GC(Garbage Collection) - 2

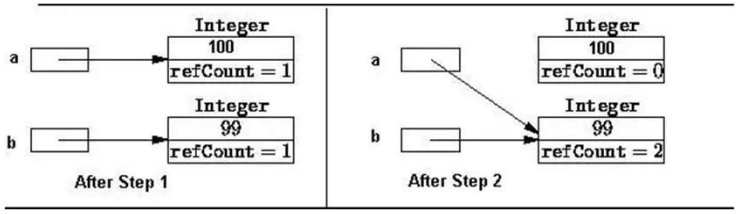
Java

목차

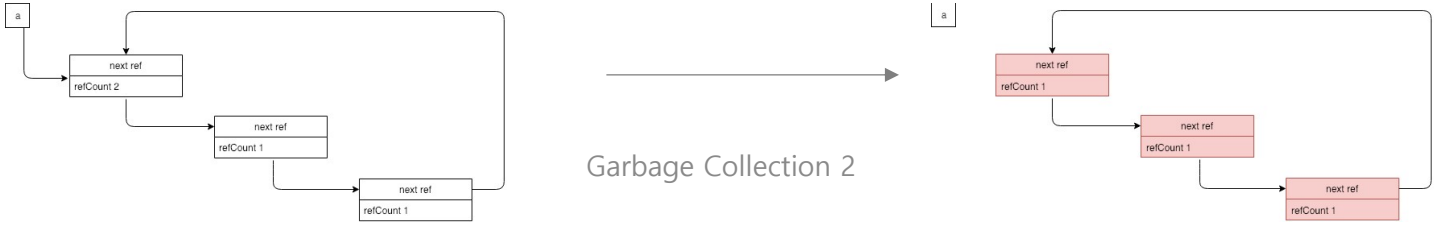
Garbage Collection Process

- **GC Algorithm**
  - Garbage Collection Algorithm이란, Garbage를 수집하는 알고리즘으로, 수집하기 위한Garbage 객체인지 남겨야 할 객체인지 구별하는 알고리즘을 말한다.
  - **Reference Counting Algorithm**

    LISP에서 사용되었던 최초 알고리즘이자 Python, PHP등에 사용되는 알고리즘. 각 객체마다 Reference Count를 관리하여 Reference Count가 0이 되면 GC를 수행한다.



*연결리스트 같은 구조에서 나타나는 순환참조 문제로 인해 Memory Leak 발생 가능성이 있다.



Garbage Collection Process

- **Mark-and-Sweep Algorithm, Mark-and-Compact Algorithm, Copying Algorithm**

  이 세가지 알고리즘은 Reachable 객체에 마킹을 한 후 마킹 되지 않은 Unreachable 객체를 수집하는 두가지 순서로 진행되는데 마킹하는 방법(Reachable 객체 찾아내기)은 동일하고, 마킹 후 객체들을 관리하는 방법이 각각 나뉜다. (Root Set 지정 및 관련한 간단한 내용은 GC1 PPT 참조) 아래와 같이 Reachable 객체에 마킹할 때 STW가 발생한다.

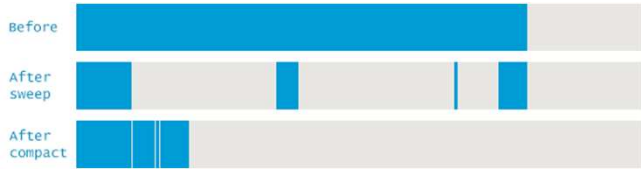
Garbage Collection Process

- **Mark-and-Sweep Algorithm**

  Mark한 객체를 제외한 나머지 객체를 모두 해제하는 방법. Sweep이 완료되면 살아남은 객체의 마킹 정보를 모두 초기화. GC 후 비어 있는 메모리 공간 때문에 단편화 발생 가능 (GC1 PPT 참고)

- **Mark-and-Compact Algorithm**

  Mark-and-Sweep을 보완한 알고리즘으로, Sweep으로 인한 단편화를 없애기 위해 Compaction (메모리를 차례대로 재할당) 단계를 하나 더 추가. 하지만 Compaction 시 남은 객체들의 Reference 정보를 업데이트해야 하는 오버헤드가 추가 된다.



- **Copying Algorithm**

  General Algorithm의 전신으로, Mark-and-Sweep을 보완한 또다른 알고리즘. Heap을 Active, Inactive 영역 (논리적) 으로 나눈 후 객체는 Active 영역에 할당한다. Active 영역이 모두 차면 GC를 수행하는데 먼저 살아 남은 객체를 Inactive에 Copy하고(Suspend 발생) Active에 남은 Garbage 객체를 제거하면 Active는 Free Memory가 된다. Active와 Inactive 영역을 서로 바꾸고 다시 진행한다(Scavenge)

Garbage Collection Process

- **Generational Algorithm**
  GC 1 PPT 참고

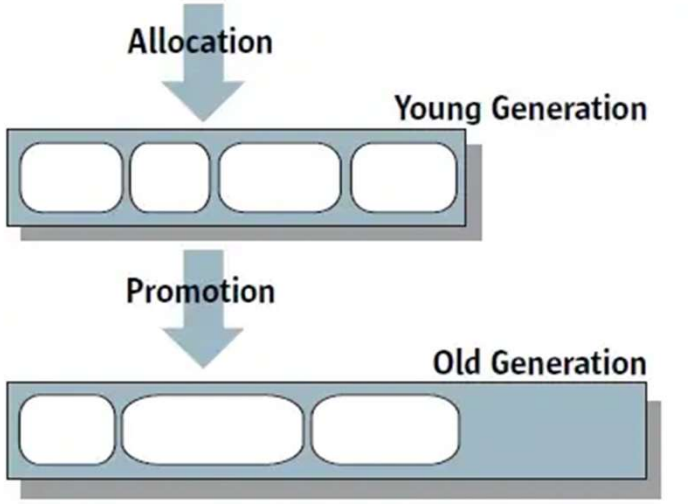


Figure 1. Generational garbage collection

Garbage Collector

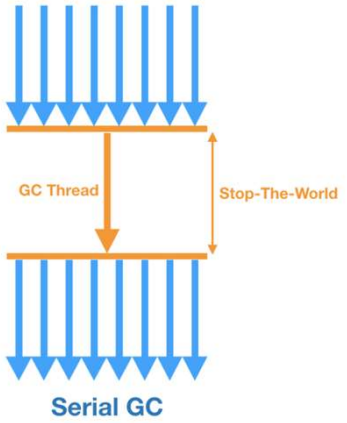
- 자바의 Garbage Collection을 수행하는 Garbage Collector에는 여러가지 종류가 있다. 여기서는 HotSpot JVM이 제공하는 Garbage Collector 종류와 간단한 옵션을 알아보도록 한다.

*HotSpot JVM에서는 Generation에 맞춰 원하는 Garbage Collector를 선택하여 적용할 수 있다. 예시로 아래표는 Java8에서 사용할 수 있는 Garbage Collector 종류와 그에 따른 옵션을 나타낸다. 각 Garbage Collector 설명은 다음으로 이어진다.

Young	Tenured	JVM options
Incremental	Incremental	-Xincgc
Serial	**Serial**	**-XX:+UseSerialGC**
Parallel Scavenge	Serial	-XX:+UseParallelGC -XX:-UseParallelOldGC
Parallel New	Serial	N/A
Serial	Parallel Old	N/A
Parallel Scavenge	**Parallel Old**	**-XX:+UseParallelGC -XX:+UseParallelOldGC**
Parallel New	Parallel Old	N/A
Serial	CMS	-XX:-UseParNewGC -XX:+UseConcMarkSweepGC
Parallel Scavenge	CMS	N/A
Parallel New	**CMS**	**-XX:+UseParNewGC -XX:+UseConcMarkSweepGC**
G1		**-XX:+UseG1GC**

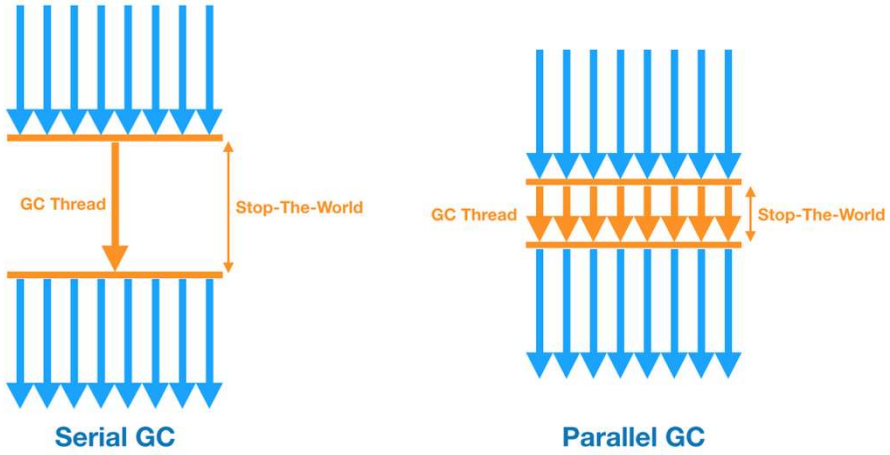
Garbage Collector

- **Serial Garbage Collector**
  - Java gc opt 명령어 : **-XX:+UseSerialGC**
  - 가장 원시적 방식의 GC, 싱글 스레드로 동작한다.
  - Young Generation : Generational Algorithm, Old Generation : Mark and Compaction Algorithm
  - 운영서버에서 사용 불가 - CPU 코어가 하나만 있을 때 사용되는 방식으로 성능이 좋지 않고, 지연시간이 길다.



Garbage Collector

- **Parallel Garbage Collector**
  - Java gc opt 명령어 : **-XX:+UseParallelGC**
  - Java 8의 기본 GC
  - Serial GC와 기본적 알고리즘은 동일. 하지만 여러 개의 스레드로 GC를 처리한다.
  - Through GC라고도 한다.



Garbage Collector

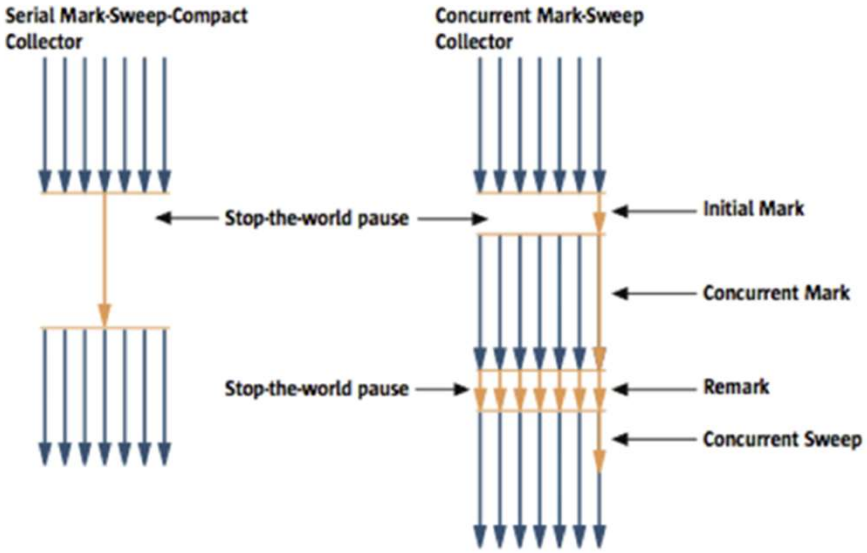
- **Parallel Old Garbage Collector**
  - Java gc opt 명령어 : **-XX:+UseParallelOldGC**
  - Parallel GC에서 Old 영역의 알고리즘만 다르다.
  - Mark-Summary-Compaction 단계를 거친다. (GC를 수행한 영역에 대해 별도로 살아있는 객체를 식별한다. Mark-Sweep-Compaction의 Sweep 단계와 다르고, 더 복잡하다.)

Garbage Collector

- **CMS (Concurrent Mark-and-Sweep) Garbage Collector**
  - java gc opt 명령어 : **-XX:+UseConcMarkSweepGC** (Java 버전 확인해서 사용)
  - **removed** in Java 14, **deprecated** on Java 9
  - 순서
    1. Initial Mark : 클래스로더에서 가장 가까운 객체 중 살아있는 객체만 찾는다.
    2. Concurrent Mark : 1에서 확인한 객체에서 참조하고 있는 객체들을 따라가며 확인한다. (스레드 동시 진행)
    3. Remark : 2에서 새로 추가되거나 참조가 끊긴 객체를 확인한다.
    4. Concurrent Sweep : Garbage Collection 실행 (스레드 동시 진행)
  - 장점 : STW(Stop the world) 시간이 매우 짧음.
  - 단점 : 다른 GC에 비해 CPU 사용량이 크고, Compaction 단계가 기본제공 되지 않는다.
  - 단점을 신경 써서 사용해야 한다. (단편화 발생 가능)

Garbage Collector

- **CMS Garbage Collector**



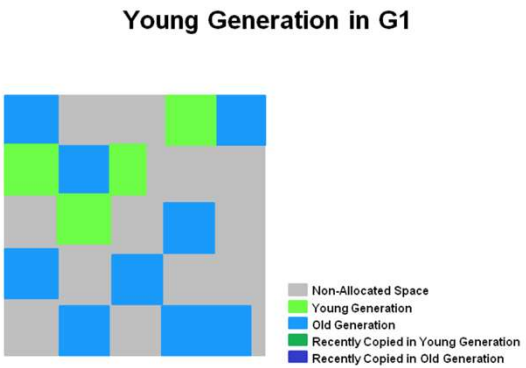
<Serial GC와의 비교>

Garbage Collector

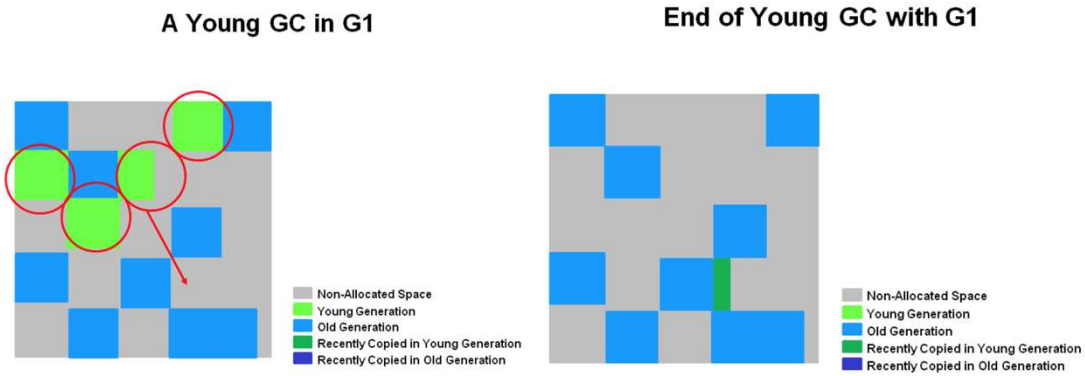
- **G1 Garbage Collector**
  - Java gc opt 명령어 : **-XX:+UseG1GC** (Java 버전 확인해서 사용)
  - Java 7부터 포함 되었고, *Java 9*부터 기본 GC로 채택
  - Garbage First Garbage Collector, Garbage region부터 먼저 회수하는 방식
  - 앞의 Young/Old Generation과는 다른 개념을 가진다.

Garbage Collector

- **G1 Garbage Collector**
  - Young Generation (Minor GC)



1. 연속되지 않은 공간에 Young Generation이 Region 단위로 할당.

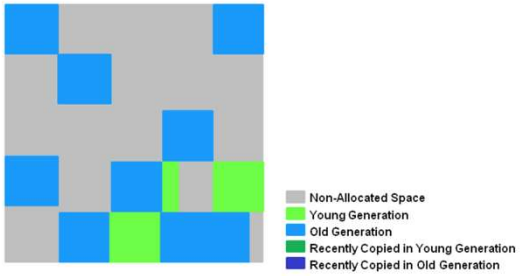


2. 살아있는 객체가 Survivor region이나 old generation으로 evacuation 된다.
→ 이 단계에서 STW가 발생, Eden과 Survivor의 크기는 다음 Minor GC를 위해 재계산

Garbage Collector

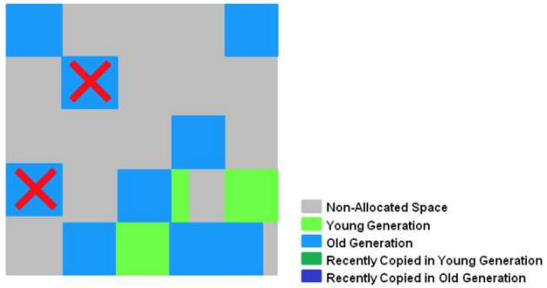
- **G1 Garbage Collector**
  - Old Generation (Major GC)

Initial Marking Phase



1. **Initial Mark** : Old Region에 존재하는 객체들이 참조하는 Survivor Region이 있는지 파악한 후 *Survivor Region*에 Mark
   → Survivor Region에 아무것도 없어야 한다; Minor GC가 모두 끝난 상태 → Minor GC에 의존적이며 STW를 발생시킴
2. **Root Region Scan** : 1에서 마킹 된 Survivor Region에서 Old Region에 대해 참조하고 있는 *객체를 마킹*, 멀티스레드로 동작하고 다음 Minor GC 전 동작을 완료한다.

Concurrent Marking Phase

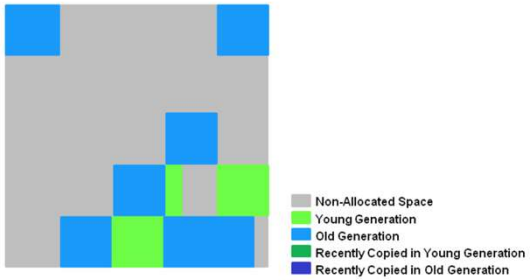


3. **Concurrent Marking** : Old Generation 내 생존해 있는 모든 객체를 마킹. STW는 발생하지 않지만 Minor GC와 같이 진행되므로 종종 인터럽트가 발생한다. 이미지에서 X표시가 된 Region은 모든 객체가 Garbage 상태인 영역.

Garbage Collector

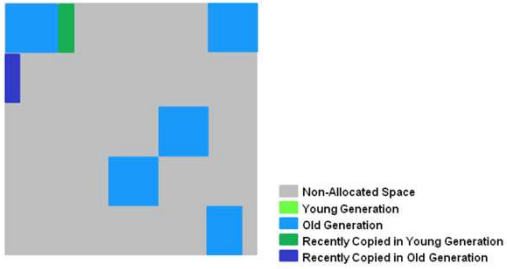
- **G1 Garbage Collector**
  - Old Generation (Major GC)

Remark Phase



Copying/Cleanup Phase

After Copying/Cleanup Phase



4. **Remark** : 3에서 X 표시한 영역을 바로 회수한다.(STW) 그리고 3에서 작업하던 Mark를 이어서 완전히 끝낸다. 이 때, SATB기법(Snapshot-At-The-Beginning : STW이후 살아있는 객체에만 마킹하는 알고리즘)을 사용하므로 GMS CG보다 빠르다.

5. Copying / Cleanup : STW 발생, 살아있는 객체의 비율이 낮은 영역 순으로 순차적으로 수거. 해당 영역에서 살아있는 객체를 다른 영역으로 evacuation한 후 Garbage를 수집한다.
→ G1GC는 이렇게 Garbage 수집을 우선(First)으로 해서 지속적으로 신속하게 여유공간을 확보한다.

* Major GC 이후 살아있는 객체가 새로운 Region으로 이동하고 메모리 Compaction이 일어나서 깔끔해진 모습 (이미지)

Garbage Collector

- **Z Garbage Collector**
  - java gc opt 명령어 : **-XX:+UseZGC** (Java 버전 확인해서 사용)
  - ***Experimental feature*** in Java 11, ***production feature*** in java 15
  - ZGC는 아래 목표를 충족하기 위해 설계된 확장 가능하고 낮은 지연율(low latency)을 가진 GC이다.
    - 정지 시간이 최대 10ms를 초과하지 않음
    - Heap의 크기가 증가하더라도 정지 시간이 증가하지 않음
    - 8MB~16TB에 이르는 다양한 범위의 Heap 처리 가능

Garbage Collection

- 주요 원리는 Load barrier와 Colored Pointer를 함께 사용하는 것이다. (ZGC에서 Compact는 새로운 region을 생성한 후 살아남은 객체를 채우면서 진행하는데 이 작업을 Thread 동작 중에도 시행 가능할 수 있게 함)
  - Load barrier : Concurrent GC를 사용하여 객체의 GC 메타데이터를 객체 주소에 저장
  - Colored Pointer : JIT를 사용해 GC를 돕기 위한 작은 코드를 주입
- ZGC는 메모리를 ZPages라고 불리는 영역으로 나눈다. ZPages는 동적 사이즈(G1 GC와 다름)로 2MB의 배수가 동적으로 생성 및 삭제될 수 있다. (크기별 힙영역 이미지 참고)