1장 +) GC

Garbage Collector (G.C)

> 옛 GC와 최근 GC와의 차이

GC 유튜브 - 테코톡

> **Mark and Sweep** 과정이라고 함

GC 과정

1. GC가 Stack의 모든 변수를 스캔하면서 각각 **어떤 객체를 참조하고 있는지** 찾아서 **마킹**
2. Reachable Object가 참조하고 있는 객체도 찾아서 마킹
3. **마킹되지 않은 객체**를 **Heap에서 제거**한다.

Heap

Young Gen (Eden, survival 0, 1) , Old Gen으로 나눠져있음

GC 과정

New Gen → Eden에 새로운 객체가 할당된다. → Eden이 다 차면 GC 발생(Minor GC)

→ Eden영역에서만 Mark & Sweep 발생 → 생존자만 Survival 0으로 이동 → 이 과정 반복

→ Survival 0 영역이 가득 차면 이 영역에 다시 Mark & Sweep 과정 → 생존자 Survival 1으로 이동후 Age 증가 → Eden에서 Mark & Sweep이 발생하는데 Survival 1으로 이동 → 가득

차면 Mark & Sweep 하고 Survival 0으로 이동 후 Age값 증가 → 위 과정을 반복할 때 특정 Age값을 넘으면 Old Gen으로 이동 → Old Gen이 가득 차면 GC 실행 (Major GC)

❓ 왜 Minor GC와 Major GC를 나누는가

GC설계에는 아래 두가지 가설을 두고 만들었다.

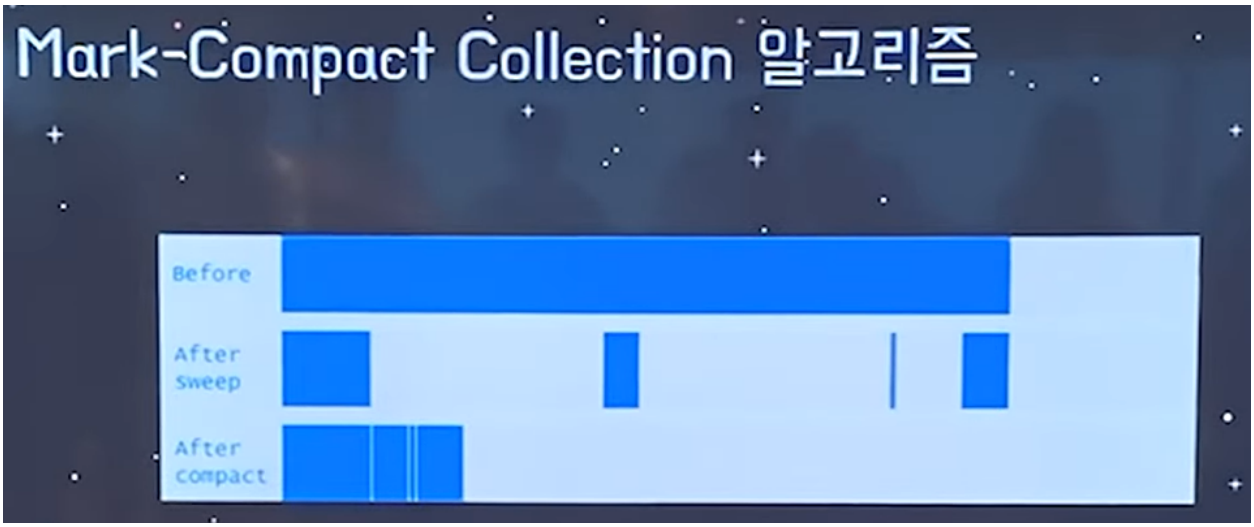
- 대부분 객체는 금방 접근 불가능한 상태(unreachable)가 된다. (= 금방 garbage가 된다)
- 오래된 객체에서 젊은 객체로의 참조는 아주 적게 존재한다.

GC 종류

Serial GC

- GC를 처리하는 스레드가 1개이다
- CPU 코어가 1개만 있을 때 사용
- Mark-Compact collection 알고리즘 사용

❓ Mark-Compact collection 알고리즘



▼ After sweep을 compact에 몰아버림

Parallel GC

- GC 처리하는 스레드가 여러개
- 속도가 빠름
- CPU 코어가 여러개일때 사용

Concurrent Mark Sweep GC (CMS GC)

- stop-the-world를 줄임으로 응답시간이 빨라진다
- 다른 GC보다 메모리와 CPU를 많이 사용한다
- Compaction 단계가 제공되지 않음

G1 GC

- 각 영역을 Region 영역으로 나눈다.
- GC가 일어날 때 전체영역을 탐색하지 않는다 → stop-the-world 시간을 줄인다
- Java 9+의 default GC

❓ Stop-The-World

- GC를 실행하기 위해 JVM이 애플리케이션 실행을 멈추는 것
- stop-the-world가 발생하면 GC를 실행하는 스레드를 제외한 나머지 스레드는 모두 작업을 멈춤
- GC 작업을 완료한 후 작업 재개

D2 GC를 읽고

GC에 관심을 가진다 ? → 규모가 일정 이상인 애플리케이션을 제작해 본 경험 有

GC는 언제 동작하나?

- OS로부터 할당 받은 시스템의 메모리가 부족한 경우
- 관리하고 있는 힙에서 사용되는 메모리가 허용된 임계값을 초과하는 경우
- 프로그래머가 직접 GC를 실행하는 경우 (System.gc()라는 메소드가 있지만 가급적 사용 x)

왜 Young, Old로 나눌까?

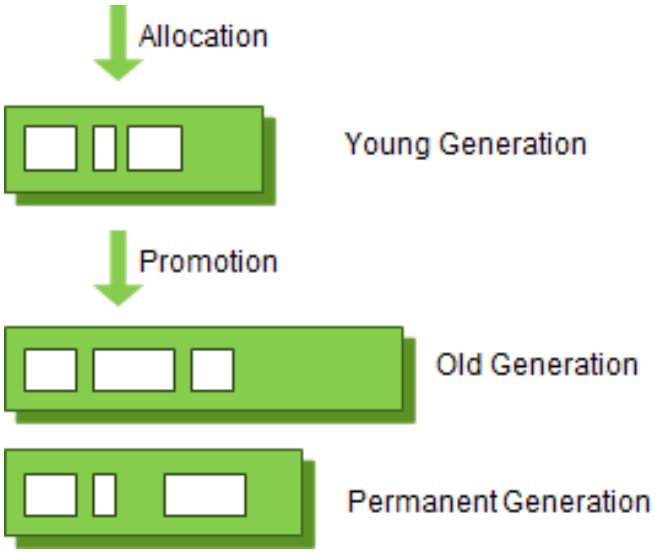
- 새로 할당된 객체일수록 금방 해제될 확률이 높다는 특성때문

Stop - the - world

- GC 튜닝은 이 stop-the-world시간을 줄이는 것을 말한다.
- System.gc() 메서드는 절대 사용하면 안됨 (시스템 성능에 매우 큰 영향)

HotSpot VM의 두개의 물리적 공간

- `Young 영역` **:** 새롭게 생선한 객체의 대부분이 여기에 위치. 대부분의 객체가 금방 접근 불가능한 상태가 되기 때문에 많은 객체가 Young 영역에 생성됐다가 사라진다. 이 영역에서 객체가 사라질 때 **Minor GC**가 발생했다고 한다.
- `Old 영역` : 접근 불가능 상태로 되지 않아 Young영역에서 살아남은 객체가 여기로 복사된다. 대부분 Young 영역보다 크게 할당하며, 크기가 큰 만큼 Young 영역보다 GC가 적게 발생한다. 이 영역에서 객체가 사라질 때 **Major GC(혹은 Full GC)** 가 발생했다고 한다.

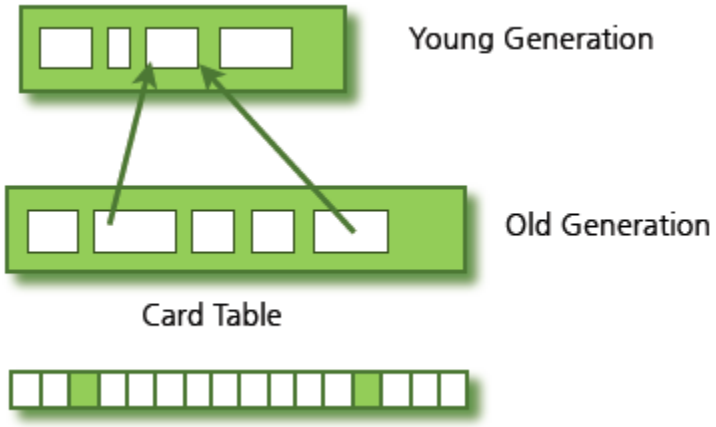


영역 및 데이터 흐름도 (출처 : D2)

위 그림의 **Permanent Generation** 영역 (이하 **Perm** 영역) 은 **Method Area**라고도 한다. 객체나 억류 (intern) 된 문자열 정보를 저장하는 곳이며, Old 영역에서 살아남은 객체가 영원히 남아 있는 곳은 절대 아니다. 이 영역에서 GC가 발생할 수도 있는데, 여기서 GC가 발생해도 Major GC의 횟수에 포함된다.

Old 영역의 객체가 **Young 영역의 객체를 참조**하는 경우 → 512바이트의 덩어리 (chunk)로 되어있는 **카드 테이블(card table)** 이 존재한다.

카드 테이블에는 Old 영역에 있는 객체가 **Young 영역의 객체**를 **참조할 때마다 정보가 표시**된다. Young 영역의 GC를 실행할 때는 Old 영역에 있는 모든 객체를 확인하는 대신 **카드 테이블만 뒤져서 GC 대상인지 식별**한다.



카드 테이블 구조

카드 테이블은 **write barrier**를 사용하여 관리한다. 이는 **Minor GC를 빠르게** 할 수 있도록 하는 장치이다. 약간의 오버헤드는 발생해도 **전반적인 GC 시간을 줄일** 수 있다.

Young 영역의 구성

> Young 영역은 3개의 영역으로 나뉜다.

- **Eden 영역**
- **Survivor 영역 (2개)**

각 영역의 처리 절차

- 새로 생성한 대부분의 객체는 Eden에 위치
- Eden에서 GC가 발생한 후 살아남은 객체는 Survivor 영역 중 하나로 이동
- Eden에서 GC가 발생하면 살아남은 객체가 존재하는 Survivor 영역으로 객체 계속 쌓임
- 하나의 Survivor 영역이 가득 차면 그중 살아남은 객체를 다른 Survivor으로 이동후 비우기
- 이 과정을 반복하다 계속 살아남은 객체는 Old 영역으로 이동

Survivor 영역 중 하나는 반드시 비어있는 상태. 두 Survivor 영역 모두 데이터가 존재 || 모두 0 이면

비정상적인 상황이다.

> ## HotSpot VM에서 빠른 메모리 할당을 위해 사용하는 두가지 기술

bump-the-pointer

- Eden 영역에 할당된 마지막 객체를 추적한다.
- 마지막 객체는 Eden의 맨 위(top)에 있다.
- 다음 객체가 생성되면, 해당 객체의 크기가 Eden 영역에 넣기 적당한지 확인후 맨위에 할당
- 새로운 객체를 생성할 때 마지막에 추가된 객체만 점검하면 되므로 **매우 빠른 메모리 할당 가능**

But, 멀티 스레드 환경을 고려하면 이야기가 달라짐!

Thread-Safe하기 위해 만약 여러 스레드에서 사용하는 객체를 Eden영역에 저장하려면 락(lock)이 발생할 수 밖에 없고, **lock-contention** 때문에 성능은 매우 떨어진다.

→ **해결방안 : TLABs**

TLABs(Thread-Local Allocation Buffers)

- 각각의 스레드가 각각의 몫에 해당하는 Eden 영역의 작은 덩어리를 가질 수 있도록 함

Old 영역에 대한 GC

> Old 영역은 기본적으로 데이터가 가득 차면 GC를 실행한다. JDK 7 기준 5개가 있다.

- Serial GC
- Parallel GC
- Paralel Old GC(Parallel Compacting GC)
- Concurrent Mark & Sweep GC(이하 CMS)
- G1 (Garbage First) GC

이중 운영 서버에 절대 사용하면 안되는 방식이 Serial GC이다.

→ CPU 코어가 하나만 있을 때 사용하는 것인데, 이를 사용하면 app의 성능이 매우 떨어진다.

Serial GC

- Young 영역에서의 GC는 앞 절에서 설명한 방식을 사용
- Old 영역 GC는 `mark-sweep-compact` 알고리즘 사용

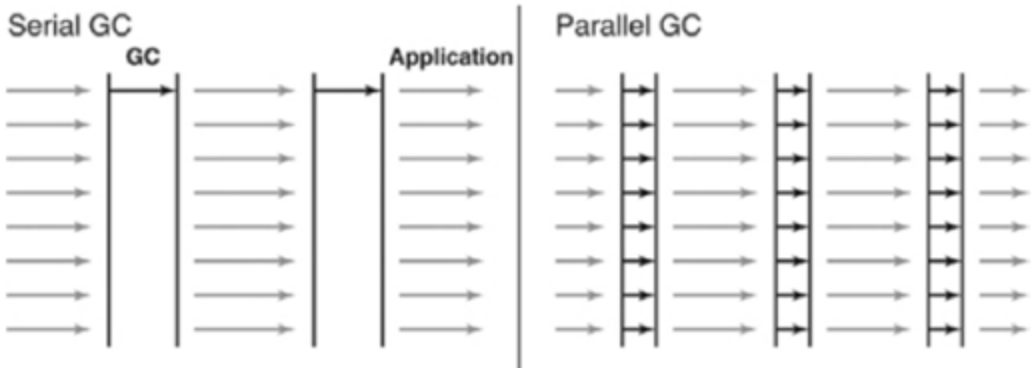
> ### Mark-weep-compact

Old 영역에 살아 있는 **객체를 식별(Mark)**, heap의 앞 부분부터 확인해 **살아있는 것만 남김 (Sweep)**

마지막에 각 객체들이 연속되게 쌓이도록 힙의 가장 앞 부분부터 채워서 **객체가 존재하는 부분과 없는 부분으로 나눔 (Compaction)**

Parallel GC (Throughput GC)

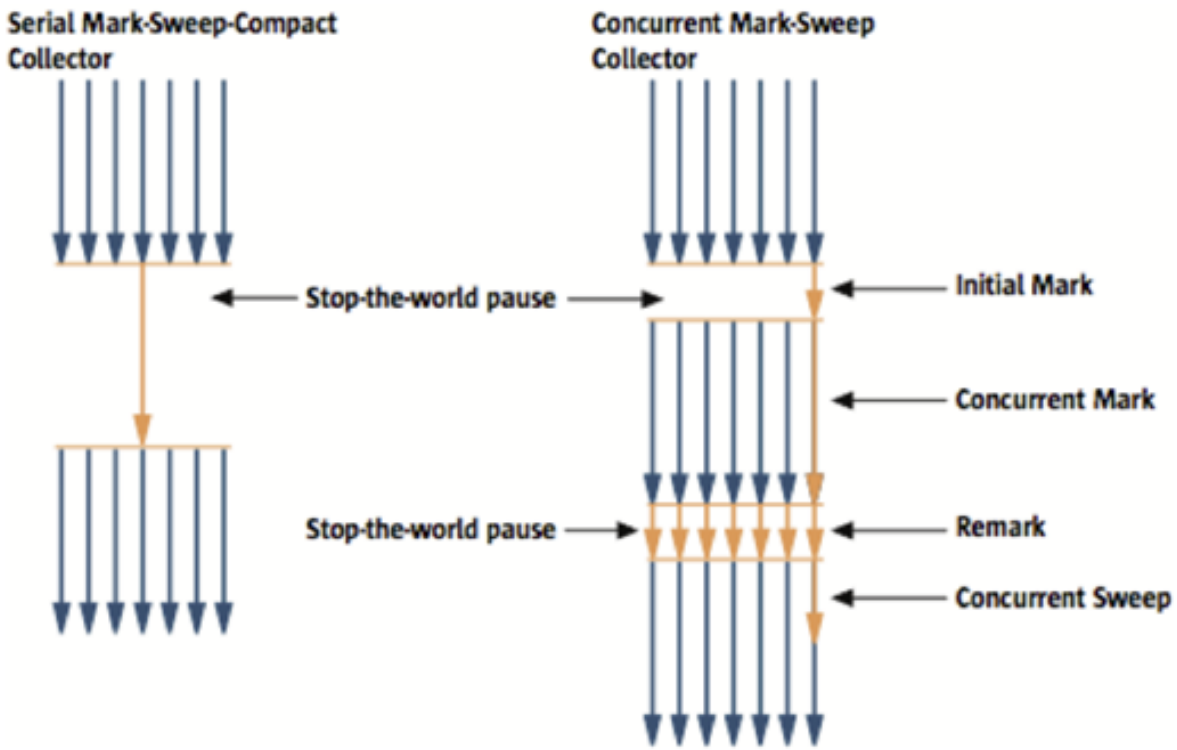
- Serial GC와 기본적인 알고리즘은 같지만 GC를 처리하는 쓰레드가 여러개이다.
- Serial GC보다 빠르게 객체를 처리 가능하다.



Parallel Old GC

- JDK 5 update 6부터 제공한 GC 방식이다.
- Old 영역의 GC 알고리즘만 다르다.
- `Mark-Summary-Compaction` 단계를 거친다.
- **Summary** 단계는 앞서 GC를 수행한 영역에 대해 **별도로 살아있는 객체를 식별한다**는 점에서 Sweep과 다르며, 약간 더 복잡한 단계를 거친다.

CMS GC (Low Latency GC)



그림에서 보듯, Serial GC보다 복잡하다.

Initial Mark 단계

- 클래스 로더에서 가장 가까운 객체 중 **살아있는 객체만 찾는다**.
- 멈추는 시간은 매우 짧다.

Concurrent Mark 단계

- 방금 살아있다고 확인한 객체에서 참조하고 있는 객체들을 따라가면서 확인
- **다른 스레드가 실행 중인 상태에서 동시에 진행**된다.

Remark 단계

- Concurrent Mark 단계에서 새로 추가되거나 참조가 끊긴 객체를 확인

Concurrent Sweep 단계

- 쓰레기를 정리하는 작업을 실행
- **이 작업도 다른 스레드가 실행중인 상태에서 동시에 진행**

장점

- stop-the-world 시간이 매우 짧다. 모든 애플리케이션의 응답속도가 매우 중요할 때 CMS GC를 사용.

단점

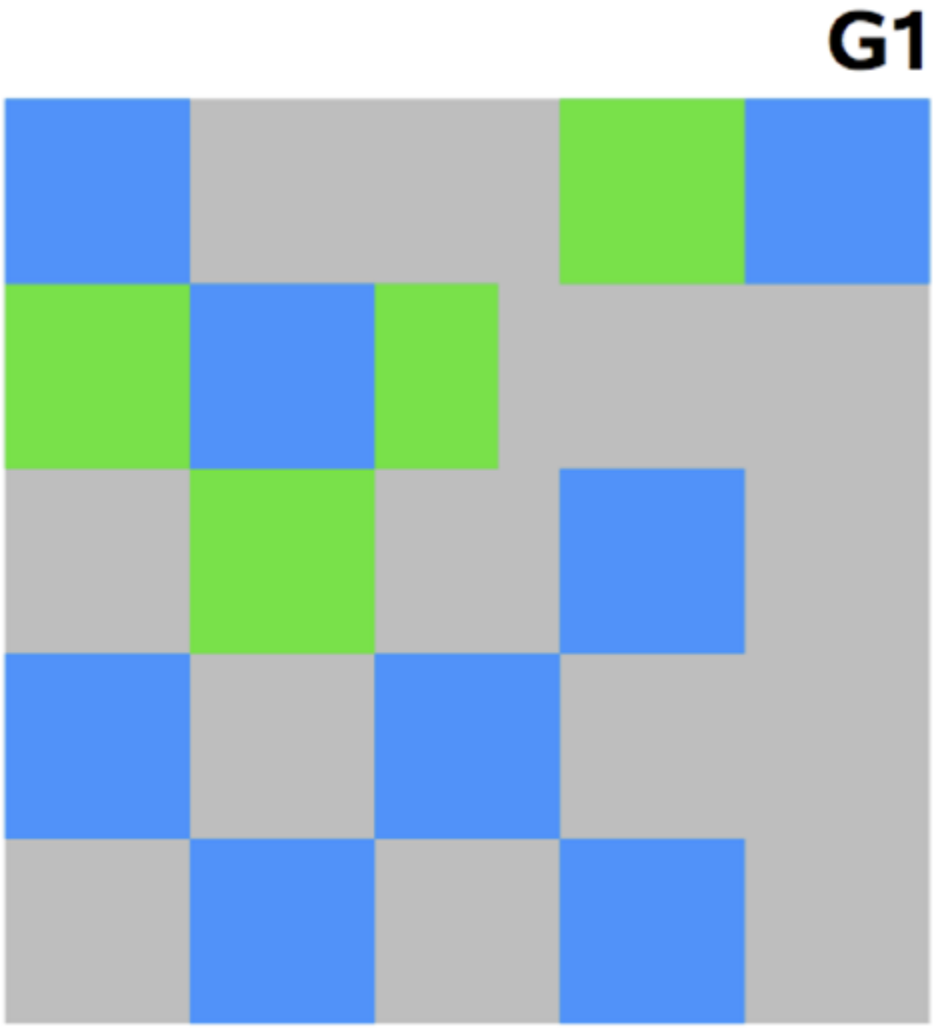
- 다른 GC방식보다 메모리와 CPU를 더 많이 사용한다.
- Compaction 단계가 기본적으로 제공되지 않는다.

따라서, CMS GC를 사용할 때에는 신중히 검토한 후에 사용해야 한다.

조각난 메모리가 많아 Compaction 작업을 실행하면 다른 GC 방식의 stop-the-world 시간보다 CMS의 **stop-the-world 시간이 더 길기** 때문에 Compaction 작업이 얼마나 자주, 오랫동안 수행되는지 확인해야 한다.

G1 GC

지금까지의 Young 영역과 Old 영역은 잊는것이 좋다.



위와 같이 바둑판의 각 영역에 객체를 할당하고 GC를 실행한다. 그러다가, 해당 영역이 꽉 차면 다른 영역에서 객체를 할당하고 GC를 실행한다. 즉, 지금까지 설명한 Young의 세가지 영역에서 데이터가 Old 영역으로 이동하는 단계가 사라진 GC 방식이라고 이해하면 된다. CMS GC를 대체하기 위해 만들어졌다.

지금까지 설명한 어떤 GC보다 빠르다.

하지만 JDK 6에서 G1 GC를 적용했다가 JVM Crash가 발생했다는 말이 있다.

→ 지금은 Java 9+에서 Default GC로 사용됨

전체 힙 메모리 영역을 Region 이라는 특정한 크기로 나눠 각 Region의 상태에 따라 역할이 동적으로 부여되는 상태이다. JVM 힙은 2048개의 Region으로 나뉠 수 있으며, 각 Region의 크

기는 1~32 MB로 지정될 수 있다.

G1 GC가 적용된 JVM Heap 구조

이 heap에서는 여태 못보던 두가지가 있는데,

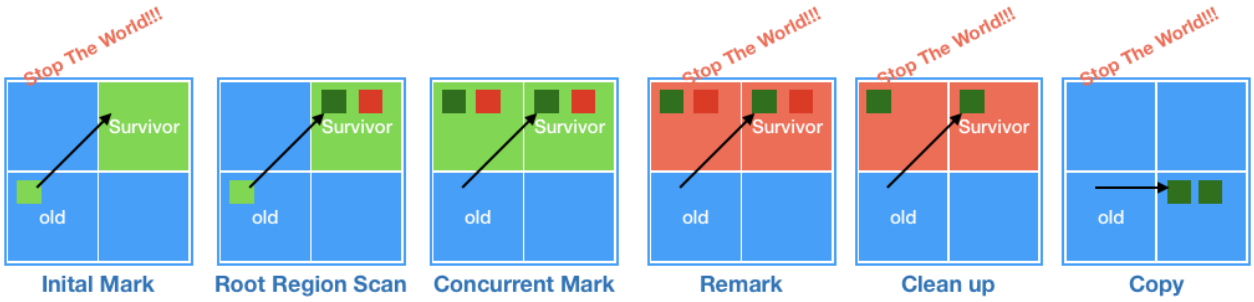
- `Humongous` : Region 크기의 50%를 초과하는 큰 객체를 저장하기 위한 공간이며, 이 Region에서는 GC 동작이 최적으로 동작하지 않는다.
- `Available/Unused` : 아직 사용되지 않은 Region을 의미한다.

G1 GC에서 Young GC를 수행할 때는 STW(stop-the-world) 현상이 발생하며, 이를 최소화하기 위해 멀티스레드로 수행된다. Young GC는 각 Region중 GC대상 객체가 가장 많은 Region에서 수행되며, 살아남은 객체를 다른 Region으로 옮긴 후 비워진 Region을 사용가능한 Region으로 돌리는 형태로 동작한다.

G1 GC에서 Full GC가 수행될 때는

> initial Mark → Root Region Scan → Concurrent Mark → Remark → Cleanup → Copy 단계를 거친다

- `Initial Mark` : **Old Region에 존재하는 객체들이 참조**하는 Survivor Region을 찾는다. 이 가정에서는 **STW현상**이 발생한다.
- `Root Region Scan` : 앞서 찾는 Survivor Region에 대한 **GC 대상 객체 스캔 작업**을 진행한다.
- `Concurrent Mark` : **전체 힙**의 Region에 대해 **스캔 작업**을 진행하며, GC 대상 객체가 **발견되지 않은 Region은 이후 단계를 처리하는데 제외**한다.
- `Remark` : **STW후** 최종적으로 **GC 대상에서 제외될 객체를 식별**해낸다.
- `Cleanup` : **STW후 살아있는 객체가 가장 적은 Region**에 대한 **미사용 객체 제거를** 수행한다. 이후 **STW를 끝내고**, 앞선 GC과정에서 **완전히 비워진 Region을 Freelist에 추가**하여 재사용될 수 있게 한다.
- `Copy` : GC대상 Region이었지만 Cleanup 과정에서 **완전히 비워지지 않은 Region의 살아남은 객체**들은 **새로운(Available/Unused) Region에 복사**하여 **Compaction** 작업을 수행한다.



특징

- STW에 의한 Pause Time을 예측할 수 있다 (Soft Real-time)
- Compaction을 동반하므로, Fragmentation을 줄일 수 있다.
- 세대별(Generational)로 객체를 관리하며 Parallel/Concurrent 수행방식은 기존의 Collector아 비슷하나, Young영역과 Old영역의 구분을 특정하지 않는 일정한 크기의 Region으로 관리됨

G1 GC에 대해

최근 Java의 GC

Java 11

> Epsilon, Z Garbage Collector(ZGC) 추가됨

Epsilon

- 메모리 할당은 처리하지만 **사용되지 않은 영역에 대해 재활용하지 않는다**. **Java Heap 영역을 모두 소진하게 되면 JVM이 Shut down** 된다. (기존엔 OS에 요청해 추가 Heap 영역 할당받음)
- **제한된 영역의 메모리 할당**을 허용함으로 최대한 **latency overhead를 줄인다**.
- Epsilon GC를 사용할 경우 우리가 작성한 어플리케이션이 **외부 환경으로부터 고립**된 채로 실행되기 때문에 실제 내 어플리케이션이 얼마나 **메모리를 사용하는지**에 대한 **임계치**나 **어플리케이션 퍼포먼스** 등을 **정확하게 측정**할 수 있다.

ZGC

- **대량의 메모리를 low-latency**로 잘 처리하기 위해 디자인된 GC

- Oracle에 따르면 **multi-tera byte** 크기의 heap도 관리할 수 있다고 한다.
- 어플리케이션과 **Concurrently하게 동작**하는데, Heap Reference를 위해 **Load barrier**을 사용함
- 이 Load barrier은 G1 GC보다 딜레이가 낮다.
- Java 12기준 64bit OS에서만 동작함 → 64bit 크기의 Color Point 방식으로 Heap 객체 관리하기 때문

Shenandoah GC

- ZGC와 비슷하게 메모리 처리에 우수한 퍼포먼스를 내지만 더 많은 옵션을 제공하는 장점
- 레드햇에서 개발함
- Java 8, 10버전에서도 호환 가능

옛 GC

최근 GC