JVM 튜닝

Basic IT 2015. 9. 10. 14:44

1. JVM Options

http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html#BehavioralOptions

http://wiki.ex-em.com/index.php/JVM_Options#UseCMSInitiatingOccupancyOnly

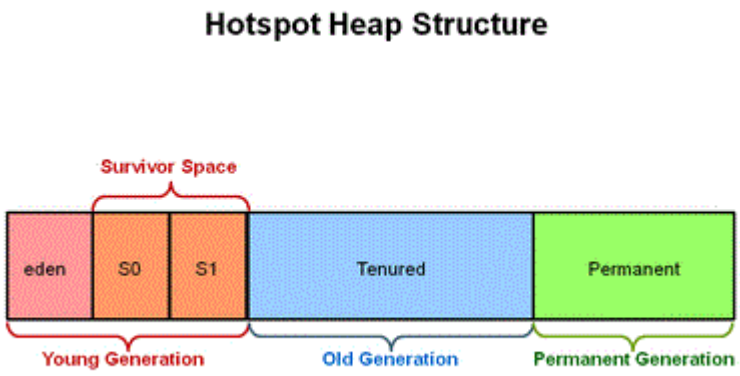
2. G1 GC

Java process 엔진이 대용량의 실시간 처리를 요구할 때, 성능은 JVM의 GC 시간에 의존적일 수 밖에 없다. 특히 64bit 환경에서는 메모리 제약 사항이 없어졌다고 볼 수 있기 때문에 large heap memory를 cleaning하는 시간을 단축하는 것이 전체 성능을 개선하는 중요 key라고 할 수 있다. 전통적인 Serial, Parallel 그리고 CMS로는 이 성능을 충족하기에는 매우 어려운 것이 현실이다. 그렇다고 사용 JVM인 azul을 쓰기에도 비용의 제약사항이 있다.

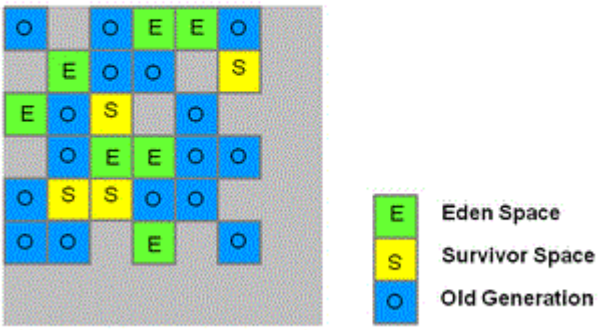
최근 linux 환경(Min / Max heap size = 50g)에서 수십만 TPS를 요구하는 시스템을 튜닝하기 위해 CMS와 Parallel 옵션을 통해 테스트를 해보면 Parallel GC로 10초 내의 Full GC를 시간당 5~7회 정도 발생하는 규모로 튜닝을 하였다. 또 다른 개선책이 있는지를 확인하는 과정에서 G1GC를 알게되었으며, G1GC로 변경 테스트 전에 이에 대한 주요 내용을 정리하고자 한다. 좀더 자세한 사항은 테스트 후에 이 해당 블로그에 업데이트 할 예정이다.

G1 Garbage Collector

G1 GC는 대용량 메모리가 탑재된 multi-processor용을 위한 Garbage Collector이며, 높은 처리량의 시스템을 위한 용도로 사용된다. 전통적인 Garbage Collector(Serial, Parallel 및 CMS)는 세개의 메모리 영역의 구조를 갖는데 비해 G1은 다른 접근 방식을 갖는다:



G1 Heap Allocation



G1 GC의 경우 Heap은 같은 크기의 heap region으로 분활하며, 이전 Garbage Collector와 마찬가지로 특정 region별로 역할을 부여하여 특정한 object만이 거주하도록 하였다. GC의 수행 시점은 CMS collector와 유사한 방식으로 진행된다. 즉, 1) 전체 heap에 alive한 object를 식별하기 위해 concurrent global marking phase를 수행한 뒤, 2) Mark phase가 완료되면 G1은 어떤 region이 거의 empty 인지 여부를 식별할 수 있으며, 이러한 region을 먼저 collect를 한다. 일반적으로 큰 용량의 free space를 확보할 수 있게 해주기 때문에 이 Collector의 이름이 Garbage-First라 부르는 이유이다. 즉, G1은 collection와 compaction의 operation을 집중하여 사용자에게 목표로 하는 pause time을 달성하고자 한다. 정리하자면,

G1 GC는

- 전체 heap을 Region이라는 영역으로 분활하여 관리하며, Eden, Survivor, Old에 거주했던 objects들은 G1에서는 이 Region에 상주한다. 즉, Region이 곧 Eden, Survivor 또는 Old일 수 있다.
- Region의 목표 수치는 2048으로 분활된다. 즉, 8G의 Heap이라면 하나의 Region의 크기는 4MB(8192MB/2048 = 4MB)이다.
- Region의 크기는 조정가능(1MB~32MB)하지만 권장하니 않는다(-XX:G1RegionSize).
- G1에는 전통적인 type(Eden, Survior 및 Old Generation) 외에 Homongous Region과 Available / Unused Region이 추가로 존재하며, 1) Available / Unused Region은 아직 사용되지 않은 영역을 의미하며, 2) Homongous Region은 단일 객체의 크기가 JVM 실행시 할당된 하나의 Region의 크기에 1/2보다 큰 대용량 객체를 저장하는 영역이다. 만일 이와 같은 객체가 존재한다는 것은 애플리케이션 설계의 이슈가 있는지를 확인해야 할 것이다.

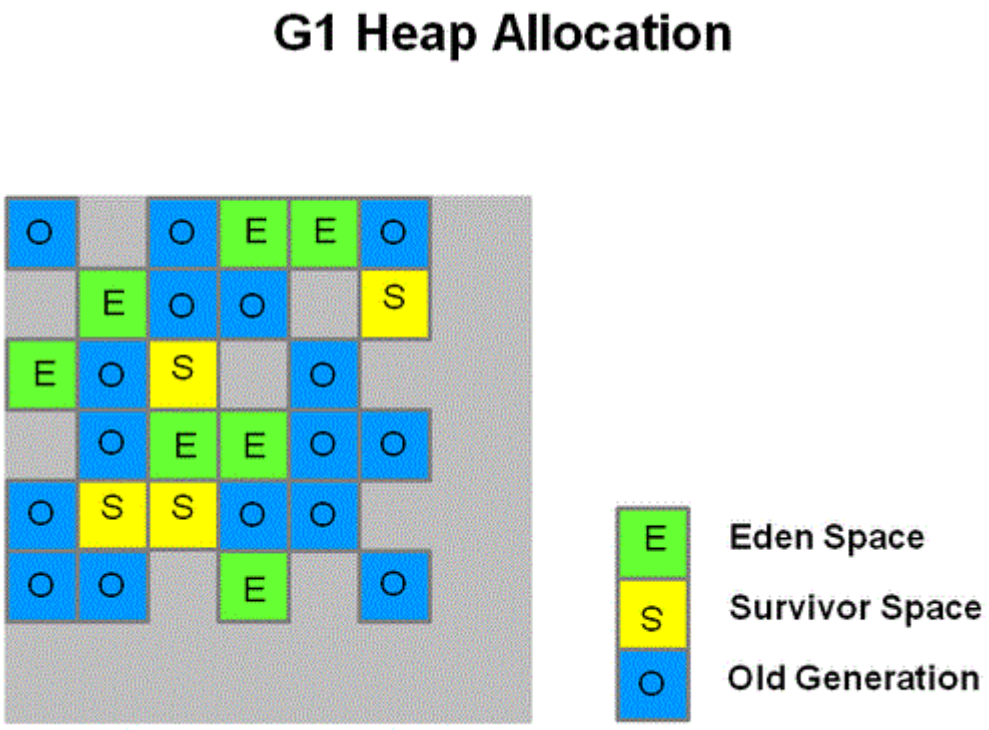
The G1 Garbage Collector Step by Step

1. **G1 Heap Structure :**

Heap은 고정 크기의 Region으로 분리된 하나의 메모리 구조이며, Region의 크기는 JVM이 실행될 때 결정된다. 일반적으로 Region의 크기는 1~32MB의 크기로 대략 2000개의 Region으로 분활한다.

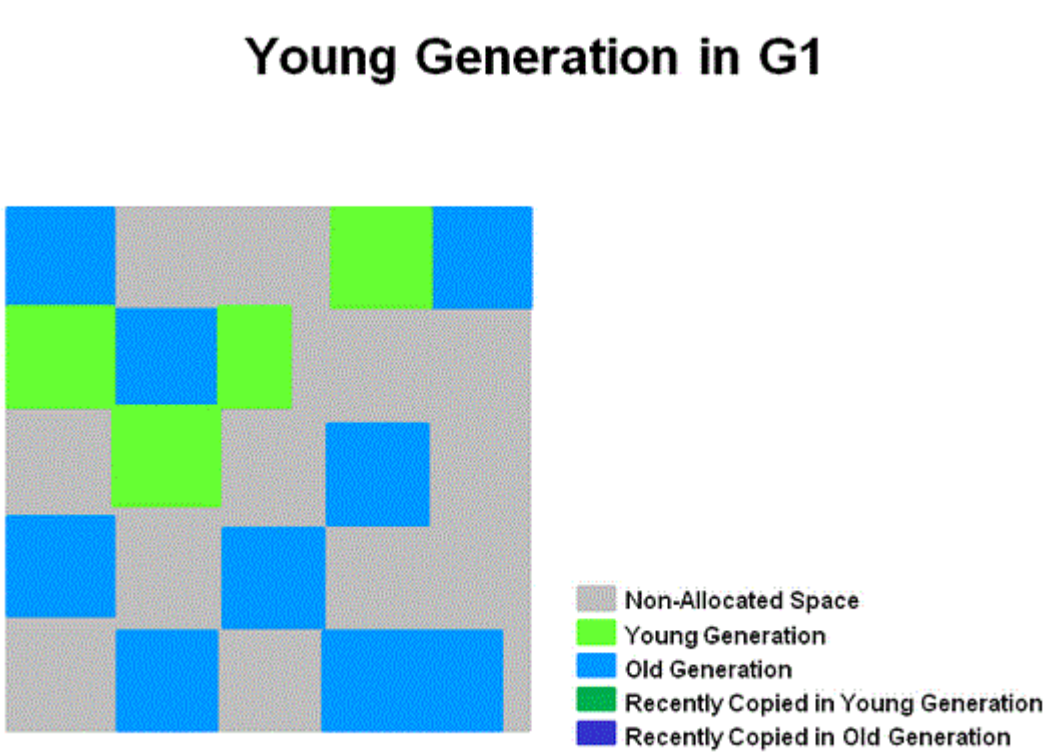
2. **G1 Heap Allocation**

하나의 메모리 공간에 분활된 Region은 Eden, Survivor 및 Old Generation을 위한 공각으로 논리적으로 매핑되며, Live objects들은 Region간의 이동을 할 수 있으며, Region은 parallel 및 애플리케이션 중지 없이 collection을 할 수 있도록 설계되었다.



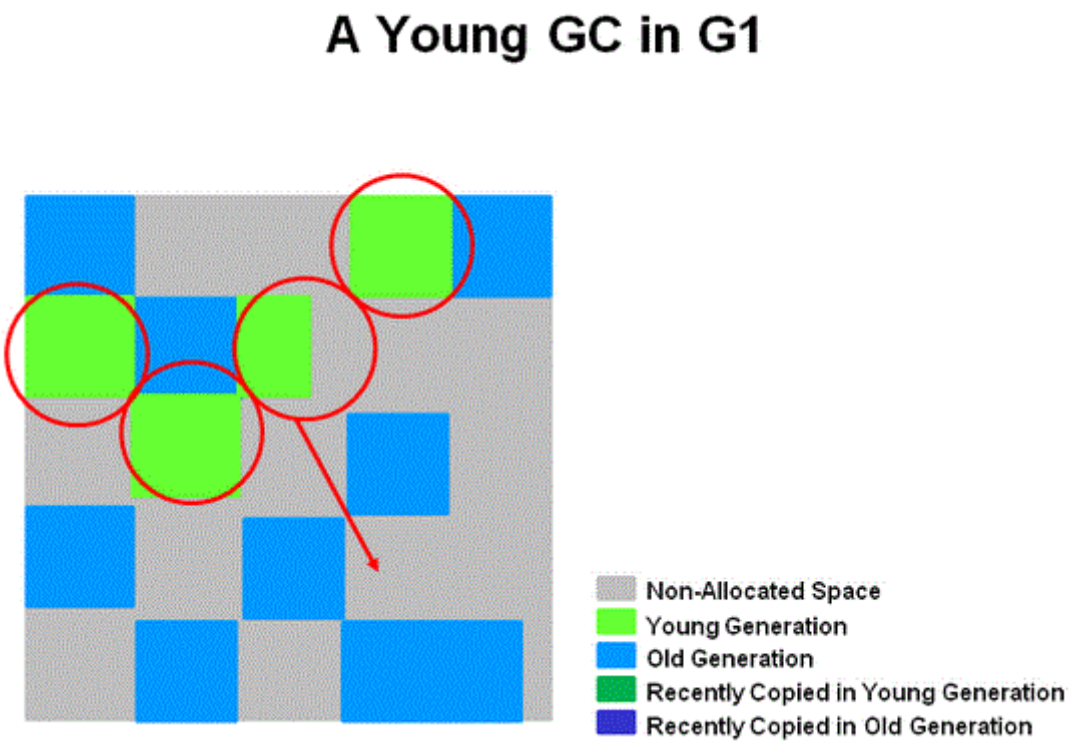
3. **Young Generation in G1**

Heap은 대략 2000개의 Region으로 분활되며, 하나의 Region의 크기는 1~32MB의 크기로 할당된다. 아래 그림의 파란색 Region이 Old Generation이며, 초록색 Region이 Young Generation의 객체가 저장되어 있다.

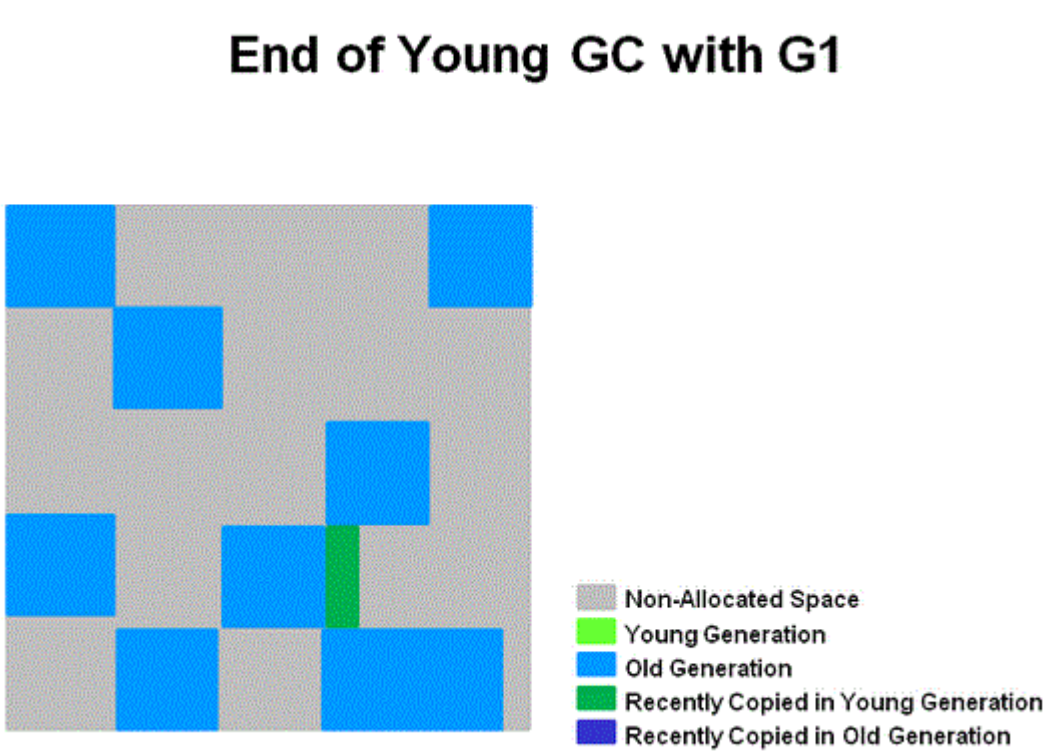


4. **A Young GC in G1**

Live 객체들이 하나 이상의 Survivor region으로 이동(Copy / Move) 되며, 이를 Evacuation Pauses(=Minor GC)이라한다. 노화 임계값(aging threshold)이 충족되는 경우, 이 객체의 일부가 Old Generation region으로 승격된다. 이 단계는 STW(Stop-The-World)이며, Eden / Survivor 크기는 다음 Young GC를 위해 계산되어 지며, 사용자가 옵션으로 지정한 pause time과 G1 GC 내부에서 사용되는 휴리스틱 알고리즘에 의해 Evacuation Pauses의 GC 대상 Region을 선택한다. G1 GC의 목표는 실시간성 이기 때문에 이 단계는 multi-thread로 동작함.



5. **End of a Young GC with G1**



Live 객체는 Survivor나 Old Generation region으로 이동되며, 최근에 승격된 객체는 진한 청색과 진한 녹색으로 영역임을 확인할 수 있다. 즉, G1의 Young Generation은,

- **Heap은 단일 메모리 공간은 Region으로 분할**
- **Young generation memory는 non-contiguous region으로 구성되며, 이는 필요 시 크기 조정을 용이하게 함.**
- **Young GC는 Stop the world 구간이며, multiple thread를 사용하여 병렬로 처리**
- **Live 객체는 새로운 Survivor나 old generation region으로 복사**

Old Generation Collection with G1

CMS collector와 같이 G1 collector는 Old Generation 객체를 위해 low pause collector로 설계되었음.

G1 Collection Phases - Concurrent Marking Cycle Phases

G1 collector는 Heap의 Old Generation에 대해 다음 단계를 수행하며, 이는 Young Generation Collection의 일 부분으로 포함된다. 이는 -XX:InitiatingHeapOccupancyPercent(IHOP)에서 정의한 수치가 넘아가면 동작된다.

Phase	Description
(1) Initial Mark *(Stop the World Event)*	이 단계는 Young GC 시점에 수행되기 때문에 Stop the world 구간이며, Old Generation내의 객체의 Reference를 가지는 Survivor Regions(Root Region)을 Mark.
(2) Root Region Scanning	Old Generation의 Reference를 가지는 Survivor Region을 검색하며, Young GC 전에 수행이 완료되어야 함. multi-thread로 동작하며 애플리케이션과 concurrent하게 동작
(3) Concurrent Marking	Heap내의 모든 reachable / live 객체를 마킹하며, Multi-thread이며 애플리케이션과 Concurrent하게 동작하며, Young GC와 동시에 실행 가능하다.
(4) Remark *(Stop the World Event)*	Heap내의 live 객에 마킹을 완료하는 단계로 G1은 SATB(Snapshot-At-The-Beginning) 알고리즘을 사용함으로 CMS collector보다 빠르다.

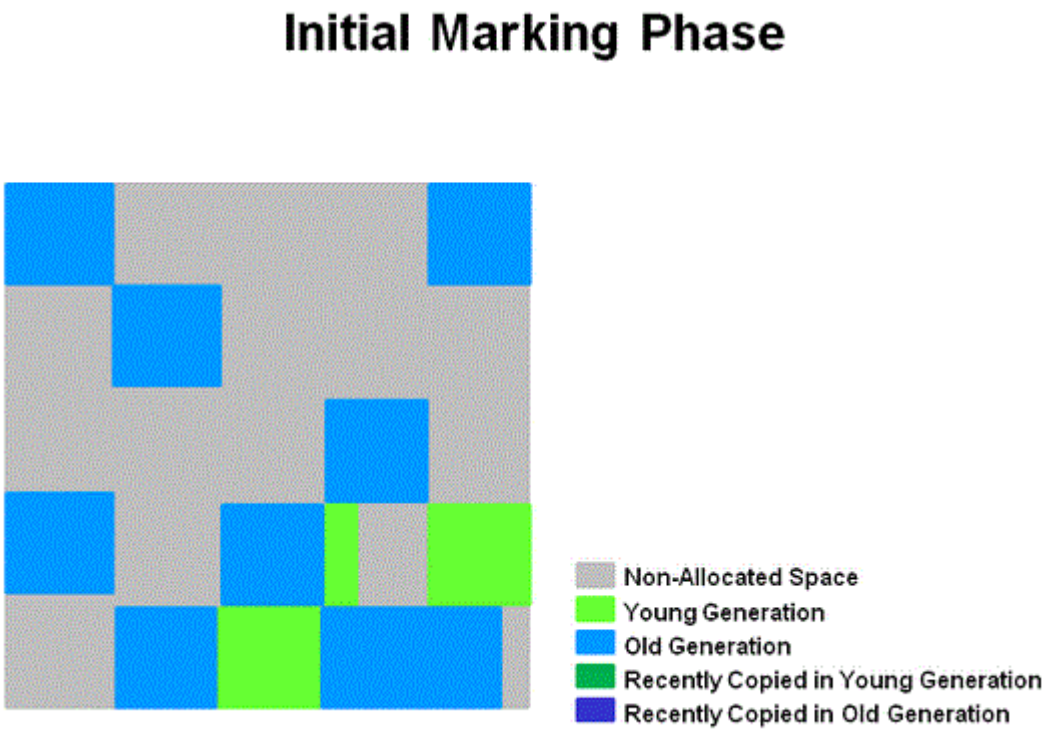
Phase	Description
(5) Cleanup *(Stop the World Event and Concurrent)*	◦ Performs accounting on live objects and completely free regions. (Stop the world) ◦ Remembered Set을 정리 (Stop the world) ◦ Empty Region을 재정리하여 이 Region을 Free List에 추가(Concurrent)
(*) Copying *(Stop the World Event)*	Stop-the-World 구간으로 live 객체를 새로운 unused region으로 복사나 이동. ◦ [GC pause (young)] : young region ◦ [GC pause (mixed)]: young / old generation region

*** Remember Set(RSet):** Reference를 가진 객체가 어느 Region에 할당되어 있는지 알기 쉽게 하기 위한 자료 구조(Live Object 여부를 판단하기 위한 근거).

G1 Old Generation Collection Step by Step

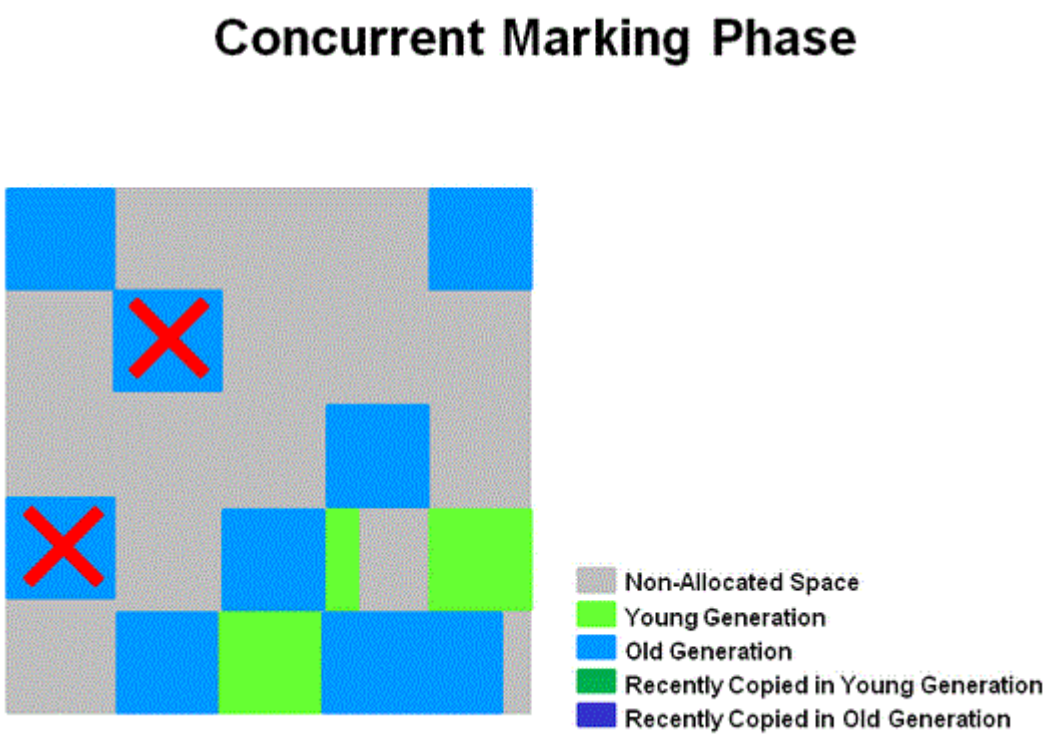
6. **Initial Marking Phase**

live 객체에 대한 Initial marking 단계는 Young GC와 함께 수행되며, "`GC pause (young)(inital-mark)`"로 로깅됨.



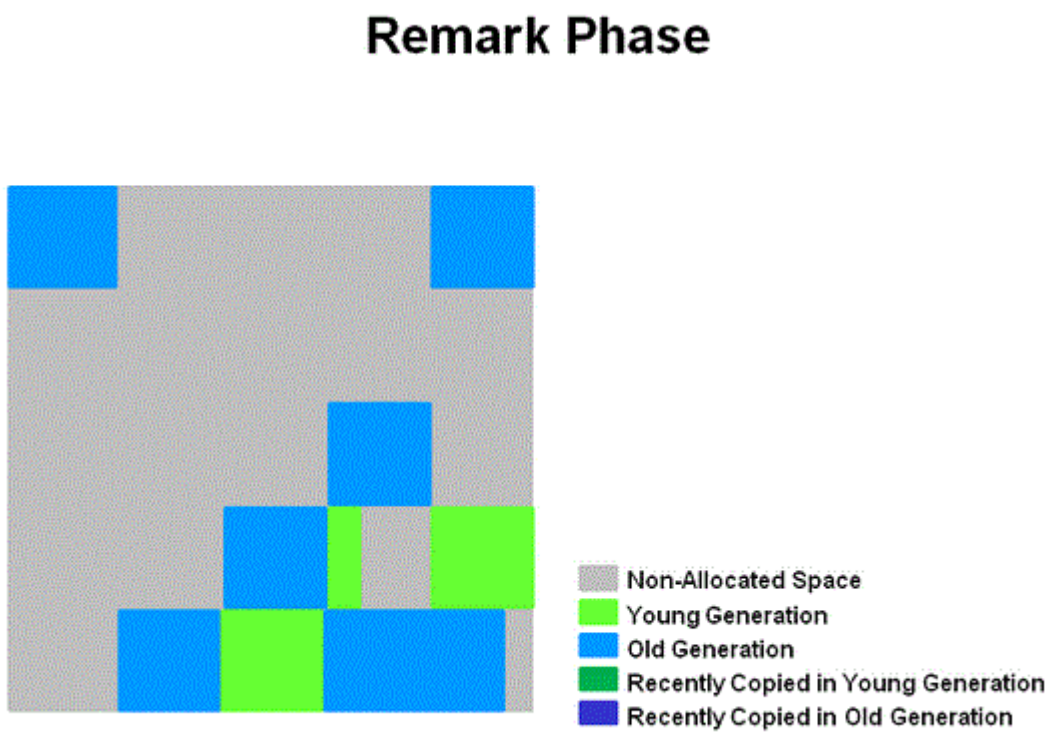
7. **Concurrent Marking Phase**

Empty Region이 발견되는 경우(아래 그림에서 X), Remark Phase에서 즉시 삭제된다.



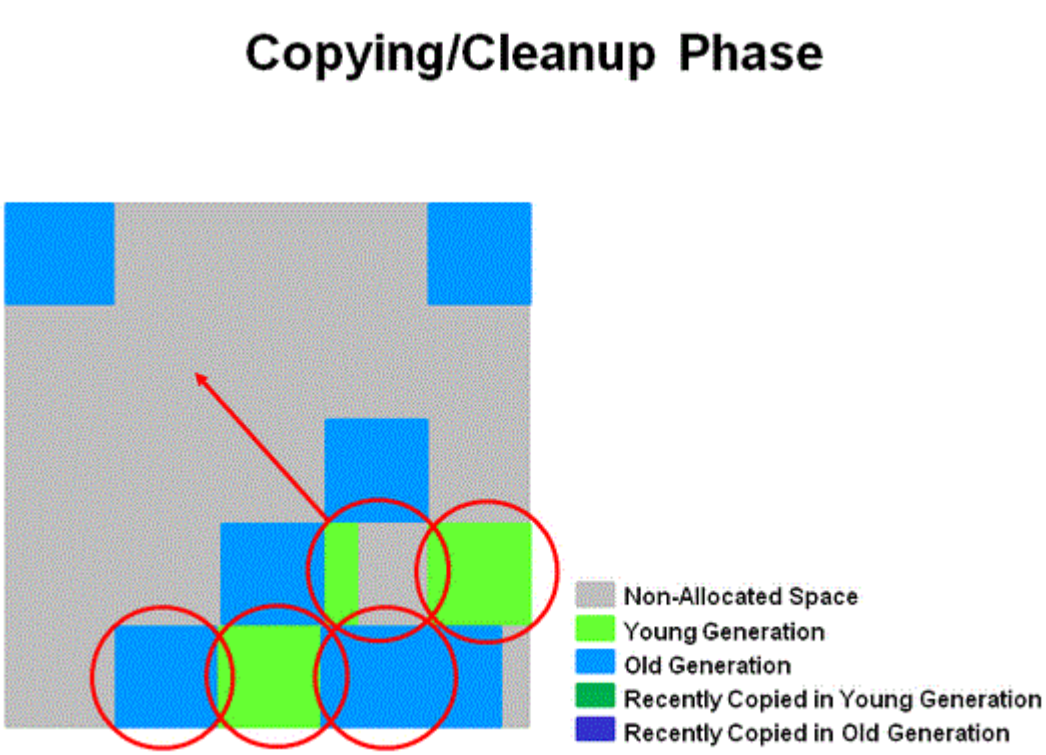
8. **Remark Phase**

Empty region은 제거되어 반한되며, 모든 Region에 대해 Region Liveness가 재계산된다.



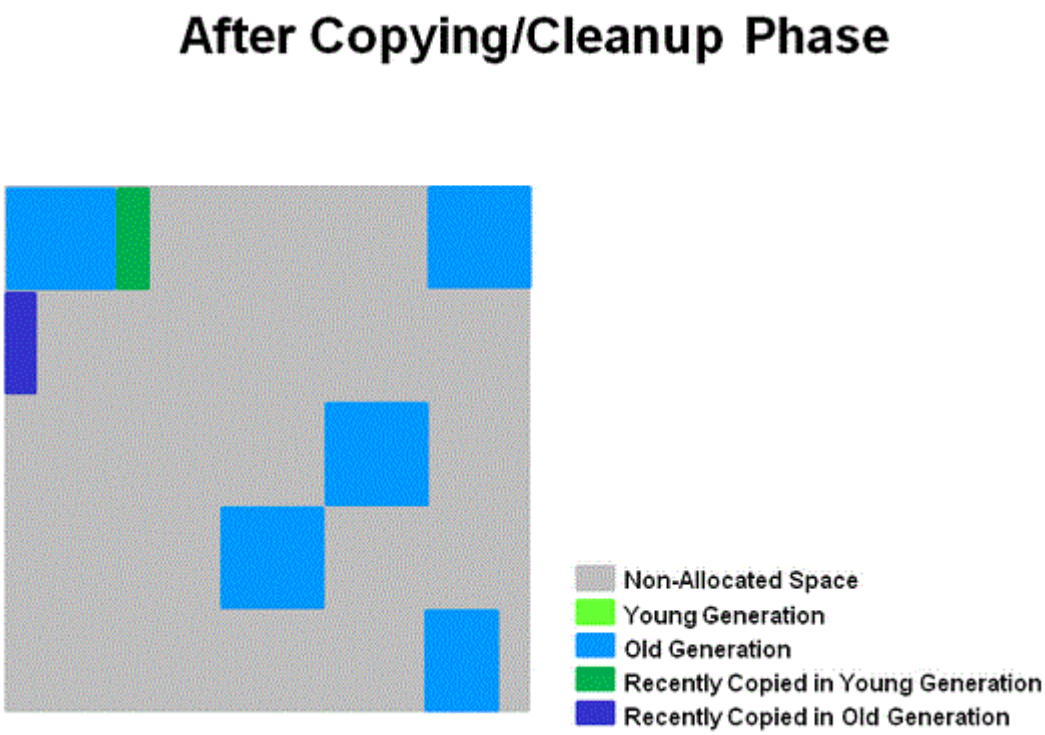
9. **Copying/Cleanup Phase**

G1은 가장 빠르게 collect되어지는 Region 즉, liveness가 가장 적은 region을 선택하며, 이는 Young GC와 같은 시간에 수집된다. 로그에는 [GC pause (mixed)]로 기록되며, Yong / old Generation이 동시에 collecting된다.



10. **After Copying/Cleanup Phase**

다음 그림의 진한 청색 및 녹색과 같이 선택된 Region은 collect되고 compact된다.



Summary of Old Generation GC

Old 영역에 대한 G1 GC의 중요한 내용은:

- **Concurrent Marking Phase**
  - **Liveness 정보는 애플리케이션 실행과 동시에 계산되며, 이 정보는 evacuation phase동안 어느 region을 reclaim하는 것이 최선책인지를 판별한다.**
  - **CMS와 같은 sweeping phase가 존재하지 않는다.**
- **Remark Phase**
  - **CMS에서 사용되는 것 보다 빠른 SATB(Snapshot-at-the-Beginning) 알고리즘을 사용**
  - **완벽하게 empty region으로 reclaim됨.**

- **Copying/Cleanup Phase**
  - **Young 영역과 Old 영역이 동시에 reclaim됨.**
  - **Old generation region은 liveness 정보를 참조하여 선택됨**

Best Practices

G1 GC는 다른 Garbage Collector와 다르게 별도의 옵션 조정없이 사용가능하며, 만일 옵션 조정을 하고자 한다면 다음의 주요 옵션의 조정을 고려해 보자:

-XX:+UseG1GC : Garbage First(G1) Collector 사용 - 기본 옵션

-XX:G1HeapRegionSize=n : 하나의 G1 Region의 크기 설정(1MB ~ 32MB). 최대 heap size를 대략 2048 region으

-XX:MaxGCPauseMillis=200 : 목표로 하는 maximum pause time을 설정.

-XX:ParallelGCThreads=n: STW Worker thread 수를 지정. 논리 processor의 수의 값을 설정한다. 오라클에서는

-XX:ConcGCThreads=n: Parallel marking thread의 수를 지정. ParallelGCThread에서 지정한 값의 1/4을 정의한

-XX:InitiatingHeapOccupancyPercent=45: marking cycle이 실행된 Java heap occupancy threshold의 값 설정.

-XX:G1MixedGCLiveThresholdPercent=65

Sets the occupancy threshold for an old region to be included in a mixed garbage collection cycle. The default occupancy is 65 percent. This is an experimental flag. See "How to unlock experimental VM flags" for an example. This setting replaces the -XX:G1OldCSetRegionLiveThresholdPercent setting. This setting is not available in Java HotSpot VM, build 23.

-XX:G1HeapWastePercent=10

Sets the percentage of heap that you are willing to waste. The Java HotSpot VM does not initiate the mixed garbage collection cycle when the reclaimable percentage is less than the heap waste percentage. The default is 10 percent. This setting is not available in Java HotSpot VM, build 23.

-XX:G1MixedGCCountTarget=8

Sets the target number of mixed garbage collections after a marking cycle to collect old regions with at mostG1MixedGCLIveThresholdPercent live data. The default is 8 mixed garbage collections. The goal for mixed collections is to be within this target number. This setting is not available in Java HotSpot VM, build 23.

-XX:G1OldCSetRegionThresholdPercent=10

Sets an upper limit on the number of old regions to be collected during a mixed garbage collection cycle. The default is 10 percent of the Java heap. This setting is not available in Java HotSpot VM, build 23.

-XX:G1ReservePercent=10

Sets the percentage of reserve memory to keep free so as to reduce the risk of to-space overflows. The default is 10 percent. When you increase or decrease the percentage, make sure to adjust the total Java heap by the same amount. This setting is not available in Java HotSpot VM, build 23.

How to Unlock Experimental VM Flags

To change the value of experimental flags, you must unlock them first. You can do this by setting `-XX:+UnlockExperimentalVMOptions` explicitly on the command line before any experimental flags. For example:

```
> java -XX:+UnlockExperimentalVMOptions -XX:G1NewSizePercent=10 -XX:G1MaxNewSizePercent=75  G1test
```

Recommendations

When you evaluate and fine-tune G1 GC, keep the following recommendations in mind:

Young Generation Size: Avoid explicitly setting young generation size with the `-Xmn` option or any or other related option such as `-XX:NewRatio`. Fixing the size of the young generation overrides the target pause-time goal.

Pause Time Goals: When you evaluate or tune any garbage collection, there is always a latency versus throughput trade-off. The G1 GC is an incremental garbage collector with uniform pauses, but also more overhead on the application threads. The throughput goal for the G1 GC is 90 percent application time and 10 percent garbage collection time. When you compare this to Java HotSpot VM's throughput collector, the goal there is 99 percent application time and 1 percent garbage collection time. Therefore, when you evaluate the G1 GC for throughput, relax your pause-time target. Setting too aggressive a goal indicates that you are willing to bear an increase in garbage collection overhead, which has a direct impact on throughput. When you evaluate the G1 GC for latency, you set your desired (soft) real-time goal, and the G1 GC will try to meet it. As a side effect, throughput may suffer.

Taming Mixed Garbage Collections: Experiment with the following options when you tune mixed garbage collections. See "Important Defaults" for information about these options:

`-XX:InitiatingHeapOccupancyPercent`
For changing the marking threshold.

`-XX:G1MixedGCLiveThresholdPercent` and `-XX:G1HeapWastePercent`
When you want to change the mixed garbage collections decisions.

`-XX:G1MixedGCCountTarget` and `-XX:G1OldCSetRegionThresholdPercent`
When you want to adjust the CSet for old regions.

Overflow and Exhausted Log Messages

로그에서 to-space overflow/exhausted 메시지가 출력된다면, G1 GC는 survivor나 승격된 객체용으로 메모리가 부족한 상황이다. 즉, 메모리가 최대치이기 때문에 Java heap이 확장할 수 없는 상황이다:

```
924.897: [GC pause (G1 Evacuation Pause) (mixed) (to-space exhausted), 0.1957310 secs]
```

또는

```
924.897: [GC pause (G1 Evacuation Pause) (mixed) (to-space overflow), 0.1957310 secs]
```

이 문제를 해결하기 위해서는, "`-XX:G1ReservePercent`"옵션의 값을 증가하여 to-space용으로 확보한 메모리의 양을 증가시키고, "`-XX:InitiatingHeapOccupancyPercent`"의 값을 줄여서 빠른 시점에 marking cycle이 실행되도록 한다. 또한, "`-XX:ConcGCThreads`"의 값을 증가하여 parallel marking thread의 수를 증가시켜라.

Humongous Objects and Humongous Allocations

For G1 GC, any object that is more than half a region size is considered a "Humongous object". Such an object is allocated directly in the old generation into "Humongous regions". These Humongous regions are a contiguous set of regions. `StartsHumongous` marks the start of the contiguous set and `ContinuesHumongous` marks the continuation of the set.

Before allocating any Humongous region, the marking threshold is checked, initiating a concurrent cycle, if necessary.

Dead Humongous objects are freed at the end of the marking cycle during the cleanup phase also during a full garbage collection cycle.

In-order to reduce copying overhead, the Humongous objects are not included in any evacuation pause. A full garbage collection cycle compacts Humongous objects in place.

Since each individual set of StartsHumongous and ContinuesHumongous regions contains just one humongous object, the space between the end of the humongous object and the end of the last region spanned by the object is unused. For objects that are just slightly larger than a multiple of the heap region size, this unused space can cause the heap to become fragmented.

If you see back-to-back concurrent cycles initiated due to Humongous allocations and if such allocations are fragmenting your old generation, please increase your `-XX:G1HeapRegionSize` such that previous Humongous objects are no longer Humongous and will follow the regular allocation path.

Conclusion
G1 GC is a regionalized, parallel-concurrent, incremental garbage collector that provides more predictable pauses compared to other HotSpot GCs. The incremental nature lets G1 GC work with larger heaps and still provide reasonable worst-case response times. The adaptive nature of G1 GC just needs a maximum soft-real time pause-time goal along-with the desired maximum and minimum size for the Java heap on the JVM command line.

==========

reference site:

http://www.oracle.com/technetwork/articles/java/g1gc-1984535.html

https://docs.oracle.com/cd/E40972_01/doc.70/e40973/cnf_jvmgc.htm#autoId2
https://blogs.oracle.com/jonthecollector/entry/top_10_gc_reasons
http://initproc.tistory.com/archive/20130814
https://software.intel.com/en-us/blogs/2014/06/18/part-1-tuning-java-garbage-collection-for-hbase
https://databricks.com/blog/2015/05/28/tuning-java-garbage-collection-for-spark-applications.html
http://yckwon2nd.blogspot.com/2014/04/garbage-collection.html

Garbage First Garbage Collector Tuning

The Garbage First Garbage Collector (G1 GC) is the low-pause, server-style generational garbage collector for Java HotSpot VM. The G1 GC uses concurrent and parallel phases to achieve its target pause time and to maintain good throughput. When G1 GC determines that a garbage collection is necessary, it collects the regions with the least live data first (garbage first).

A garbage collector (GC) is a memory management tool. The G1 GC achieves automatic memory management through the following operations:

- Allocating objects to a young generation and promoting aged objects into an old generation.
- Finding live objects in the old generation through a concurrent (parallel) marking phase. The Java HotSpot VM triggers the marking phase when the total Java heap occupancy exceeds the default threshold.
- Recovering free memory by compacting live objects through parallel copying.

Here, we look at how to adapt and tune the G1 GC for evaluation, analysis and performance—we assume a basic understanding of Java garbage collection.

The G1 GC is a regionalized and generational garbage collector, which means that the Java object heap (heap) is divided into a number of equally sized regions. Upon startup, the Java Virtual Machine (JVM) sets the region size. The region sizes can vary from 1 MB to 32 MB depending on the heap size. The goal is to have no more than 2048 regions. The eden, survivor, and old generations are logical sets of these regions and are not contiguous.

The G1 GC has a pause time-target that it tries to meet (soft real time). During young collections, the G1 GC adjusts its young generation (eden and survivor sizes) to meet the soft real-time target. During mixed collections, the G1 GC adjusts the number of old regions that are collected based on a target number of mixed garbage collections, the percentage of live objects in each region of the heap, and the overall acceptable heap waste percentage.

The G1 GC reduces heap fragmentation by incremental parallel copying of live objects from one or more sets of regions (called Collection Set (CSet)) into different new region(s) to achieve compaction. The goal is to

reclaim as much heap space as possible, starting with those regions that contain the most reclaimable space, while attempting to not exceed the pause time goal (garbage first).

The G1 GC uses independent Remembered Sets (RSets) to track references into regions. Independent RSets enable parallel and independent collection of regions because only a region's RSet must be scanned for references into that region, instead of the whole heap. The G1 GC uses a post-write barrier to record changes to the heap and update the RSets.

Garbage Collection Phases

Apart from evacuation pauses (described below) that compose the stop-the-world (STW) young and mixed garbage collections, the G1 GC also has parallel, concurrent, and multiphase marking cycles. G1 GC uses the Snapshot-At-The-Beginning (SATB) algorithm, which takes a snapshot of the set of live objects in the heap at the start of a marking cycle. The set of live objects is composed of the live objects in the snapshot, and the objects allocated since the start of the marking cycle. The G1 GC marking algorithm uses a pre-write barrier to record and mark objects that are part of the logical snapshot.

Young Garbage Collections

The G1 GC satisfies most allocation requests from regions added to the eden set of regions. During a young garbage collection, the G1 GC collects both the eden regions and the survivor regions from the previous garbage collection. The live objects from the eden and survivor regions are copied, or evacuated, to a new set of regions. The destination region for a particular object depends upon the object's age; an object that has aged sufficiently evacuates to an old generation region (that is, promoted); otherwise, the object evacuates to a survivor region and will be included in the CSet of the next young or mixed garbage collection.

Mixed Garbage Collections

Upon successful completion of a concurrent marking cycle, the G1 GC switches from performing young garbage collections to performing mixed garbage collections. In a mixed garbage collection, the G1 GC optionally adds some old regions to the set of eden and survivor regions that will be collected. The exact number of old regions added is controlled by a number of flags that will be discussed later (see "Taming Mixed GCs"). After the G1 GC collects a sufficient number of old regions (over multiple mixed garbage collections), G1 reverts to performing young garbage collections until the next marking cycle completes.

Phases of the Marking Cycle

The marking cycle has the following phases:

Initial mark phase: The G1 GC marks the roots during this phase. This phase is piggybacked on a normal (STW) young garbage collection.

Root region scanning phase: The G1 GC scans survivor regions of the initial mark for references to the old generation and marks the referenced objects. This phase runs concurrently with the application (not STW) and must complete before the next STW young garbage collection can start.

Concurrent marking phase: The G1 GC finds reachable (live) objects across the entire heap. This phase happens concurrently with the application, and can be interrupted by STW young garbage collections.

Remark phase: This phase is STW collection and helps the completion of the marking cycle. G1 GC drains SATB buffers, traces unvisited live objects, and performs reference processing.

Cleanup phase: In this final phase, the G1 GC performs the STW operations of accounting and RSet scrubbing. During accounting, the G1 GC identifies completely free regions and mixed garbage collection candidates. The cleanup phase is partly concurrent when it resets and returns the empty regions to the free list.

Posted by imp51
트랙백 0개, 댓글 0개가 달렸습니다

댓글을 달아 주세요

: 이름

: 패스워드

http:// : 홈페이지

비밀글

댓글 달기

이전 1 … 31 32 33 34 35 36 37 38 39 … 96 다음