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

posted 2 years ago in **Dev Platform** category by [[[Level:0]](http://www.cubrid.org/blog/dev-platform/understanding-java-garbage-collection/)Sangmin Lee](http://www.cubrid.org/blog/dev-platform/understanding-java-garbage-collection/)

[**267**](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.)

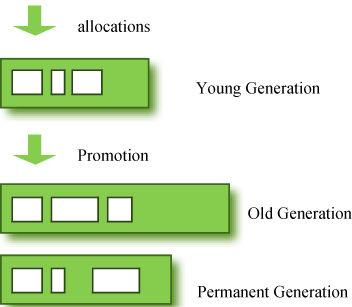
* Most objects soon become unreachable.
* References from old objects to young objects only exist in small numbers.

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.



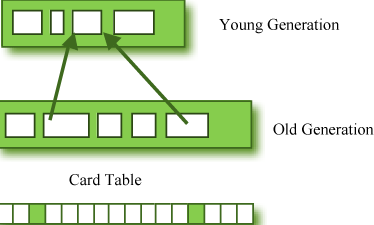
**Figure 1: GC Area & Data Flow.**

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.

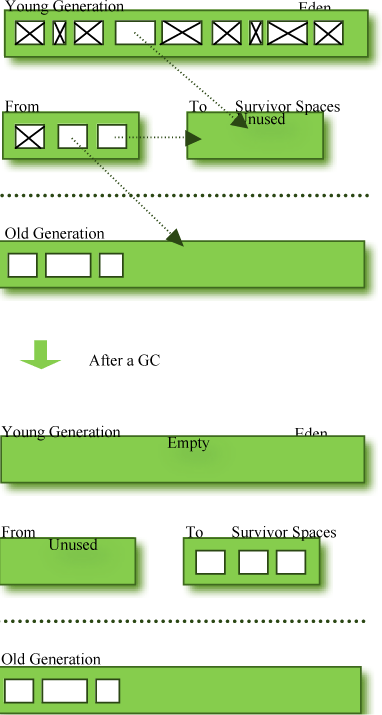
* One **Eden** space
* Two **Survivor** spaces

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.

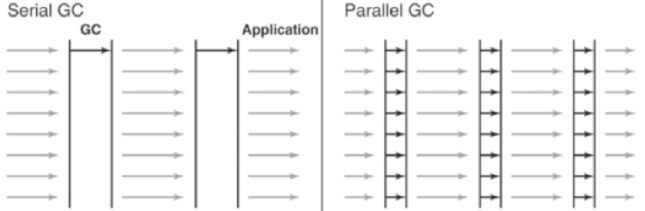
**Serial GC (-XX:+UseSerialGC)**

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)**



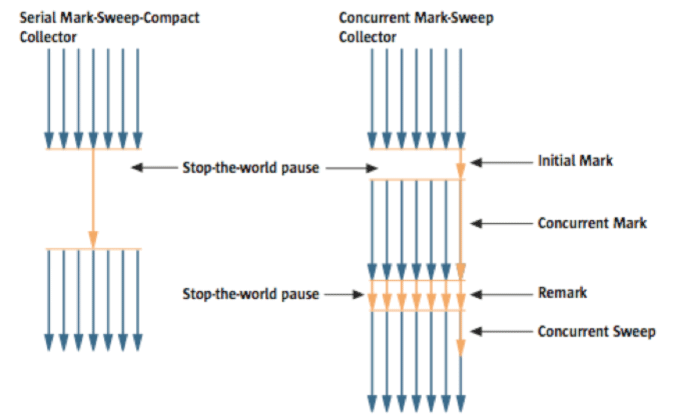
**Figure 4: Difference between the Serial GC and Parallel GC.**

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**.

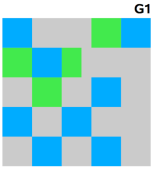
While this GC type has the advantage of short stop-the-world time, it also has the following disadvantages.

* 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**.

I would like to note that I referred to a new book released in December 2011 called "*Java Performance*" ([Amazon](http://amzn.com/0137142528), 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**](http://www.cubrid.org/blog/dev-platform/how-to-monitor-java-garbage-collection/)

posted 2 years ago in **Dev Platform** category by [[[Level:0]](http://www.cubrid.org/blog/dev-platform/how-to-monitor-java-garbage-collection/)Sangmin Lee](http://www.cubrid.org/blog/dev-platform/how-to-monitor-java-garbage-collection/)

[**130**](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:

1. when an object in young has moved to old and by how much,
2. 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.

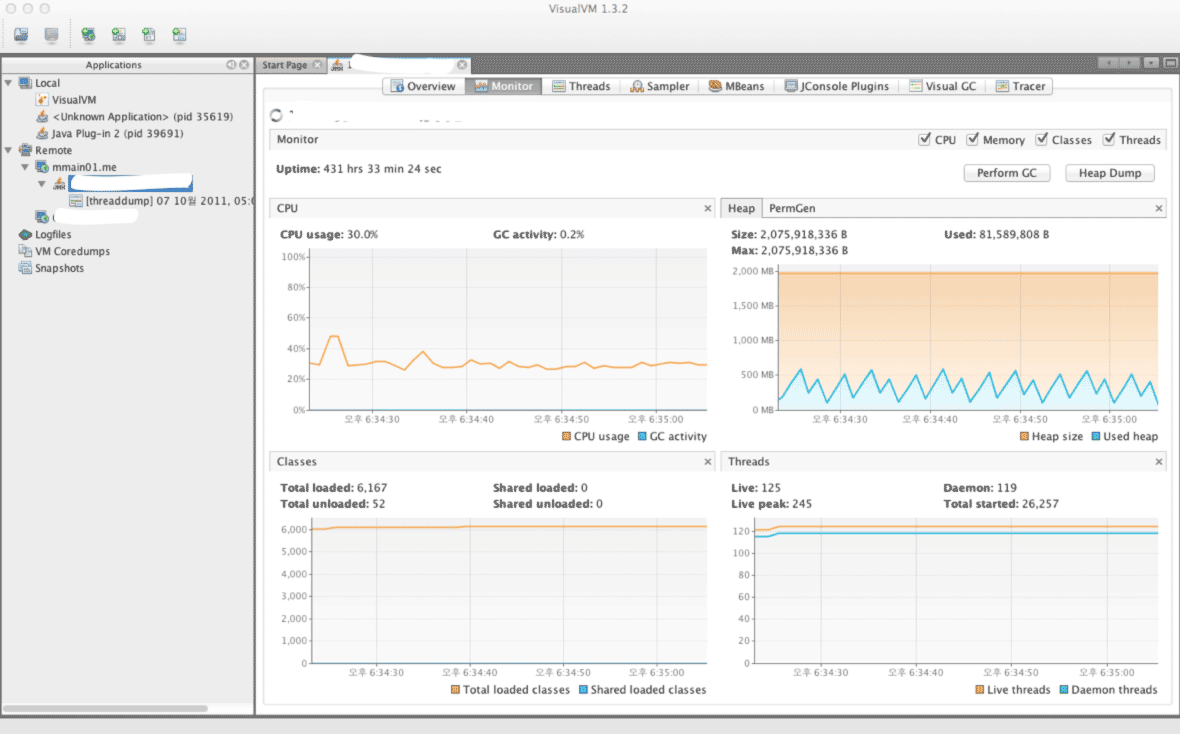
|  |  |
| --- | --- |
| 1 | [Full GC [Tenured: 3485K->4095K(4096K), 0.1745373 secs] 61244K->7418K(63104K), [Perm : 10756K->10756K(12288K)], 0.1762129 secs] [Times: user=0.19 sys=0.00, real=0.19 secs] |

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.**

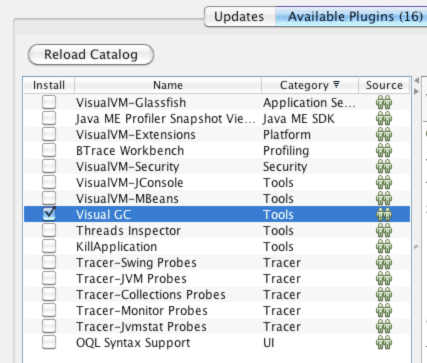
Instead of the version that is included with JDK, you can download Visual VM directly from its website. For the sake of convenience, the version included with JDK will be referred to as Java VisualVM (jvisualvm), and the version available from the website will be referred to as Visual VM (visualvm).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

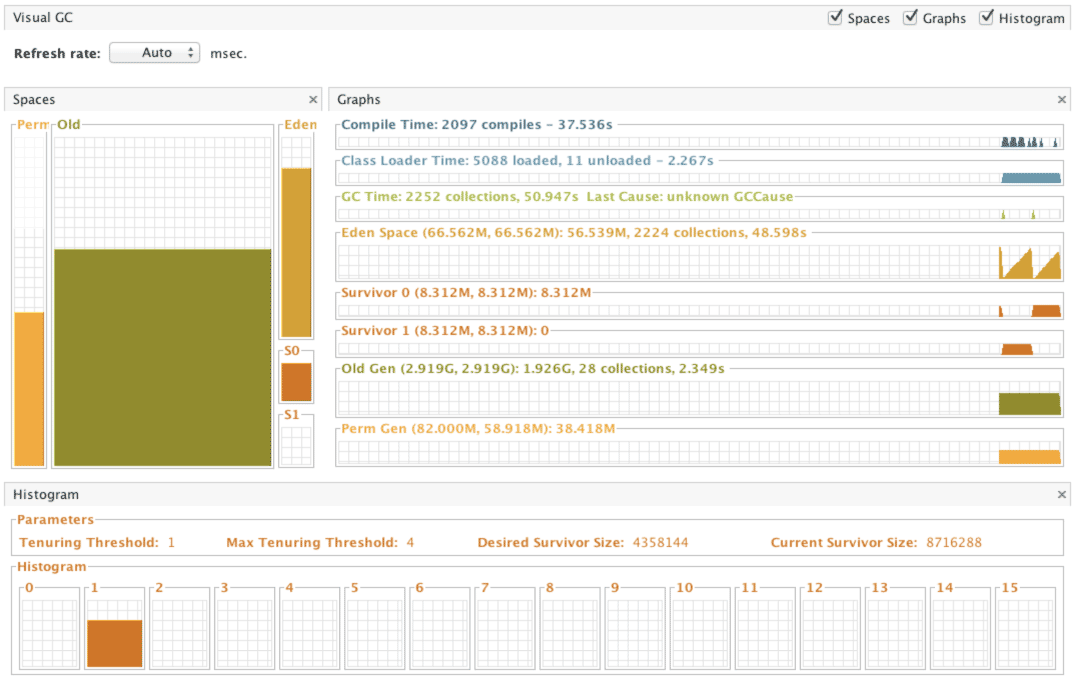
If you want the detailed information provided by jstat, then it is recommended to install the Visual GC plug-in.

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.**

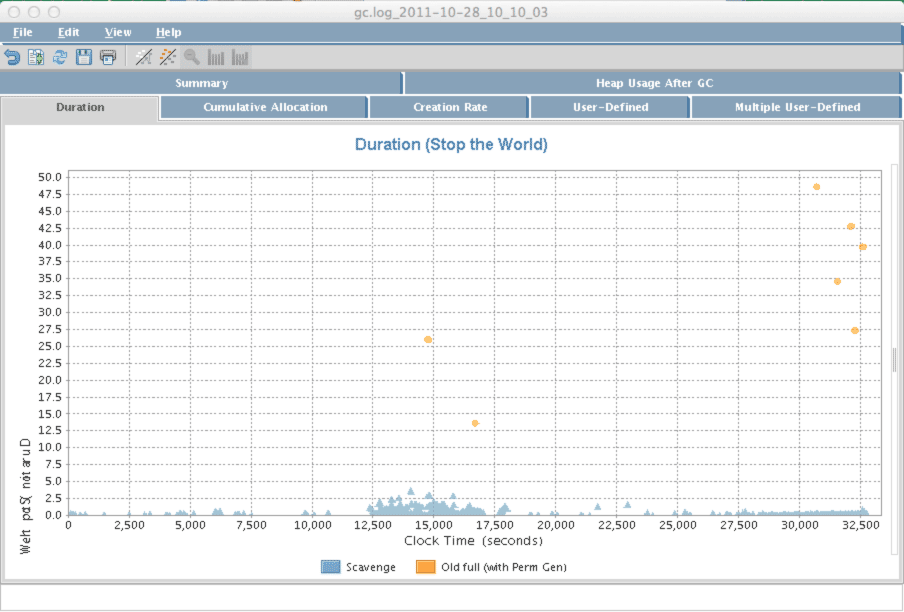
**HPJMeter**

[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.

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

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

posted 2 years ago in **Dev Platform** category by [[[Level:0]](http://www.cubrid.org/blog/dev-platform/how-to-tune-java-garbage-collection/)Sangmin Lee](http://www.cubrid.org/blog/dev-platform/how-to-tune-java-garbage-collection/)

[**80**](http://www.cubrid.org/blog/dev-platform/how-to-tune-java-garbage-collection/)

This is the *third* 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/) I 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 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](http://www.cubrid.org/blog/dev-platform/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](http://www.cubrid.org/blog/dev-platform/understanding-java-garbage-collection/). 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](http://www.cubrid.org/blog/dev-platform/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](http://book.naver.com/bookdb/book_detail.nhn?bid=6654751) (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](http://www.cubrid.org/blog/dev-platform/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).

|  |  |
| --- | --- |
| 1  2  3  4 | $ 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](https://blogs.oracle.com/poonam/entry/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 -verbosegcoption, 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.

|  |  |
| --- | --- |
| 1  2 | 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.

|  |  |
| --- | --- |
| 1  2 | 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.

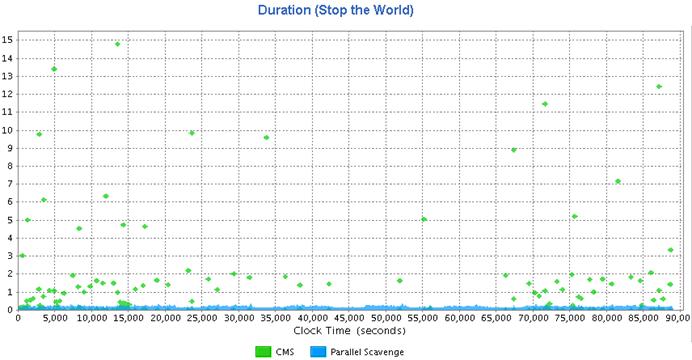
|  |  |
| --- | --- |
| 1  2 | S0 S1 E O P YGC YGCT FGC FGCT GCT  8.61 0.00 30.67 24.62 22.38 2424 30.219 0 0.000 30.219 |

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.

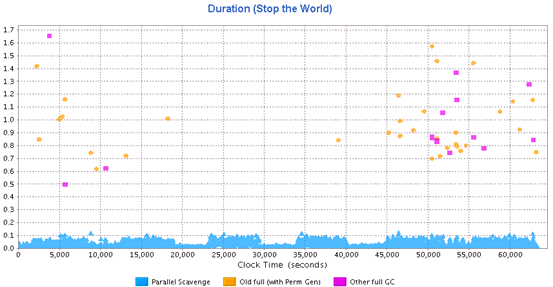
So I changed the GC type from CMS GC to Parallel GC. I changed the memory size to 2 GB and then set theNewRatio to 3. The result of jstat –gcutil after a few hours is as follows.

|  |  |
| --- | --- |
| 1  2 | 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 causesOutOfMemoryError 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](http://www.slideshare.net/) for related materials. The most recommendable material is [Everything I Ever Learned About JVM Performance Tuning @Twitter](http://www.slideshare.net/aszegedi/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".

[**How Statement Pooling in JDBC affects the Garbage Collection**](http://www.cubrid.org/blog/dev-platform/how-statement-pooling-in-jdbc-affects-garbage-collection/)

posted 2 years ago in **Dev Platform** category by [[[Level:0]](http://www.cubrid.org/blog/dev-platform/how-statement-pooling-in-jdbc-affects-garbage-collection/)Dongsun Choi](http://www.cubrid.org/blog/dev-platform/how-statement-pooling-in-jdbc-affects-garbage-collection/)

[**29**](http://www.cubrid.org/blog/dev-platform/how-statement-pooling-in-jdbc-affects-garbage-collection/)

There are various techniques to improve the performance of your Java application. In this article I will talk about**Statement Pooling Configuration** and **its effect on Garbage Collection process**.

Statement Pooling allows to improve the performance of an application by caching SQL statements that are used repeatedly. Such caching mechanism allows to prepare frequently used statements only once and reuse them multiple times, thus reducing the overall number of times the database server has to parse, plan, and optimize these queries. A well-configured number of statements (maxStatements) to be cached can be as good as tuning the [Garbage Collection](http://www.cubrid.org/blog/tags/Garbage%20Collection/). Now let's see how Statement Pooling can affect the Garbage Collection.

**Why Check the Number of Statement in the Pool?**

Often the size of the JDBC statement pool is set to the default value. Using the default value, of course does not usually lead to any special issue. But a well-configured maxStatements value can be as effective as GC tuning. If you are using the default maxStatements value and would like to optimize the use of memory, let's think about the correct statement pool value before attempting GC tuning.

As was discussed in [Understanding Java Garbage Collection](http://www.cubrid.org/blog/dev-platform/understanding-java-garbage-collection/), a **weak generational hypothesis** (*most objects quickly become unreachable and a reference from an old object to a new object is rare*) was used as the precondition when creating garbage collector in Java. For the majority of [NHN](http://www.cubrid.org/blog/tags/NHN/) web services there should be a response within 300ms at the latest, unless it is a special case. Therefore, NHN web services are more applicable to the above situations than the general stand-alone type applications.

**The GC Process between HTTP Request and Response**

When developing a web service using web containers like Tomcat and other frameworks, the lifespan of objects created by a developer tend to be either very short or very long. Web developers usually write codes like Interceptor, Action, BO, or DAO (*BO and DAO are generated and used as singletons from applicationContex in Spring, and are not the target of GC*). The objects generated from these codes stay alive for a very brief time that exists **between the time** HTTP is **requested** and the time it has **responded**. For this reason, such objects are usually collected during [Young GC](http://www.cubrid.org/blog/dev-platform/understanding-java-garbage-collection/).

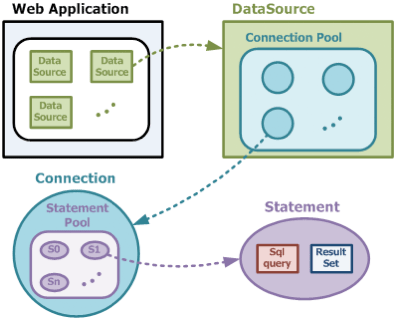
There are also objects, such as ***singleton objects***, that stay alive long enough to exist for the lifecycle of Tomcat. Such objects will be promoted to the ***old*** area soon after Tomcat starts running. Yet, when continuously[monitoring web applications](http://www.cubrid.org/blog/dev-platform/how-to-monitor-java-garbage-collection/) through **jstat** and the like, there are always some objects promoted to the *old* area during Young GC. These objects are usually used *after being stored in the cache* used for improving the performance of frameworks in most of the containers and projects. Whether the cached objects become the target of GC or not is determined by their ***cache hit ratio***, not their age, so unless the hit ratio is 100%, they cannot avoid being promoted to the old area, even when the Young GC cycle is set to be long.

Among these caches, **statement pooling affects the memory usage the most**. If you are using **iBatis**, as iBatis processes all SQLs as preparedStatment, you will be using statement pooling. *If the size of statement pooling is smaller than the number of SQLs being used*, the cache hit ratio will decrease and result in ***cache maintenance cost***. Objects that are reachable in the old area become the target of GC and will be retrieved, then will be regenerated during the HTTP request process, only to be cached and promoted to the old area. The full GC cycles are affected by this process.

**Size of the Statement Objects**

It would be safe to say that the size of a single statement object is proportional to the length of the SQL code processed by the same statement. Even for a long and complex SQL, the size of the object should be around 500 bytes. The object's small size would seem to have little effect on the full GC cycles, but **such an assumption would be incorrect**.

When you look at the JDBC specifications, each connection has its own statement pool (maxStatementsPerConnection), as described in **Figure 1** below. So, although a statement object is as small as 500 bytes, if there are many connections, the statements cache may occupy the proportional amount of the heap.



**Figure 1: Relationship between the Connection and the Statement.**  
*(Though the statement has the ResultSet, it should be clarified that ResultSet is not an object for caching. ResultSet is allocated as null when rs.close() is called by iBatis, then retrieved in the young area during young GC.)*

**The Effect of Statement Pool's Cache Hit Ratio on the Full GC**

A simple test program was created to **assess the effect of cache hit ratio on the full GC**. One cache hit ratio was set to 100% while the other was set to 50%. When the same amount of load was applied, the results presented in **Table 1** and **2** were obtained.

In both cases, the occurrences of young GC were very similar but the results for the full GC was different. When the cache hit ratio was 100%, full GC occurred only once, because the number of objects promoted to the old area during young GC was small. When the ratio was 50%, full GC occurred 4 times because the number of statement objects promoted to the old area during young GC was high, as the objects were cached in the statement pool, then removed from the pool in LRU way, then cached again at the next request.

**Table 1. Cache hit ratio = 100%.**

| ... | OC | OU | YGC | FGC | FGCT | GCT |
| --- | --- | --- | --- | --- | --- | --- |
| ... | 10688.0 | 6940.9 | 532 | 1 | 0.190 | 1.274 |
| ... | 10688.0 | 6940.9 | 532 | 1 | 0.190 | 1.274 |

**Table 2. Cache hit ratio = 50%.**

| ... | OC | OU | YGC | FGC | FGCT | GCT |
| --- | --- | --- | --- | --- | --- | --- |
| ... | 10240.0 | 7092.7 | 554 | 4 | 0.862 | 2.253 |
| ... | 10240.0 | 7412.0 | 555 | 4 | 0.862 | 2.255 |

I would like to add one more thing. When the cache hit ratio is 50%, it violates the 2nd category of *weak generational hypothesis* I introduced previously. When low *cache hit ratio* causes frequent pool registration and subsequent removal, it means the statement object generated in the young area is being referenced in the pool from the old area, which leads to additional strain during GC because the [*card marking technique*](http://www.cubrid.org/blog/dev-platform/understanding-java-garbage-collection/) is used to manage the references separately.

**In Conclusion**

In Lucy (NHN's internal Java Framework), the maxStatements value for statement pooling in Oracle and MySQL is **500**. In most cases, 500 should be enough. However, when more SQL is being used, increasing the default value to meet such demand would be a way to improve the system efficiency (w*hen using $(String replacement) for query on iBatis for the reason of table partitioning and the like, the number of queries must be multiplied by the number of partitioned tables*).

However, when the default value is higher than necessary, this leads to a different problem. A higher value means more memory usage and higher likelihood of an Out Of Memory (OOME) occurrence.

In a situation where the number of SQLs are 10,000 and the number of connections are 50, then the total size of statement objects is about 250 MB. (500 byte \* 50 \* 10,000 = 250 MB). It should be easy to determine the likelihood of OOME occurrence by checking the Xmx configuration for the service in use.

What strategy do you follow to determine the correct number of statements to be pooled? Share your experience in the comments below.

[**Understanding JDBC Internals & Timeout Configuration**](http://www.cubrid.org/blog/dev-platform/understanding-jdbc-internals-and-timeout-configuration/)

posted 2 years ago in **Dev Platform** category by [[[Level:0]](http://www.cubrid.org/blog/dev-platform/understanding-jdbc-internals-and-timeout-configuration/)Woon Duk Kang](http://www.cubrid.org/blog/dev-platform/understanding-jdbc-internals-and-timeout-configuration/)

[**68**](http://www.cubrid.org/blog/dev-platform/understanding-jdbc-internals-and-timeout-configuration/)

An application with a proper JDBC timeout can cut down the failure time. In this article we would like to talk about different kinds of timeout values and recommended timeout application methods when you import values from DBMS.

**Web Application Server became unresponsive after a DDos attack one day**

*(This is a close reconstitution of an actual event.)*

The entire service did not work normally after a DDos attack. The network was disconnected because L4 was not working, which caused WAS to be inoperable as well. Shortly afterwards, the security team blocked all DDos attacks, and restored the network back to normal. Yet, WAS was still not working.

Through the [ThreadDump](http://www.cubrid.org/blog/dev-platform/how-to-analyze-java-thread-dumps/) of WAS, the service team was able to confirm that WAS had stopped during API call from JDBC. After 20 minutes, WAS was still in WAITING status and the service was still not working. About 30 minutes had passed when an exception suddenly occurred, and the service was restored.

**Why was WAS in WAITING status for 30 minutes when QueryTimeout value was set to 3 seconds, and why did WAS start working again after 30 minutes?**

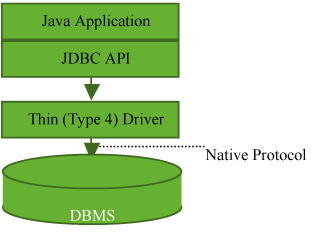
You can find the answer if you understand how the JDBC Timeout works.

**Why Do We Need to Know about the JDBC Driver?**

When there is a performance issue or an error, WAS and DBMS are the two important tiers we pay attention to. In [NHN](http://www.cubrid.org/blog/tags/NHN/), WAS and DBMS are generally handled by different departments, so each department tries to figure out this situation by focusing on their own area of expertise. When this happens, you get a blind spot between WAS and DBMS, that does not receive much attention. For Java applications, the blind spot would be between DBCP and JDBC. In this article we will focus on JDBC.

**What is a JDBC Driver?**

JDBC is a standard API that you use to access the DBMS in Java applications. There are [4 types of JDBC drivers](http://en.wikipedia.org/wiki/JDBC_driver)(Wikipedia) defined by Sun. NHN mainly uses the *type 4*. JDBC type 4 driver is written entirely in Java (pure Java) and communicates with a DBMS using sockets in Java applications.

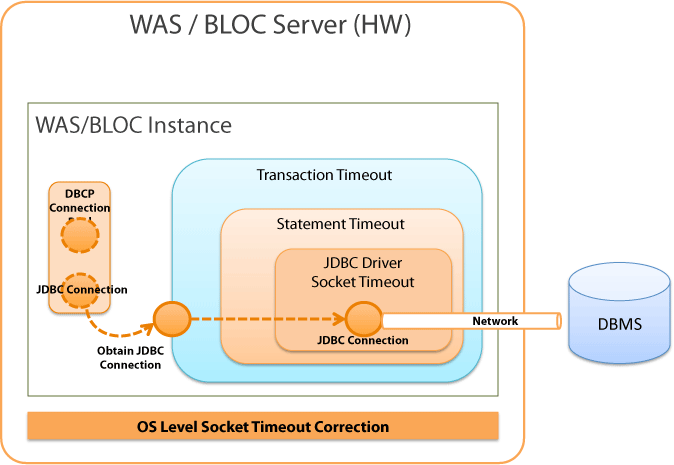


**Figure 1: JDBC Type 4.**

Type 4 drivers process byte stream via sockets, and have the same basic operations as a network library like*HttpClient*. This uses up a lot of CPU resources and loses response timeout, while sharing the same error points with other network libraries. If you have used HttpClient before, then you must have encountered errors from not setting the timeout value. Type 4 driver may have the same error (a hang occurs) if the socket timeout value is not set properly.

Let's learn about how to configure the socket timeout value for JDBC driver, and what needs to be considered.

**Timeout Class at WAS - DBMS Communication**



**Figure 2: Timeout Class.**

Figure 2 above shows a simplified version of the timeout class when WAS and DBMS are communicating.

The higher level timeout is dependent on the lower level timeout. The higher level timeout will operate normally only if the lower level timeout operates normally as well. If the JDBC driver socket timeout does not work properly, then higher level timeouts such as statement timeout and transaction timeout will not work properly either.

We have received a lot of comments that said:

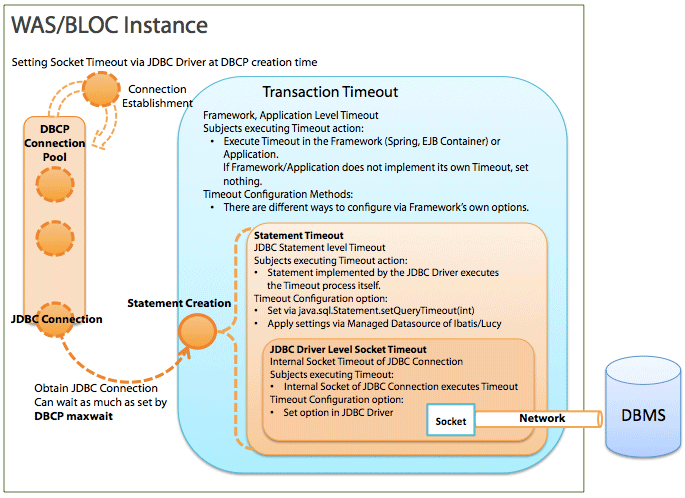
Even after the statement timeout was configured, the application still did not recover from the error because the statement timeout did not work at the time of network failure.

**The statement timeout does not handle the timeouts at the time of network failure.** Statement timeout does only one thing: *restricts the operation time of 1 statement*. Handling timeout to prevent network failure must be done by JDBC Driver.

The JDBC driver's socket timeout is affected by the OS's socket timeout configuration. This would explain why JDBC connection hang recovers 30 minutes after the network connection failure, even when the JDBC driver's socket timeout is not configured.

DBCP Connection Pool is located on the left side of Figure 2. You can see that the timeout classes and DBCP are separated. DBCP is in charge of creating and managing connections, and is not involved in processing timeouts. When a connection is created within DBCP or a validation query is sent to check the validity of the connection, the socket timeout does affect these processes but does not affect the application directly.

However, when getConnection() is called to DBCP from the application logic, then you can specify the timeout until the application acquires the connection. However, this has nothing to do with the JDBC's connect timeout.



**Figure 3: Timeout for Each Levels.**

**What is Transaction Timeout?**

**Transaction timeout** is a timeout valid in *frameworks* (Spring, EJB container) or at the *application level*.

Transaction timeout can be an unfamiliar concept. Simply put, transaction timeout is "Statement Timeout \* N (number of statements being processed) + @ (garbage collection, etc.)." Transaction timeout is used to limit the total statement processing time to the maximum amount allowed.

For example, if it takes 0.1 second to process 1 statement, processing a few statements would not be a problem, but processing 100,000 statements would take 10,000 seconds (approx. 7 hours). Statement timeout can be used here.

EJB CMT (Container Managed Transaction) would be a typical example of actual implementations. EJB CMT varies in its implementation methods and operating process depending on developers. NHN does not use EJB Container, so transaction timeout of Spring Framework would be the most common example. In Spring, you may use XML as shown below or use @Transactional from Java source codes, for configuration.

|  |  |
| --- | --- |
| 1  2  3 | <tx:attributes>          <tx:method name="…" timeout="3"/>  </tx:attributes> |

Statement timeout provided by Spring is very simple. It records the starting time and the elapsed time for each transaction, and checks the elapsed time when an event occurs. If the timeout is abnormal, it generates an exception.

In Spring, the connection is stored in, and used from ThreadLocal. This is called **Transaction Synchronization**. When a connection is saved in ThreadLocal, the starting time and the timeout time of the transaction is also recorded. When a statement is being created by using the proxy connection, the elapsed time is checked to generate an exception.

The EJB CMT implementation is done in a similar way. The structure itself is very simple. If the transaction timeout is very important but the container or the framework you are using does not provide this feature, you could implement it yourself without major problems. There is no standard API for transaction timeout.

Lucy 1.5 and 1.6 Framework does not have a transaction timeout feature, but you can get the same result by using Transaction Manager from Spring.

If the processing time of the statement (5 or less) is 200 ms and the processing time of other business logics or framework operation is 100 ms, the transaction timeout time should be set to 1,100 ms ((200 \* 5) + 100) or more.

**What is Statement Timeout?**

It is a *limitation on how long a statement should run*. It sets the timeout value for the statement, which is a JDBC API. The JDBC driver processes the statement timeout based on this value. Statement timeout is configured via java.sql.Statement.setQueryTimeout(int timeout), which is a JDBC API. In recent developing environments, the developers rarely configure the statement timeout value directly through Java source codes, but often configure it by using the framework.

To use **iBatis** as an example, the default value can be configured by using @defaultStatementTimeoutvalue in *sqlMapConfig/settings* of *sql-map-config.xml*. By using @timeout value, you can configure statement, select, insert and update syntax of *sql-map.xml* separately.

When MangedDatasource of Lucy 1.5 and 1.6 is used, the queryTimeout option can be used to get a statement of which timeout is configured at the datasource level.

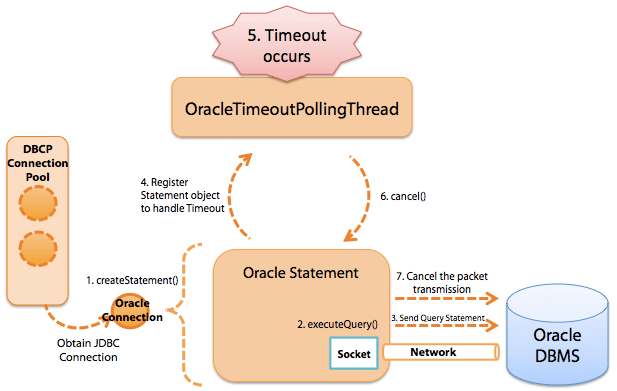
The statement timeout time is configured based on the features of each application, so there is no recommended configuration value.

**Statement Timeout Execution Process for JDBC Driver**

Statement timeout works differently per DBMS and driver. The way it works is similar between Oracle and MS SQLServer. It is also similar between MySQL and CUBRID.

**QueryTimeout for Oracle JDBC Statement**

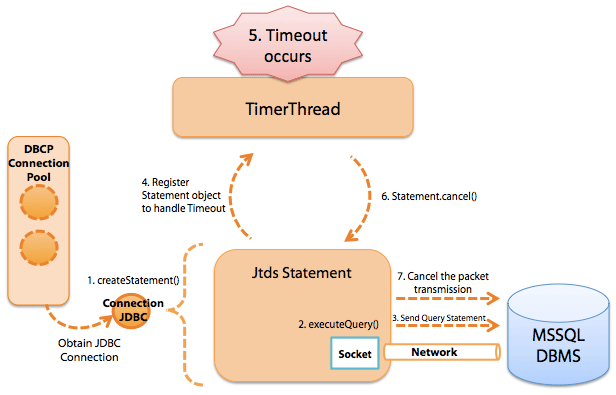
1. Creates a statement by calling Connection.createStatement().
2. Calls Statement.executeQuery().
3. The statement transmits the Query to Oracle DBMS by using its own connection.
4. The statement registers a statement to OracleTimeoutPollingThread (1 for each classloader) for timeout process.
5. Timeout occurs.
6. OracleTimeoutPollingThread calls OracleStatement.cancel().
7. Sends a cancel message through the connection and cancels the query being executed.



**Figure 4: Query Timeout Execution Process for Oracle JDBC Statement.**

**QueryTimeout for JTDS (MS SQLServer) Statement**

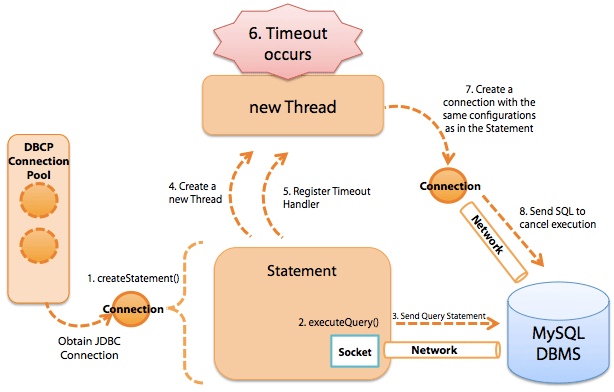
1. Creates a statement by calling Connection.createStatement().
2. Calls Statement.executeQuery().
3. The statement transmits the Query to MS SqlServer by using the internal connection.
4. The statement registers a statement in TimerThread for timeout process.
5. Timeout occurs.
6. TimerThread calls up TsdCore.cancel() inside the JtdsStatement object.
7. Sends a cancel message through the ConnectionJDBC and cancels the query being executed.



**Figure 5: QueryTimeout Execution Process for JTDS (MS SQLServer) Statement.**

**QueryTimeout for MySQL JDBC Statement (5.0.8)**

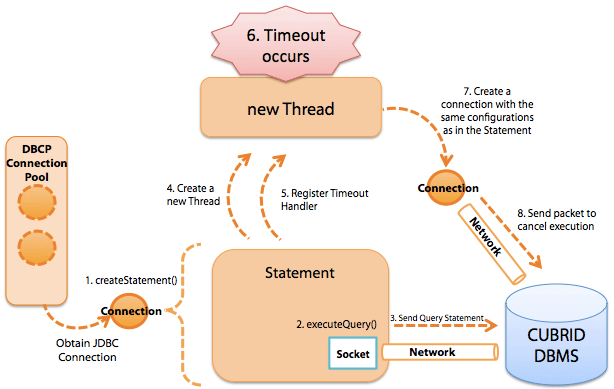
1. Creates a statement by calling Connection.createStatement().
2. Calls Statement.executeQuery().
3. The statement transmits the Query to MySqlServer by using the internal connection.
4. The statement creates a new timeout-execution thread for timeout process.
5. For version 5.1.x, it changes to assign 1 thread for each connection.
6. Registers the timeout execution to the thread.
7. Timeout occurs.
8. The timeout-execution thread creates a connection that has the same configurations as the statement.
9. Transmits the cancel Query (KILL QUERY "connectionId“) by using the connection.



**Figure 6: QueryTimeout Execution Process for MySQL JDBC Statement (5.0.8).**

**QueryTimeout for CUBRID JDBC Statement**

1. Creates a statement by calling Connection.createStatement().
2. Calls Statement.executeQuery().
3. The statement transmits the Query to CUBRID DBMS by using the internal connection.
4. The statement creates a new timeout-execution thread for timeout process.
5. Registers the timeout execution to the thread.
6. Timeout occurs.
7. The timeout-execution thread creates a connection that has the same configurations as the statement.
8. Transmits the cancel message using the connection.

****

**Figure 7: QueryTimeout Execution Process for CUBRID JDBC Statement.**

**What is Socket Timeout for JDBC Driver?**

JDBC driver type 4 uses the socket to connect to the DBMS, and the connection timeout process between the application and the DBMS is not carried out by the DBMS.

Socket timeout value for JDBC driver is necessary when the DBMS is terminated abruptly or an network error has occured (equipment malfunction, etc.). Because of the structure of TCP/IP, there are no means for the socket to detect network errors. Therefore, the application cannot detect any disconnection with the DBMS. If the socket timeout is not configured, then the application may wait for the results from the DBMS indefinitely. (This connection is also called a "**dead connection**.") To prevent dead connections, a timeout must be configured for the socket. Socket timeout can be configured via JDBC driver. By setting up the socket timeout, you can prevent the infinite waiting situation when there is a network error and shorten the failure time.

It is not recommended to use the socket timeout value to limit the statement execution time. So **the socket timeout value must be higher than the statement timeout value**. If the socket timeout value is smaller than the statement timeout value, as the socket timeout will be executed first, and the statement timeout value becomes meaningless and will not be executed.

Socket timeout has 2 options listed below, and their configurations vary by driver.

* Timeout at socket connection: Time limit for Socket.connect(SocketAddress endpoint, int timeout)
* Timeout at socket reading/writing: Time limit for Socket.setSoTimeout(int timeout)

By checking the source for CUBRID, MySQL, MS SQL Server (JTDS) and Oracle JDBC, we confirmed that all the drivers we checked use the 2 APIs above.

How to configure SocketTimeout is as explained below.

| JDBC Driver | connectTimeout | Default | Unit | Application Method |
| --- | --- | --- | --- | --- |
| socketTimeout | Default | Unit |
| MySQL Driver | connectTimeout | 0 | ms | Specify the option in the DriverURL. **Format:** jdbc:mysql://[host:port],[host:port].../[database] **[?propertyName1][=propertyValue1][&propertyName2][=propertyValue2]...**  **Example:**   |  | | --- | | jdbc:mysql://xxx.xx.xxx.xxx:3306/database?connectTimeout=60000&socketTimeout=60000 | |
| socketTimeout | 0 | ms |
| MS-SQL Driver jTDS Driver | loginTimeout | 0 | sec | Specify the option in the DriverURL. **Format:** jdbc:jtds:<server\_type>://<server>[:<port>][/<database>]**[;<property>=<value>[;...]]**  **Example:**   |  | | --- | | jdbc:jtds:sqlserver://server:port/database;loginTimeout=60;socketTimeout=60 | |
| socketTimeout | 0 | sec |
| Oracle Thin Driver | oracle.net.CONNECT\_TIMEOUT | 0 | ms | Not possible with the driverURL. Must be delivered to the properties object via OracleDatasource.setConnectionProperties() API. When DBCP is used, use the following APIs:   * BasicDatasource.setConnectionProperties() * BasicDatasource.addConnectionProperties() |
| oracle.jdbc.ReadTimeout | 0 | ms |
| CUBRID Thin Driver | No separate configuration | 5,000 | ms | Not possible with the driverURL. Timeout occurs in 5 seconds.   * Note 1: When timeout occurs with althost option specified in the URL, it can be connected to the designated host. * Note 2: C API can be used to state the **login\_time** option in ms in the URL. |
|  |  |  |  |

* Note 1: The default value for connectTimeout and socketTimeout is "0," which means that the timeout does not occur.
* Note 2: You can also configure through properties without directly using the separate API of DBCP.

When you configure properties, pass on the character string where the key value is “connectionProperties”, and the format value is “[propertyName=property;]\*”. The following example shows configuring properties through xml in iBatis.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <transactionManager type="JDBC">    <dataSource type="com.nhncorp.lucy.db.DbcpDSFactory">       ....       <property name="connectionProperties" value="oracle.net.CONNECT\_TIMEOUT=6000;oracle.jdbc.ReadTimeout=6000"/>    </dataSource>  </transactionManager> |

**OS Level SocketTimeout Configuration**

If the socket timeout or the connect timeout is not configured, most of the time, applications cannot detect network errors. So, until the applications are connected or are able to read data, they will wait indefinitely. However, if you look at the actual issues NHN services encountered, the problems were often resolved after the applications (WAS) tried to reconnect to the network 30 minutes after. This is because the OS can also configure socket timeout time. Linux servers used by NHN have set the socket timeout to 30 minutes. This checks the network connection at the OS level. Because the KeepAlive checking cycle for NHN's Linux servers is 30 minutes, even when socket timeout is set to 0, the DBMS network connection problems caused by network issues do not surpass 30 minutes.

Generally, the application hangs from network issues when the application is calling Socket.read(). However, depending on the network composition or the error type, it can rarely be in waiting status while running Socket.write(). When the application calls Socket.write(), the data is recorded to the OS kernel buffer and then the right to control is returned to the application immediately. Thus, as long as a valid value is recorded to the kernel buffer, Socket.write() is always successful. However, if the OS kernel buffer is full due to a special network error, even Socket.write() can be put into waiting status. In this case, the OS tries to resend the packet for a certain amount of time, and generates an error when it reaches the limit. In NHN's Linux server environment, the timeout for this situation is set to 15 minutes.

I have explained the internal operations of JDBC so far. I hope that this will help you with the correct timeout configuration and reducing errors.

If you have more questions or any good information related to JDBC, please leave your comments below.

Lastly, I have listed some of the frequently asked questions below.

**FAQ**

Q1. **I configured the query timeout by using Statement.setQueryTimeout(), but it does not work as expected when there is a network error.**

 ➔ Query Timeout only works when it is connected to the socket correctly. Therefore, it cannot be used to solve an exceptional situation with a network error. To be prepared for network errors, socket timeout in JDBC driver must be configured.

Q2. **How are transaction timeout, statement timeout and JDBC driver socket timeout related to the DBCP configuration values?**

➔ When the connection is acquired from DBCP to JDBC, nothing but waitTimeout is affected.

Q3. **If JDBC SocketTimeout is configured, wouldn't the connections that stayed in idle status for a long time in DBCP be closed?**

➔ No. The socket option is applied when the actual data is being written or read, so it does not affect the connections in idle status in DBCP. The socket option can have certain effect when new connections that lack in inside of DBCP are created, old idle connections are removed, or the validation is checked, but this does not cause any significant issues unless the network has an error.

Q4. **How long should SocketTimeout be set to?**

➔As I have mentioned in the main article above, it must be much bigger than the statement timeout, and there is no recommended value. Socket timeout value for the JDBC driver becomes effective after a network error occurs. A careful configuration for the value cannot prevent such the errors from happening, but sometimes shortens the time that the network is disabled (if the network is restored right away).

By Woon Duk Kang, Software Engineer at Web Platform Development Lab, NHN Corporation.