshot.**

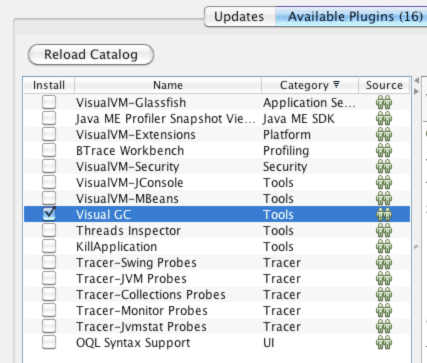
Instead of the version that is included with JDK, you can download Visual VM directly from its website. For the sake of convenience, the version included with JDK will be referred to as Java VisualVM (jvisualvm), and the version available from the website will be referred to as Visual VM (visualvm). 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.

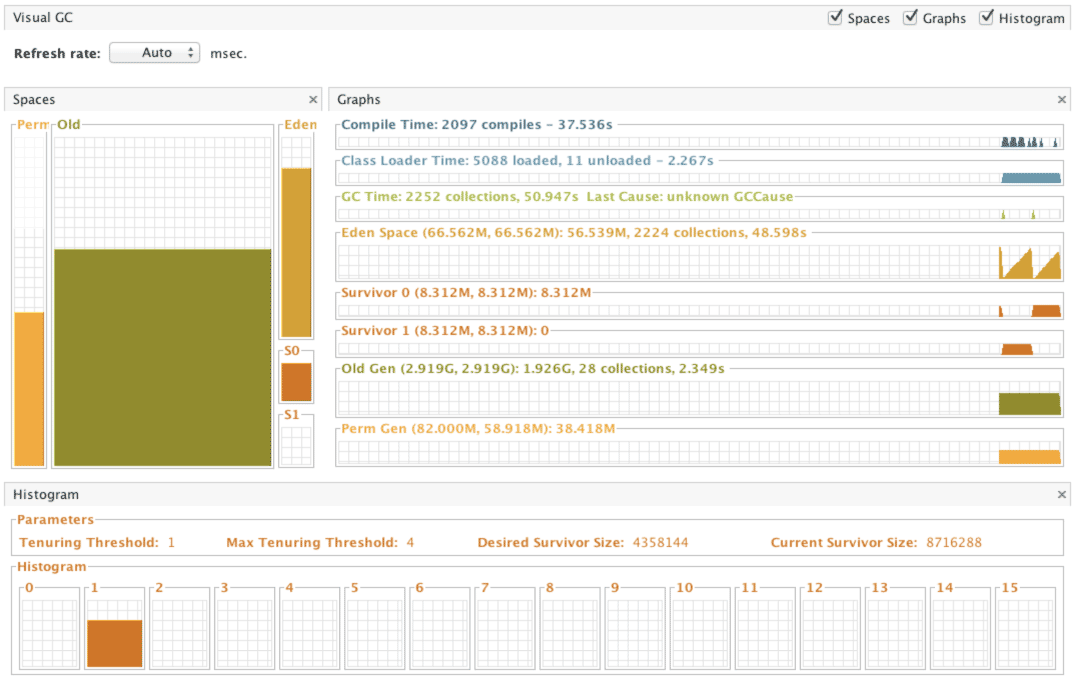
If you want the detailed information provided by jstat, then it is recommended to install the Visual GC plug-in.

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



**Figure 2: Viusal GC Installation Screenshot.**

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

[](http://www.cubrid.org/files/attach/images/220547/126/316/visual-gc-execution.png)

**Figure 3: Visual GC execution screenshot.**

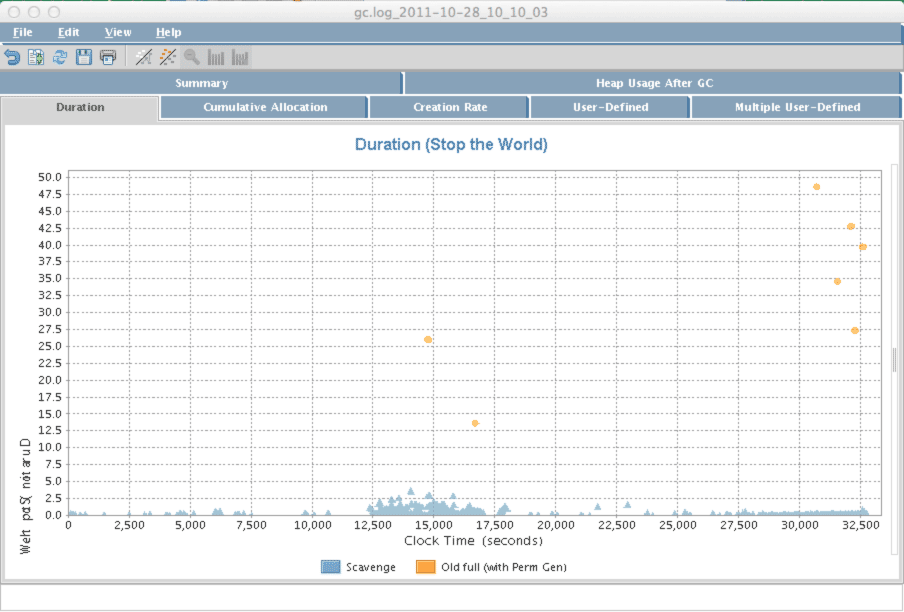
**HPJMeter**

[HPJMeter](https://h20392.www2.hp.com/portal/swdepot/displayProductInfo.do?productNumber=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.

[](http://www.cubrid.org/files/attach/images/220547/126/316/hpjmeter.png)

**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.