**Hot-Spot**

1. Uses Generational Heap by default - Objects first allocated in Eden. When Eden space if full a young generation GC is executed and remaining objects are moved to one of the Survivor spaces, at same time the other Survivor space is checked and the objects are either GCd or also moved to first Survivor space. That way only one Survivor space is used. Objects that survived couple of young generation moves are promoted to Tenured space.
2. GC of young generation is very fast, especially if most objects are dead. The GC of Tenured space is many times more expansive thus we don’t want to copy object to Tenured space unless we really know that they will stay for a while
3. Different types of GS
   1. Serial Collection – when all threads are stopped and one of the GC thread does Mark – And – Sweep – uses least of resources but longer delays.
   2. Parallel Collection – When threads stop doing what they are doing and collection is done in parallel by many GC threads. Faster than serial collection but utilized more resources. Also the Tenured collection is still done serially unless the –XX:+UseParallelOldGC is selected.
   3. Concurrent collection – The GC thread is running parallel to application and only pauses the other threads occasionally, for a much shorter period of time than in parallel collection, for marking the root pointers on the heap. Otherwise marking and swipping in mostly done in parallel to application threads. The drawback is that it is a more complex collection process as it it running over possibly changing objects and thus more resources and heap will be used to occasionally copy the objects into local GC memory.
4. The hot-spot has a permanent generation separated from others, where all class and static variable information is stored.

Garbage Collection Selection

|  |  |
| --- | --- |
| –XX:+UseSerialGC | Serial compacting and collection (Default if jvm is started with –client) |
| –XX:+UserParallelGC | Parallel compacting and collection (Default if JVM started with –server). The Tenured space is still collected serially. |
| –XX:+UseParallelOldGC | Parallel compacting and collection of all the spaces. |
| –XX:+UseConcMarkSweepGC  together with  –XX:+UseParNewGC | Concurrent mark – sweep (CMS) collector for tenure collection. Combined with ParNewGC it provides a parallel young generation collection and concurrent tenured collection. |

GC stats

|  |  |
| --- | --- |
| –XX:+PringGC | Basic GC info |
| –XX:+PringGCDetails | More detailed GC info |
| –XX:+PringGCTimeStamps | Output timestamp for GC |

Heap and Generation Sizes

|  |  |
| --- | --- |
| –Xmsn | Initial size, in bytes, of the heap. |
| –Xmxn | Maximum size, in bytes, of the heap. |
| –XX:MinHeapFreeRatio=min (Default: 40) and –XX:MaxHeapFreeRatio=max (Default: 70) | Target range for the proportion of free space to total heap size. These are applied per generation. For example, if minimum is 30 and the percent of free space in a generation falls below 30%, the size of the generation is expanded so as to have 30% of the space free. Similarly, if maximum is 60 and the percent of free space exceeds 60%, the size of the generation is shrunk so as to have only 60% of the space free. |
| –XX:NewSize=n | Default initial size of the new (young) generation, in bytes. |
| –XX:NewRatio=n  (Default: 2 on client JVM, 8 on server JVM) | Ratio between the young and old generations. For example, if n is 3, then the ratio is 1:3 and the combined size of Eden and the survivor spaces is one fourth of the total size of the young and old generations. |
| –XX:SurvivorRatio=n (Default: 32) | Ratio between each survivor space and Eden. For example, if n is 7, each survivor space is one–ninth of the young generation (not one–eighth, because there are two survivor spaces). |
| –XX:MaxPermSize=n | Maximum size of the permanent generation. |
| –XX:+DisableExplicitGC | Disable System.gc calls. Calling System.gc can cause CMS GC collector to fail |

Options for the parallel and parallel compacting collectors

|  |  |
| --- | --- |
| –XX:ParallelGCThreads=n (Default: The number of CPUs) | Number of garbage collector threads. |
| –XX:MaxGCPauseMillis=n | Indicates to the collector that pause times of n milliseconds or less are desired. |
| –XX:GCTimeRatio=n (Default: 99) | Number that sets a goal that 1/(1+n) of the total time be spent on garbage collection. |

CMS collector options

|  |  |
| --- | --- |
| –XX:+CMSIncrementalMode (Default: Disabled) | Enables a mode in which the concurrent phases are done incrementally, periodically stopping the concurrent phase to yield back the processor to the application. |
| –XX:+CMSIncrementalPacing (Default: Disabled) | Enables automatic control of the amount of work the CMS collector is allowed to do before giving up the processor, based on application behavior. |
| –XX:ParallelGCThreads=n (Default: The number of CPUs) | Number of garbage collector threads for the parallel young generation collections and for the parallel parts of the old generation collections. |

**Oracle JRockit**



1. The Jrockit has two heap types. The generational and continuous.
2. In generational heap JRockit has only two generation - Nursery and Tenured. Nursery is split on Keep Area and remaining space. The Keep Aria is where the new objects are allocated. On Nursery GC the object are moved from Keep Area while GC runs on the remaining part of nursery. The Objects are moved to Tenure if they survived second GC.
3. The continuous heap (Default heap) Jrockit does not distinguish between young and old generation and the GC is performed on full heap. This is useful for throughput-oriented apps that would run longer without poses but would stop for longer periods of time for GC as well
4. The Jrockit allocates a separate slice of heap for each thread (TLA – thread local allocation) where threads can allocate space for objects up to predefined size. Any objects larger than the size will be allocated directly on Tenured heap space.
5. Jrockit does not have perm gen thus classes and static variables are allocated space the same way as other objects and thus are GCd the same way as well

Options

|  |  |  |
| --- | --- | --- |
| -XgcPrio:<gcType> |  | Jrockit can choose the GC algorithm dynamically, based on the collected statistics. However we can suggest the preferred type that it will try to comply with. |
|  | throughput  (Default for  -server) | The garbage collector is optimized for application throughput. This means that the garbage collector works as effectively as possible, giving as much CPU resources to the Java threads as possible. This may, however, cause non-deterministic pauses when the garbage collector stops all Java threads for garbage collection.The throughput priority should be used when non-deterministic pauses do not impact the application’s behavior. |
|  | pausetime | The garbage collector is optimized for short pauses. This means that the garbage collection will work concurrently with the Java application when necessary, in order to avoid pausing the Java threads. This inflicts a slight performance overhead to the application, as the concurrent garbage collector demands more system resources (CPU time and memory) than the parallel garbage collector that is used for optimal throughput. The target pause time is by default 200 ms. To change the default pause target -XpauseTarget |
|  | deterministic | The garbage collector is optimized for very short and deterministic pause times. The garbage collector will aim on keeping the garbage collection pauses below a given pause target. How well it will succeed depends on the application and the hardware. For example, a pause target on 30 ms has been verified on an application with 1 GB heap and an average of 30% live data or less at collection time, running on the following hardware:  2 x Intel Xeon 3.6 GHz, 2 MB level 2 cache, 4 GB RAM  4 x Intel Xeon 2.0 GHz, 0.5 MB level 2 cache, 8 GB RAM  Running on slower hardware, with a different heap size and/or with more live data might break the deterministic behavior or cause performance degradation over time, while faster hardware or less live data might allow you to set a lower pause target.  The pause target for deterministic mode is by default 30 ms, and can be changed with the command line option -XpauseTarget |
| -Xgc:<gcType> |  | You can specify the static garbage collection which will disable dynamic algorithm selection and will always run the static GC |
|  | singlecon | Sets a single-spaced (non-generational) concurrent garbage collector. This is a mostly concurrent garbage collector, which means that it will perform most of its garbage collection chores concurrently with the Java application. All objects are maintained in a single space, or “generation.” The singlecon garbage collection trades lower application throughput for minimal pause times. |
|  | gencon | Sets a generational, concurrent garbage collector. With this type of garbage collector, objects are allocated in the young generation (the nursery). When the nursery is full, JRockit JVM stops all Java threads and moves the live objects in the young generation to the old generation. This is a mostly concurrent garbage collector, which means that it will perform most of its garbage collection chores concurrently with the Java application. The gencon garbage collector is better than the singlecon garbage collector for most applications that allocate a lot of small, short-lived objects. The gencon garbage collection trades minimal pause times for larger heap sizes and lower application throughput. |
|  | singlepar  parallel | Sets a single-spaced, parallel garbage collector. A parallel garbage collector stops all Java threads when the heap is full and uses every CPU to perform a complete garbage collection of the entire heap. A parallel collector can have longer pause times than concurrent collectors, but it maximizes throughput. Even on single CPU machines, this maximized performance makes parallel the recommended garbage collector, provided that your application can tolerate the longer pause times.  The singlepar synonym is available in R27.2 and later releases. |
|  | genpar | Sets a generational garbage collector. With this type of garbage collector, objects are first allocated in the young generation. When the nursery is full, JRockit JVM stops all Java threads and performs a parallel nursery collection, that is, it uses all CPU resources available and moves all live objects in the young generation to the old generation. The old generation collector stops all Java threads when the heap is full and a complete parallel garbage collection is performed. This collector will prioritize throughput to pause times.  This collector is generally better than the singlepar garbage collector for applications that allocate a lot of short-lived objects. While it performs more garbage collections than the singlepar collector, the individual pause times of the genpar collector are shorter and it creates less fragmentation in the old generation space.  This option is available in R27.2 and later releases. |

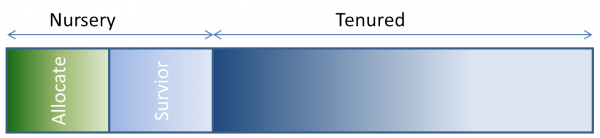
GC Stats

|  |  |
| --- | --- |
| -XgcPause | Prints out pause time caused by gc |
| -XgcReport | More detailed GC report |
| -Xverbose:<type> | Possibly types:  classes – name of loaded classes  codegen – name of each method compiles  cpuinfo – technical info about cpu during startup  exceptions – show thrown exception  exceptions=debug – more detailed exceptions with stack trace  load – name of all loaded libs (jars, native code)  gcpause – same as -XgcPause  gcReport – same as –XgcReport  memdbg – verbose memory outputs on GC and allocation  memory – verbose printout of memory during startup  opt – info on all optimized methods  starttime  systemgc – notifies when gc was done by System.gc() |
| -XverboseDecorations= | Add decorations to verbose printout  level – log level; quiet, error, warn, info, debug, trace  millis – time in millis  millisstart – the time since the start of jvm  nanos – time in nanos  nanosstart – time in nanos since start of jvm  pid – process ID  threaded – thread id  timestamp – time stamp for the process |
| -XverboseLog: | Where to write log |
| -XverboseTimeStamp: | Log how long it took to run the operation (e.g GC) same as –XverboseDecorations=timestamp |

Heap and Generation Sizes

|  |  |
| --- | --- |
| -Xms:n[g|G|m|M|k|K] | Minimum java heap size |
| -Xmx:n[g|G|m|M|k|K] | Maximum heap size |
| -Xns:n[g|G|m|M|k|K] (Default: 50% of heap) | Default nursery size |
| -Xss:n[g|G|m|M|k|K] | Thread stack size |
| -XpauseTarget= | Target in time for GC pause – between 1ms – 5 sec |

**IBM JVM**



1. The IBM JVM has two heap configurations – continuous (default) and generational one.

2. In generational heap - Allocate and Survivor works the same way as Eden and Survivor in Hot-Spot in the way that when Nursery GC is executed the objects are cleaned up from both Survivor and Allocate and the remaining objects in Survivor are then copies to Allocate. at this point Survivor becomes empty and will be switched with Allocate for new object creation.

3. The jvm also has thread local heap area where each thread can allocate space for smaller object, while larger objects are allocated on Tenured space

4. The heap does not have perm gen

Options

|  |  |  |
| --- | --- | --- |
| -Xgcpolicy:<Type> |  | Controls the behavior of the Garbage Collector. |
|  | gencon | The gencon option requests the combined use of concurrent and generational GC to help minimize the time that is spent in any garbage collection pause. |
|  | optavgpause | The optavgpause option reduces the time that is spent in these garbage collection pauses and limits the effect of increasing heap size on the length of the garbage collection pause. Use optavgpause if your configuration has a large heap. Enables concurrent mark. |
|  | optthruput | The optthruput option is the default and delivers high throughput to applications, but at the cost of occasional pauses. Disables concurrent mark. |
|  | subpool | The subpool option (AIX, Linux and IBM i on IBM POWER architecture, and z/OS) uses an improved object allocation algorithm to achieve better performance when allocating objects on the heap. This option might improve performance on large SMP systems |

Reference:

IBM:

<http://publib.boulder.ibm.com/infocenter/javasdk/v6r0/index.jsp?topic=%2Fcom.ibm.java.doc.diagnostics.60%2Fdiag%2Fappendixes%2Fcmdline%2Fcommands_gc.html>

Oracle Jrockit:

<http://download.oracle.com/docs/cd/E13150_01/jrockit_jvm/jrockit/jrdocs/refman/optionX.html#wp999522>

Oracle Hot-Spot:

<http://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=3&ved=0CDQQFjAC&url=http%3A%2F%2Fjava.sun.com%2Fj2se%2Freference%2Fwhitepapers%2Fmemorymanagement_whitepaper.pdf&ei=uKbKTq76D6vJsQLc-aVN&usg=AFQjCNH1c7omTG339FJwX2-R_PBlgMgGzA&sig2=Czm7IPODOrDOoGfsJlKYJQ>

GC article:

<http://blog.dynatrace.com/2011/05/11/how-garbage-collection-differs-in-the-three-big-jvms/>

Performance tuning paper

<http://java.sun.com/performance/reference/whitepapers/tuning.html>