Tuning the JVM

Pre-requisites

Before conducting your tuning exercise:

1. Define performance targets.
2. Determine which version of the java JVM is available. More recent JVMs usually contain performance improvements of their own, especially in the area of garbage collection.
3. Selection of a 64-bit JVM supports much more available memory.
4. Select a garbage collector according to your needs and limitations. Bear in mind that garbage collectors have differing algorithms to suit particular goals and reduce the need to "stop the world". What is right for one implementation may not be right for another. You should test and compare with realistic scenarios and load in a pre-production environment.
5. In tuning, start with the defaults and monitor the execution using jconsole and GC logs. Examine memory consumption and GC collection time and frequenscy and tune accordingly.

Caveat: These are recommendations only and it is recommended that customers refer to Oracle and other tuning documentation, and use a performance testing optimisation strategy to optimally configure IG according to deployment. Only an incremental optimisation strategy will determine the optimal configuration for a given deployment and user load.

Summary

Before tuning the JVM, be sure to understand and define performance targets, and select an appropriate JVM and garbage collector (GC).

Memory

As a proxy, IG is generally a low memory consumer, proxying request and response data between the client and server. Other factors to consider though in determining memory sizes are:

- Caching (notification, session and request): size and lifetime
- Resource size: as larger resources

Heap Memory

- IG largely consumes YoungGen space so JVM memory settings should be focussed there.
- IG is a low user of OldGen memory, which remains largely static after startup, with a couple of exceptions to note:
  - if IG is proxying large resources then you may see more consumption here - as larger objects may be stored directly in OldGen memory.
  - Caching may increase the lifetime of objects, so obviating the the need for larger available OldGen space.

Metaspace Memory

- Metaspace is the location where he JVM stores application metadata (class meta-information, method and field tables, HIT optimisations, etc). It is unlimited by default.
- GC savings can be made by determining a Metaspace size that is appropriate for your applicaton, to reduce continued expansion. This is because the JVM always performs a full GC before it resizes Metaspace, which can be expensive.

GSA: I don't expect this kind of memory to need a lot of tuning, maybe just remove that section ?

WM: Ok, but if Metaspace is resized (too small) then we require a time-consuming full-GC. I think it's not that well-known so we should mention it. I'll reduce it, to reduce its importance but keep the section with that detail in it.

GSA: Metaspace is where JVM stores classes, it's not OldGen, so the sizing should be dictated by the size of the application (all the classes that need to be loaded in memory), or am I mistaken ?

WM: You're right, but may have missed my point. Metaspace isn't OldGen - and is where we store class info, method tables, JIT optimisations, etc. With PermGen it used to be limited but with Metaspace it's now unlimited. However, if the initial allocation isn't large enough then Metaspace needs to be increased. If it's increased then the JVM does a full GC first. So, the point is, you want it to be sized appropriately first to avoid the full GC. It should be easy enough (🙂) to size and is a cost saving.

WM: I've reworded it slightly to explain why setting this to a sensible value is a good thing.

(Will remove Tuesday 30/6/2020 if no further comment)

Memory Options

While configuring memory is a trial and error process, and best optimised incrementally, the following memory options are worth considering:

Option	Description
`-server`	Ensures the JVM uses server-optimised configuration, compilation and execution. This is the default with a 64-bit JVM.

`-XX:InitialHeapSize=<size>G`
 	
`-XX:MaximumHeapSize=<size>G`
	Configure the initial and maximum heap space:
	
	• Equivalent to `-Xms` and `-Xmx` flags respectively.
	• Oracle recommend that these figures are set the same to avoid expensive allocation operations. However, recent improvements in G1 with its adaptive optimization algorithm (see below) means that this may limit vertical scaling. Though it is still recommended to follow this practice, if using G1, consider testing individual initial and maximum values too.
	• As a start point, IG has been shown to operate well in performance testing with a heap size of 5Gb. That is, with this sized heap, even with more memory available, a memory increment was never done.

`-XX:NewSize=<size>G`
 	
`-XX:MaxNewSize=<size>G`
	Configure the initial and maximum YoungGen space:
	
	• This configuration is important to IG as a large consumer of Eden space.
	• Incrementally test to optimise, with anticipated concurrent load in a pre-production environment. The maximum should be within the overall allocated maximum heap-size (`-Xmx`), leaving space for non-heap memory.
	• If using G1, it is advised not to set these values and instead allow G1 to optimise based on use. As ever, this should be tested.

`-XX:+UseStringDeduplication`
 	
`-XX:UseStringDeduplicationAgeThreshold=<N>`
	Prevent String duplication and so conserve memory (java 8u20+), reducing GC needs. The threshold can be provided to specify the age after which a String becomes a candidate for deduplication.
	
	⚠ GSA: why not, but do we think that this can have a notable impact on our performances ?
	
	WM: It's documented everywhere as advisable. It's also advised in the DJ tuning guide for 7.0. It's basically the inlining of Strings, as used to be the case pre-java8.
	
	(Will remove Tuesday 30/6/2020 if no further comment)

`-XX:MaxTenuringThreshold=<N>`	Configure the number of transitions between survivor spaces before an object is moved to OldGen. Because IG largely consumes YoungGen space, we can determine that anything that eventually would live in OldGen space could be moved there early to free up Eden space and reduce objects transitioning between survivor spaces. We could therefore set `-MaxTenuringThreshold` to a low value, possibly as low as 1 to accomplish this. ⚠ GSA: Good, this is exactly the kind of description that is useful to our users: it explains why this is worth considering this option WM: Thanks *(Will remove Tuesday 30/6/2020 if no further comment)*

`-XX:MetaspaceSize=<size>G`
 	
`-XX:MaxMetaSpaceSize=<size>G`
	Configure initial and maximum Metaspace size (from java 8). GC savings can be made by determining a Metaspace size that is appropriate for your applicaton, to reduce continual dynamic expansion to claim more memory. This is because the JVM always performs a full GC before it resizes Metaspace, which can be expensive.
	
	⚠ GSA: Unsure if we should keep this one in the doc, for sure not in the first positions, it's relatively less important than young /old gen settings
	
	WM: Probably not vital but it is included in other guides and I think it's important to note we only need a smallish Metaspace. I've moved it to last position, as I want to describe it as II've provided a recommended default.
	
	GSA: Smallish until we use SAML 🙂
	
	WM: Lol, ok 🙂 Is there anything I need to document there then?
	
	WM: Added note why it's beneficial - to avoid full GC.
	
	(Will remove Tuesday 30/6/2020 if no further comment)

Summary

IG makes low memory demands and specifically is mostly a consumer of `YoungGen` memory. Use of caches or proxying of large resources will increase `OldGen` memory use.

The specific settings described in this section may be considered when optimising JVM memory. See Oracle documentation for more information.

Garbage collection (GC)

Selecting the right garbage collector and tuning collection can reduce expensive "stop the world" pauses in collection. As IG is largely a consumer of YoungGen memory, there is a lot of scope for tuning to avoid expensive major collections (OldGen space consumption).

As mentioned, it is generally advisable to select from available, stable garbage collectors for the given JVM version according to your performance targets. Each garbage collector has its own goals and algorithms supporting that goal. For example, The Parallel GC aims to increase throughput whereas GCG1 aims to reduce latency. It is therefore advisable to test and compare.

GSA: Do not talk about CMS at all: we need Java 11 anyway, so this is not an option

WM: Ok, removed. I had included it as a transition point for users moving from java 8, to give a clear picture that CMS is now deprecated.

(Will remove Tuesday 30/6/2020 if no further comment)

Parallel GC

- Default Hotspot collector in java 8 - in resource-rich environment (multiprocessor, memory availability).
- Multiple threads for managing heap-space speeds up collection process.
- Aims to maximise throughput.
- Requires more full-GCs ("stop-the-world"), freezing application threads while in progress.
- It is advisable to configure the `-XX:InitialHeapSize` and `-XX:MaxHeapSize` to the same value to prevent unnecessary memory increases.
- It is advisable to configure the `-XX:NewSize` and `-XX:MaxNewSize` to weigh the ratio in favour of YoungGen memory, which IG makes more use of.

The following Parallel GC options are worth considering:

GSA: If we don't provide added values, I would not bother adding these things in our doc, a pointer to the official doc page would be enough (and easier to maintain)

Added value like: why this is interesting for IG, or return of experience like we observed a noticeable impact when ...

WM: When you say added value though, the below is taken from looking into multiple GC tests and guides, and is intended only as a guide for IG to consolidate info (without having done exhaustive testing). I think it's therefore still very useful.

GSA, when I say added value, I mean we found a correlation that we can explain, otherwise it's not really more that just copy paste of the GC doc (hoping there is one somewhere)

WM: Ok, well, I can remove them but they are the key useful tuning flags.

(Will remove Tuesday 30/6/2020 if no further comment)

Option	Description
`-XX:+UseParallelGC`	Use the Parallel garbage collector.
`-XX:MaxGCPauseMillis=<duration in milliseconds>`	Supports configuration of the target max time to pause in GC. This is purely a goal and not guaranteed. It does, however, allow some configuration between throughput (longer pauses) and latency (shorter pauses). Test values between 500ms - 2000ms.
`-XX:GCPauseTimeInterval=<duration in milliseconds>`	Supports configuration of the ideal max time between pauses in GC. This is purely a goal and not guaranteed. It does, however, allow some configuration between throughput (longer pauses) and latency (shorter pauses). Test values between 500ms - 2000ms.
`-XX:GCTimeRatio=<percent value>`	Supports configuration of the optimal ratio between time in GC and application time. This ratio is 1% by default and should not be configured to be greater than 5%.
`-XX:ParallelGCThreads=<size>`	Number of threads to use in parallel operations

The following is an example set of JVM options using the Parallel GC (GC logging omitted):

JVM options with Parallel GC

```
-XX:InitialHeapSize=5g -XX:MaxHeapSize=5g \
-XX:NewSize=2G -XX:MaxNewSize=4G \
-XX:MetaspaceSize=256M -XX:MaxMetaspaceSize=256M \
-XX:MaxTenuringThreshold=1 \
-XX:+UseStringDeduplication \
-XX:+UseParallelGC \
-XX:MaxGCPauseMillis=500
```

Garbage-First GC (G1)

GSA: I've never seen G1 been called G1GC, usually it's just G1

WM: In all the docs I've read, I've seen it called G1, G1 GC and G1GC. Oracle seem to use G1 GC - Happy to revert though.

(Will remove Tuesday 30/6/2020 if no further comment)

- Available since java 7u4 and the default HotSpot collector since java 9 (in resource-rich environments).
- A "Mostly Concurrent Collector", meaning a parallel, concurrent, and incrementally compacting low-pause collector.
- Concurrent collector, which conducts expensive operations concurrently with the application threads.
- Designed for multiprocessor environments with large available memory with the intent of good overall performance without the need to specify additional options.
- Partitions heap into equal sized regions, each with a contiguous range of virtual memory.
- Minimises collection pauses in multiprocessor, memory-rich environments - reduces GC latency.
- Uses an internal adaptive optimization algorithm to manage available heap-space, meaning it's adaptive to fast-memory growth/ spikes in load but also scales efficiently with lower load. With G1GC, this overrides `-Xms` increments.
- Recommended to use G1 GC with heap size set to ensure optimal region sizing. See testing here

The following G1 options are worth considering:

Option	Description
`-XX:+UseG1GC`	Use the G1 garbage collector.
`-XX:MaxGCPauseMillis`	Supports configuration of the target max time to pause in GC. This is purely a goal and not guaranteed. It does, however, allow some configuration between throughput (longer pauses) and latency (shorter pauses). Test values between 500ms - 2000ms.
`-XX:GCPauseTimeInterval`	Supports configuration of the ideal max time between pauses in GC. This is purely a goal and not guaranteed. It does, however, allow some configuration between throughput (longer pauses) and latency (shorter pauses). Test values between 500ms - 2000ms.
`-XX:GCTimeRatio`	Supports configuration of the optimal ratio between time in GC and application time. This ratio is 1% by default and should not be configured to be greater than 5%.
`-XX:ParallelGCThreads=<size>`	Number of threads to use in parallel operations - notably "stop-the-world" activities. It is recommended to set this to the number of logical cores, if this value is less than 8. Otherwise, generally, set it to 5/8 of the number of cores.
`-XX:ConcGCThreads=<size>`	Number of threads to use in concurrent operations - notably marking, which may reduce pauses. It is recommended to be 1/4 of the `-XX:ParallelGCThreads` value.

`-XX:G1NewSizePercent=<%>`
 	
`-XX:G1MaxNewSizePercent=<%>`
	Supports managing the YoungGen space allocation as a percentage of the overall heap size. Ordinarily, this is best left unset to be managed by the G1 adaptive optimisation algorithm, but could be used as a hint, given our understanding of IG's YoungGen usage.

`-XX:G1HeapRegionSize`	Not recommended as it's an ergonomic setting based on heap size. It can be used, however, to manage the heap region size - to fine-tune how objects are allocated - which may be beneficial if large resources are being processed.

The following is an example set of JVM options using the G1 GC (GC logging omitted):

Useful G1 GC configuration options

```
-XX:InitialHeapSize=5g -XX:MaxHeapSize=5g \
-XX:MetaspaceSize=256M -XX:MaxMetaspaceSize=256M \
-XX:MaxTenuringThreshold=1 \
-XX:+UseStringDeduplication \
-XX:+UseG1GC \
-XX:MaxGCPauseMillis=500\
```

Summary

Care should be taken in selecting an appropriate garbage collector, depending on your identified performance targets.

G1 is designed for multiprocessor environments with large available memory with the intent of good overall performance without the need to specify additional options. It is designed to reduce garbage collection through low-GC latency. It is largely self-tuning, with an adaptive optimisation algorithm, but there are a number of options to consider to suite the needs of the protected web application. See Oracle G1 collector documentation and tuning guide.

The Parallel GC aims to improve garbage collection by following a high-throughput strategy, but requires more full GCs. See Oracle Parallel GC documentation.

Generally, the recent addition of the G1 garbage collector is largely promoted by Oracle to satisfy most needs, but testing should be done to compare the results of G1 and the Parallel collector in typical business use cases.

References

1. KB FAQ: IG Performance and Tuning
2. KB BestPractice: Best practice for JVM Tuning (java 7)
3. KB HowTo: How do I collect JVM data for troubleshooting IG/OpenIG (All versions)?
4. KB HowTo: How do I enable Garbage Collector (GC) Logging for IG/OpenIG (All versions)?
5. DZone: Choosing the Right GC
6. Oracle Parallel Garbage Collector
7. Oracle Garbage Collection Tuning Guide
8. Oracle G1GC Tuning Guide
9. JVM Tuning with G1GC by @marknienaber on medium.com (our very own Unknown User (mark.nienaber.secure))
10. Improving G1 Out-of-the-box Performance by Stefan Joansson
11. High Performance at Low Cost – Choose the Best JVM and the Best Garbage Collector for your Needs by Jonatan Kazmierczak
12. A Step-by-step Guide to Java Garbage Collection Tuning by Rafal Kuc
13. Reduce Long GC Pauses by gceasy.io

Related articles

- Tuning IG and the ClientHandler/ ReverseProxyHandler
- Tuning the JVM
- Tuning the Tomcat Container