JVM GC 과 메모리 Tuning

BEA Systems Korea Inc
Byungwook Cho

모든 Java Application 은 JVM(Java Virtual Machine)위에서 동작한다.
이 JVM 이 동작하는데 있어서, 메모리의 구조와 특히 GC 는 Application 의 응답시간과 성능에 밀접한 관계를 미친다. 이번 강좌에서는 JVM 의 메모리 구조와 GC 알고리즘 (JDK 1.4.X 에 포함된 새로운 알고리즘 포함) 그리고, JVM 의 메모리 튜닝을 통한 Application 의 성능향상방법에 대해서 알아보도록 하자.

목 차

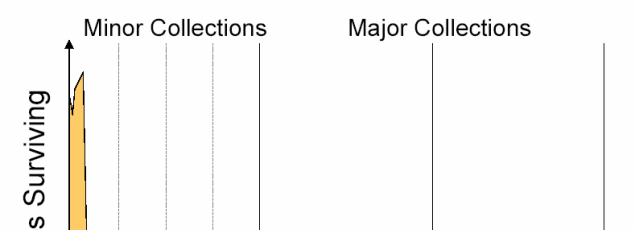
1.GC 란 무엇인가?

GC 는 Garbage Collection 의 약자로 Java 언어의 중요한 특징중의 하나이다.
GC 는 Java Application 에서 사용하지 않는 메모리를 자동으로 수거하는 기능을 말한다.
예전의 전통적인 언어 C 등의 경우 malloc, free 등을 이용해서 메모리를 할당하고, 일일이 그 메모리를 수거해줘야했다. 그러나 Java 언어에서는 GC 기술을 사용함에 따라서 개발자로 하여금 메모리 관리에서 부터 좀더 자유롭게 해주었다.

2.GC 의 동작 방법은 어떻게 되는가?

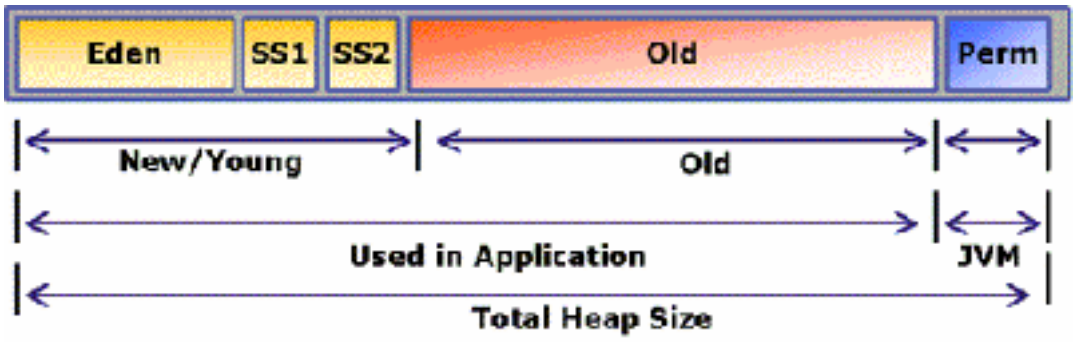
1) JVM 메모리 영역

GC 의 동작 방법을 이해하기 위해서는 Java 의 메모리 구조를 먼저 이해할 필요가 있다.
일반적으로 Application 에서 사용되는 객체는 오래 유지 되는 객체보다, 생성되고 잠시동안만 사용되는 경우가 많다. <그림 1 참조>



<그림 1. 메모리 foot print >

Java 에서는 크게 두가지 영역으로 메모리를 구분하는데 Young 영역과 Old 영역이 그것이다. Young 영역은 생성된지 얼마 안된 객체들을 저장하는 장소이고, Old 영역은 생성된지 오래된 객체를 저장하는 장소이다. 각 영역의 성격이 다른 만큼 GC 의 방법도 다르다.
먼저 Java 의 메모리 구조를 살펴보자



<그림 2. Java 메모리 구조 >

Java 의 메모리 영역은 앞에서 이야기한 두 영역 (Young 영역,Old 영역)과 Perm 영역 이렇게 3 가지로 영역으로 구성된다.

영역	설명
New/Young 영역	이 영역은 Java 객체가 생성되자마자 저장되고, 생긴지 얼마 안된 객체가 저장되는 곳이다. Java 객체가 생성되면 이 영역에서 저장되다가, 시간이 지남에 따라 우선순위가 낮아지면 Old 영역으로 옮겨진다.
Old 영역	New/Young 영역에서 저장되었던 객체중에 오래된 객체가 이동되어서 저장되는 영역
Perm 영역	Class,Method 등의 Code 등이 저장되는 영역으로 JVM 에 의해서 사용된다.

< 표 1. Java 메모리 영역 >

2) GC 알고리즘

그러면 이 메모리 영역을 JVM 이 어떻게 관리하는지에 대해서 알아보자
JVM 은 New/Young 영역과, Old 영역 이 두영역에 대해서만 GC 를 수행한다. Perm 영역은 앞에서 설명했듯이 Code 가 올라가는 부분이기 때문에, GC 가 일어날 필요가 없다.
Perm 영역은 Code 가 모두 Load 되고 나면 거의 일정한 수치를 유지한다.

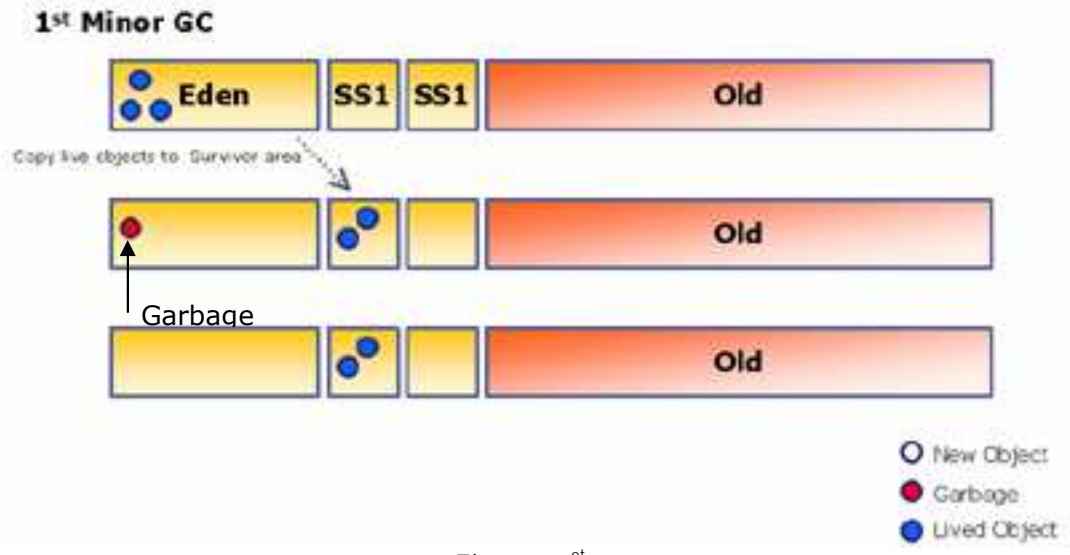
○ Minor GC
먼저 New/Young 영역의 GC 방법을 살펴보자 **New/Young 영역의 GC 를 Minor GC 라고 부르는데**, **New/Young 영역은 Eden 과 Survivor 라는 두가지 영역으로 또 나뉘어 진다**.
Eden 영역은 Java 객체가 생성되자 마자 저장이 되는곳이다. 이렇게 생성된 객체는 Minor GC 가 발생할때 Survivor 영역으로 이동된다.

Survivor 영역은 Survivor 1 과 Suvivor2 영역 두 영역으로 나뉘어 지는데, Minor GC 가 발생하면 Eden 과 Survivor1 에 Alive 되어 있는 객체를 Suvivor2 로 복사한다. 그리고 Alive 되어 있지 않는 객체는 자연히 Suvivor1 에 남아있게 되고, Survivor1 과 Eden 영역을 Clear 한다.
(결과적으로 Alive 된 객체만 Survivor2 로 이동한것이다.)
다음번 Minor GC 가 발생하면 같은 원리로 Eden 과 Survivor2 영역에서 Alive 되어 있는 객체를 Survivor1 에 복사한다. 계속 이런 방법을 반복적으로 수행하면서 Minor GC 를 수행한다.

이렇게 Minor GC 를 수행하다가, Survivor 영역에서 오래된 객체는 Old 영역으로 옮기게 된다.

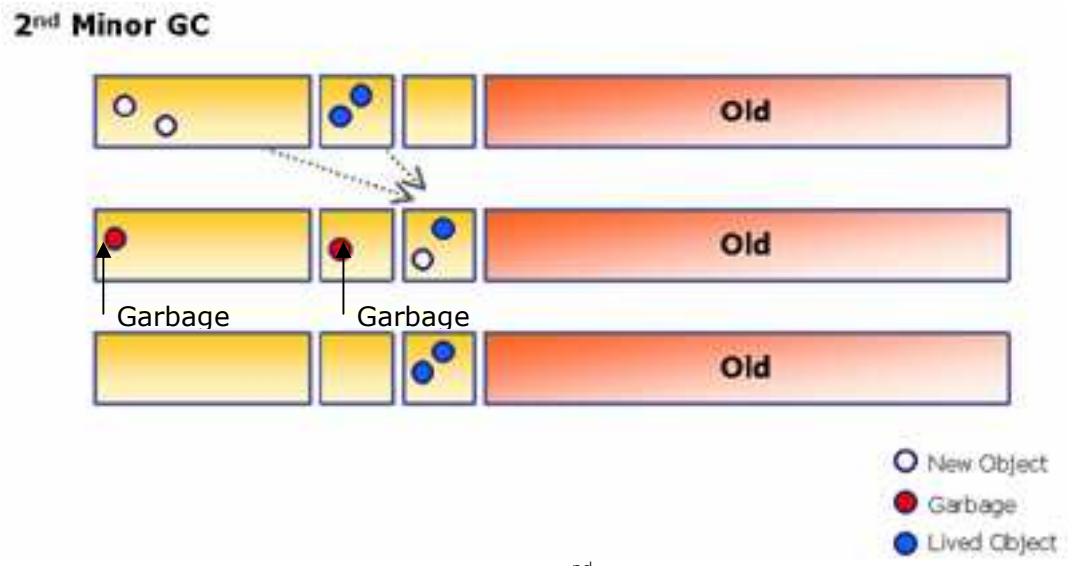
이런 방식의 GC 알고리즘을 **Copy & Scavenge** 라고 한다. 이 방법은 매우 속도가 빠르며 작은 크기의 메모리를 Collecting 하는데 매우 효과적이다. Minor GC 의 경우에는 자주 일어나기 때문에, GC 에 소요되는 시간이 짧은 알고리즘이 적합하다.

이 내용을 그림을 보면서 살펴보도록 하자



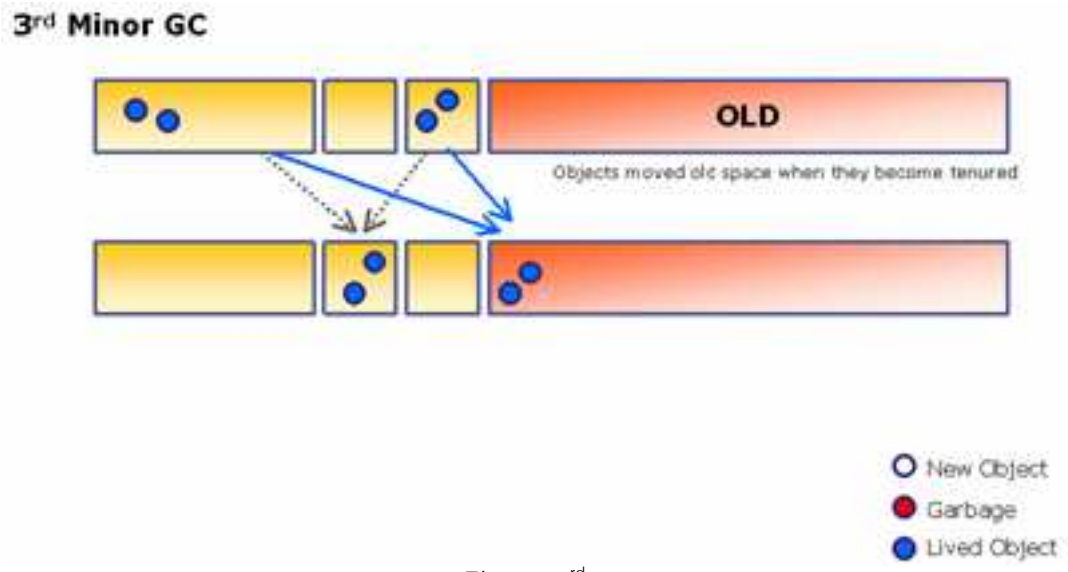
<그림 3-1. 1st Minor GC>

Eden 에서 Alive 된 객체를 Suvivor1 으로 이동한다. Eden 영역을 Clear 한다.



< 그림 3-2. 2nd Minor GC>

Eden 영역에 Alive 된 객체와 Suvivor1 영역에 Alive 된 객체를 Survivor 2 에 copy 한다 Eden 영역과 Suvivor2 영역을 clear 한다.



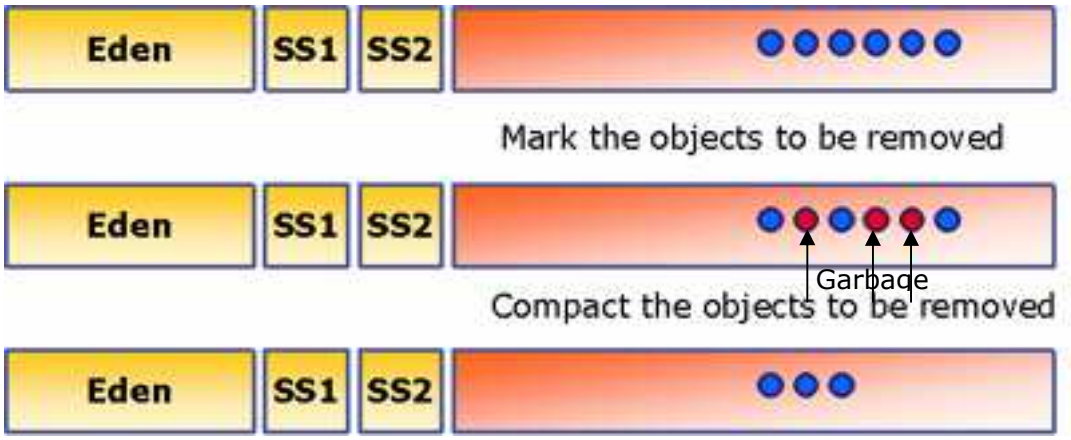
< 그림 3-3 3rd Minor GC>

객체가 생성된 시간이 오래지나면 Eden 과 Suvivor 영역에 있는 오래된 객체들을 Old 영역으로 이동한다.

○ Full GC

Old 영역의 Garbage Collection 을 Full GC 라고 부르며, Full GC 에 사용되는 알고리즘은 Mark & Compact 라는 알고리즘을 이용한다. Mark & Compact 알고리즘은 전체 객체들의 reference 를 체크하면서 reference 가 연결되지 않는 객체를 Mark 한다. 이 작업이 끝나면 사용되지 않는 객체를 모두 Mark 가 되고, 이 mark 된 객체를 삭제한다.<그림 4 참고> (실제로는 compact 라고 해서, mark 된 객체로 생기는 부분을 unmark 된 즉 사용하는 객체로 메꾸어 버리는 방법이다.)

Full GC 는 매우 속도가 느리며, Full GC 가 일어나는 도중에는 순간적으로 Java Application 이 멈춰 버리기 때문에, Full GC 가 일어나는 정도와 Full GC 에 소요되는 시간은 Application 의 성능과 안정성에 아주 큰 영향을 준다.



<그림 4. Full GC >

3. GC 가 왜 중요한가?

Garbage Collection 중에서 Minor GC 의 경우 보통 0.5 초 이내에 끝나기 때문에 큰 문제가 되지 않는다. 그러나 Full GC 의 경우 보통 수초가 소요가 되고, Full GC 동안에는 Java Application 이 멈춰버리기 때문에 문제가 될 수 있다.
예를 들어 게임 서버와 같은 Real Time Server 를 구현을 했을때, Full GC 가 일어나서 5 초동안 시스템이 멈춘다고 생각해보자.
또 일반 WAS 에서도 5~10 초 동안 멈추면, 멈추는 동안의 사용자의 Request 가 Queue 에 저장되었다가 Full GC 가 끝난후에 그 요청이 한꺼번에 들어오게 되면 과부하에 의한 여러 장애를 만들 수 있다..
그래서 원할한 서비스를 위해서는 GC 를 어떻게 일어나게 하느냐가 시스템의 안정성과 성능에 큰 변수로 작용할 수 있다.

4. 다양한 GC 알고리즘

앞에서 설명한 기본적인 GC 방법 (Scavenge 와 Mark and compact)이외에 JVM 에서는 좀더 다양한 GC 방법을 제공하고 그 동작방법이나 사용방법도 틀리다. 이번에는 다양한 GC 알고리즘에 대해서 알아보자. 현재 (JDK 1.4)까지 나와 있는 JVM 의 GC 방법은 크게 아래 4 가지를 지원하고 있다.

- Default Collector
- Parallel GC for young generation (from JDK 1.4)
- Concurrent GC for old generation (from JDK 1.4)

- Incremental GC (Train GC)

1) Default Collector

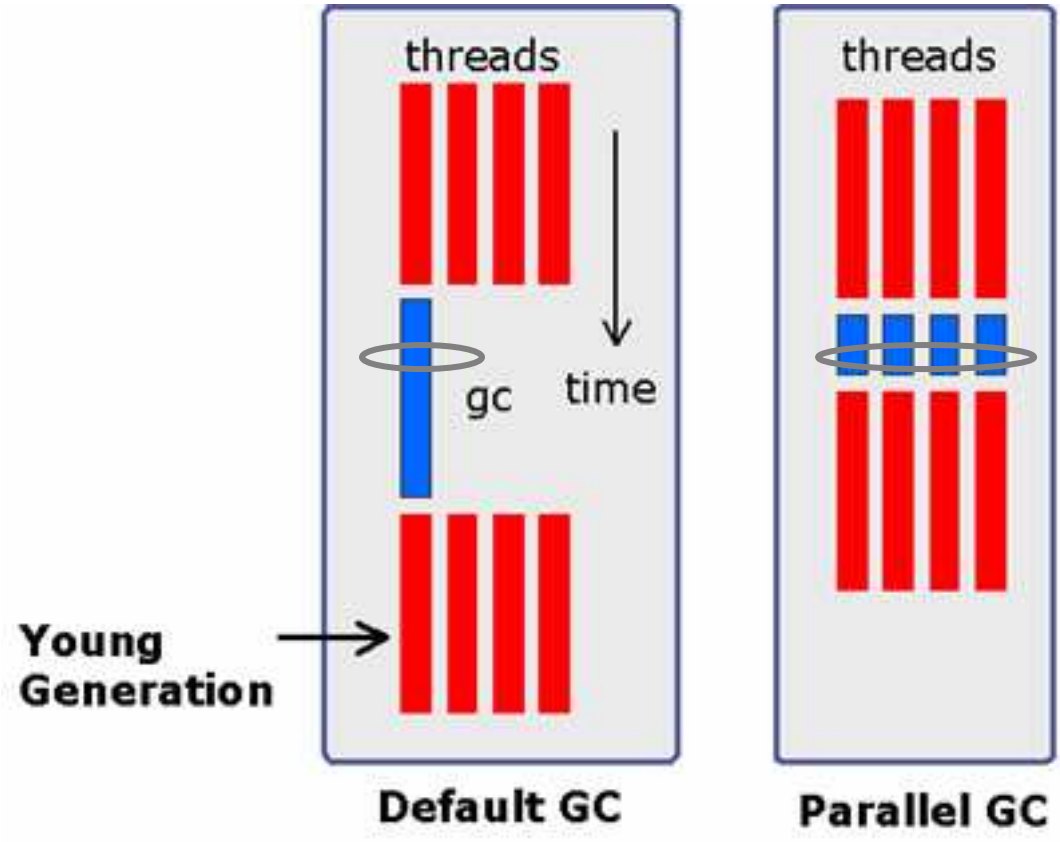
이 GC 방법은 앞에서 설명한 전통적인 GC 방법으로 Minor GC 에 Scavenge 를, Full GC 에 Mark & compact 알고리즘을 사용하는 방법이다. 이 알고리즘에는 이미 앞에서 설명했기 때문에 별도의 설명을 하지는 않는다.

JDK 1.4 에서부터 새로 적용되는 GC 방법은 Parallel GC 와 Concurrent GC 두가지 방법이 있다. Parallel GC 는 Minor GC 를 좀더 빨리하게 하는 방법이고 (Throughput 위주) Concurrent GC 는 Full GC 시에 시스템의 멈춤(Pause)현상을 최소화하는 GC 방법이다.

2) Parallel GC

JDK1.3 까지 GC 는 하나의 Thread 에서 이루어진다. Java 가 Multi Thread 환경을 지원함에도 불구하고, 1 CPU 에서는 동시에 하나의 Thread 만을 수행할 수 밖에 없기때문에, 예전에는 하나의 CPU 에서만 GC 를 수행했지만, 근래에 들어서 하나의 CPU 에서 동시에 여러개의 Thread 를 실행할 수 있는 Hyper Threading 기술이나, 여러개의 CPU 를 동시에 장착한 HW 의 보급으로 하나의 HW Box 에서 동시에 여러개의 Thread 를 수행할 수 있게 되었다.

JDK 1.4 부터 지원되는 Parallel GC 는 Minor GC 를 동시에 여러개의 Thread 를 이용해서 GC 를 수행하는 방법으로 하나의 Thread 를 이용하는것보다 훨씬 빨리 GC 를 수행할 수 있다.



<그림 5. Parallel GC 개념도>

<그림 5> 을 보자 왼쪽의 Default GC 방법은 GC 가 일어날때 Thread 들이 작업을 멈추고, GC 를 수행하는 thread 만 gc 를 수행한다. (그림에서 파란영역/동그라미로 표시된 부분),

Parallel GC 에서는 여러 thread 들이 gc 를 수행이 가능하기 때문에, gc 에 소요되는 시간이 낮아진다.

Parallel GC 가 언제나 유익한것은 아니다. 앞에서도 말했듯이 1CPU 에서는 동시에 여러개의 thread 를 실행할 수 없기 때문에 오히혀 Parallel GC 가 Default GC 에 비해서 느리다. 2 CPU 에서도 Multi thread 에 대한 지원이나 계산등을 위해서 CPU Power 가 사용되기 때문에, 최소한 4CPU 의 256M 정도의 메모리를 가지고 있는 HW 에서 Parallel GC 가 유용하게 사용된다.

Parallel GC 는 크게 두가지 종류의 옵션을 가지고 있는데,Low-pause 방식과 Throughput 방식의 GC 방식이 있다.

Solaris 기준에서 Low-pause Parallel GC 는 **-XX:+UseParNewGC** 옵션을 사용한다. 이 모델은 Old 영역을 GC 할때 다음에 설명할 Concurrent GC 방법과 함께 사용할 수 있다. 이 방법은 GC 가 일어날때 빨리 GC 하는것이 아니라 GC 가 발생할때 Application 이 멈춰지는 현상(pause)를 최소화하는데 역점을 뒀다.

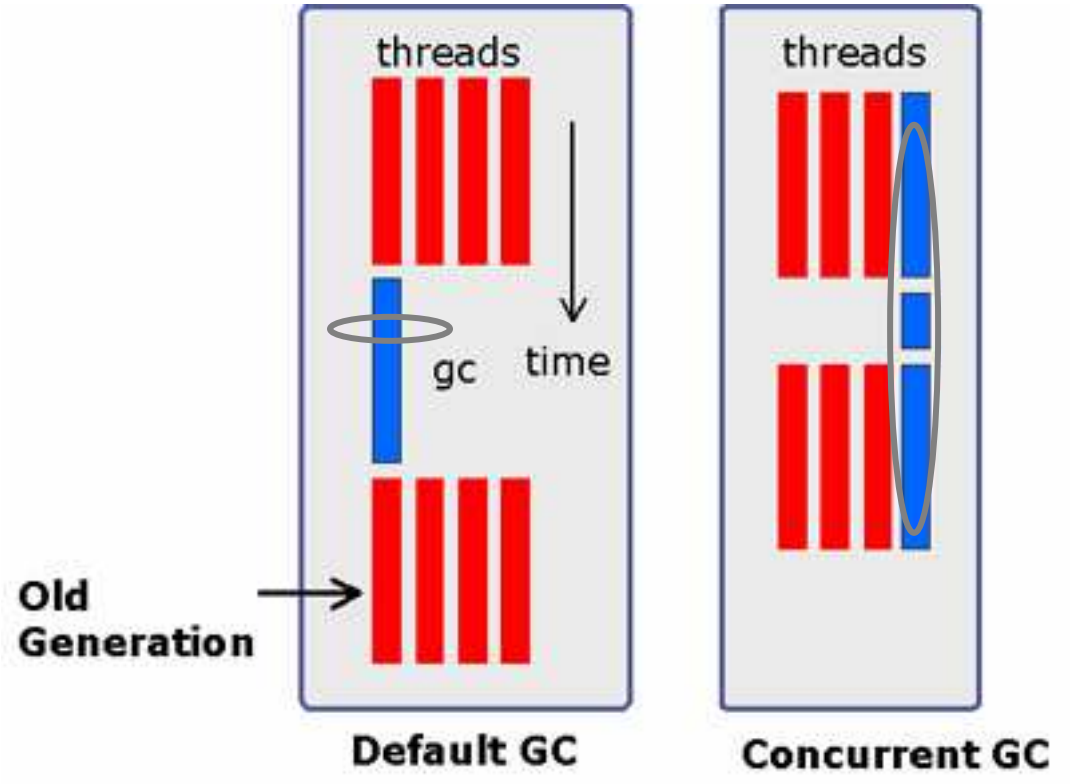
Throughput 방식의 Parallel GC 는 **-XX:+UseParallelGC** (Solaris 기준) 옵션을 이용하며 Old 영역을 GC 할때는 Default GC (Mark and compact)방법만을 사용하도록 되어 있다.Minor GC 가 발생했을때, 되도록이면 빨리 수행하도록 throughput 에 역점을 두었다.

그외에도 ParallelGC 를 수행할때 동시에 몇개의 Thread 를 이용하여 Minor 영역을 Parallel GC 할지를 결정할 수 있는데, -XX:ParallelGCThreads=<desired number> 옵션을 이용하여 Parallel GC 에 사용되는 Thread 의 수를 지정할 수 있다.

3) Concurrent GC

앞에서도 설명했듯이, Full GC 즉 Old 영역을 GC 하는 경우에는 그 시간이 길고 Application 이 순간적으로 멈춰버리기 때문에, 시스템 운용에 문제가 된다.

그래서 JDK1.4 부터 제공하는 Concurrent GC 는 기존의 이런 Full GC 의 단점을 보완하기 위해서 Full GC 에 의해서 Application 이 멈추어 지는 현상을 최소화 하기 위한 GC 방법이다. Full GC 에 소요되는 작업을 Application 을 멈추고 진행하는것이 아니라, 일부는 Application 이 돌아가는 단계에서 수행하고, 최소한의 작업만을 Application 이 멈췄을때 수행하는 방법으로 Application 이 멈추는 시간을 최소화한다.



<그림 6. Concurrent GC 개념도 >

그림 6 에서와 같이 Application 이 수행중일때(붉은 라인/동그라미로 표시된 부분) Full GC 를 위한 작업을 수행한다. (Sweep,mark) Application 을 멈추고 수행하는 작업은 일부분 (initial-mark, remark 작업)만을 수행하기 때문에, 기존 Default GC 의 Mark & Sweep Collector 에 비해서 Application 이 멈추는 시간이 현저하게 줄어든다.

Solaris JVM 에서는 -XX:+UseConcMarkSweepGC Parameter 를 이용해 세팅한다.

4) Incremental GC (Train GC)

Incremental GC 또는 Train GC 라고도 불리는 GC 방법은 JDK 1.3 에서부터 지원된 GC 방법이다. 앞에서 설명한 Concurrent GC 와 비슷하게, 의도 자체는 Full GC 에 의해서 Application 이 멈추는 시간을 줄이고자 하는데 있다.

Incremental GC 의 작동방법은 간단하다. Minor GC 가 일어날때 마다 Old 영역을 조금씩 GC 를 해서 Full GC 가 발생하는 횟수나 시간을 줄이는 방법이다.

<그림 7. Incremental GC 개념도 >

그림 7 에서 보듯이. 왼쪽의 Default GC 는 FullGC 가 일어난후에나 Old 영역이 Clear 된다. 그러나, 오른쪽의 Incremental GC 를 보면 Minor GC 가 일어난후에, Old 영역이 일부 Collect 된것을 볼 수 있다.

Incremental GC 를 사용하는 방법은 JVM 옵션에 -Xinc 옵션을 사용하면 된다.
Incremental GC 는 많은 자원을 소모하고, Minor GC 를 자주일으키고, 그리고 Incremental GC 를 사용한다고 Full GC 가 없어지거나 그 횟수가 획기적으로 줄어드는 것은 아니다. 오히려 느려지는 경우가 많다. 필히 테스트 후에 사용하도록 하자.

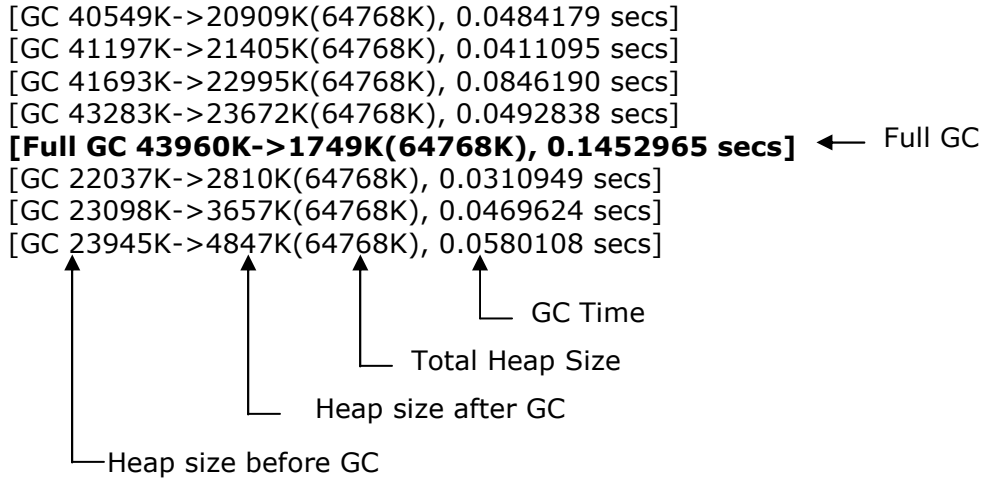
※ Default GC 이외의 알고리즘은 Application 의 형태나 HW Spec(CPU 수, Hyper threading 지원 여부), 그리고 JVM 버전(JDK 1.4.1 이냐 1.4.2 냐)에 따라서 차이가 매우 크다.
이론상으로는 실제로 성능이 좋아보일 수 있으나, 운영환경에서는 여러 요인으로 인해서 기대했던것만큼의 성능이 안나올 수 있기 때문에, 실환경에서 미리 충분한 테스트를 거쳐서 검증한후에 사용해야 한다.

5. GC 로그는 어떻게 수집과 분석

JVM 에서는 GC 상황에 대한 로그를 남기기 위해서 옵션을 제공하고 있다.
Java 옵션에 -verbosegc 라는 옵션을 주면되고 HP Unix 의 경우 -verbosegc -Xverbosegc 옵션을 주면 좀더 자세한 GC 정보를 얻을 수 있다. GC 정보는 stdout 으로 출력이 되기 때문에 “>” redirection 등을 이용해서 file 에 저장해놓고 분석할 수 있다.

Example) java -verbosegc MyApplication

그럼 실제로 나온 GC 로그를 어떻게 보는지를 알아보자



< 그림 8. 일반적인 GC 로그, Windows, Solaris >

<그림 8>는 GC 로그 결과를 모아놓은 내용이다. (실제로는 Application 의 stdout 으로 출력되는 내용과 함께 출력된다.)
Minor GC 는 "[GC "로 표기되고, Full GC 는 "[Full GC"로 표기된다.
그 다음값은 Heap size before GC 인데,GC 전에 Heap 사용량 (New/Young 영역 + Old 영역 + Perm 영역)의 크기를 나타낸다.

Heap size after GC 는 GC 가 발생한 후에 Heap 의 사용량이다. Minor GC 가 발생했을때는 Eden 과 Survivor 영역으 GC 가 됨으로 Heap size after GC 는 Old 영역의 용량과 유사하다.(Minor GC 에서 GC 되지 않은 하나의 Survivor 영역내의 Object 들의 크기도 포함해야한다.)

Total Heap Size 는 현재 JVM 이 사용하는 Heap Memory 양이다. 이 크기는 Java 에서 -ms 와 -mx 옵션으로 조정이 가능한데. 예를 들어 -ms512m -mx1024m 로 해놓으면 Java Heap 은 메모리 사용량에 따라서 512~1024m 사이의 크기에서 적절하게 늘었다 줄었다한다. (이 늘어나는 기준과 줄어드는 기준은 (-XX:MaxHeapFreeRatio 와 -XX:MinHeapFreeRation 를 이용해서 조정할 수 있으나 JVM vendor 에 따라서 차이가 나기때문에 각 vendor 별 JVM 메뉴얼을 참고하기 바란다.) Parameter 에 대한 이야기는 추후에 좀더 자세히하도록 하자

그 다음값은 GC 에 소요된 시간이다.

그림 8 의 GC 로그를 보면 Minor GC 가 일어날때마다 약 20,000K 정도의 Collection 이 일어난다. Minor GC 는 Eden 과 Suvivor 영역 하나를 GC 하는 것이기 때문에 New/Young 영역을 20,000Kbyte 정도로 생각할 수 있다.

Full GC 때를 보면 약 44,000Kbyte 에서 1,749Kbyte 로 GC 가 되었음을 볼 수 있다.
Old 영역에 큰 데이타가 많지 않은 경우이다. Data 를 많이 사용하는 Application 의 경우 전체 Heap 이 512 이라고 가정할때, Full GC 후에도 480M 정도로 유지되는 경우가 있다. 이런 경우에는 실제로 Application 에서 Memory 를 많이 사용하고 있다고 판단할 수 있기 때문에 전체 Heap Size 를 늘려줄 필요가 있다.

이렇게 수집된 GC 로그는 다소 보기가 어렵기 때문에, 좀더 쉽게 분석할 수 있게 하기 위해서 GC 로그를 awk 스크립트를 이용해서 정제하면 분석이 용이하다.

```
BEGIN{
  printf("Minor\tMajor\tAlive\tFree\n");
}
{
  if( substr($0,1,4) == "[GC "){
    split($0,array," ");
    printf("%s\t0.0\t",array[3])
    split(array[2],barray,"K")
    before=barray[1]
    after=substr(barray[2],3)
    reclaim=before-after
    printf("%s\t%s\n",after,reclaim)
  }
  if( substr($0,1,9) == "[Full GC "){
    split($0,array," ");
    printf("0.0\t%s\t",array[4])
    split(array[3],barray,"K")
    before = barray[1]
    after = substr(barray[2],3)
    reclaim = before - after
    printf("%s\t%s\n",after,reclaim)
  }
  next;
}
```

<표 2. gc.awk 스크립트 >

이 스크립트를 작성한후에 Unix 의 awk 명령을 이용해서

% awk –f gc.awk GC 로그파일명

을 쳐주면 아래<표 3>와 같이 정리된 형태로 GC 로그만 추출하여 보여준다.

Minor	Major	Alive	Freed
0.0484179	0.0	20909	19640
0.0411095	0.0	21405	19792
0.0846190	0.0	22995	18698
0.0492838	0.0	23672	19611
0.0	0.1452965	1749	42211
0.0310949	0.0	2810	19227
0.0469624	0.0	3657	19441
0.0580108	0.0	4847	19098

<표 3. gc.awk 스크립트에 의해서 정제된 로그 >

Minor 와 Major 는 각각 Minor GC 와 Full GC 가 일어날때 소요된 시간을 나타내며, Alive 는 GC 후에 남아있는 메모리양, 그리고 Freed 는 GC 에 의해서 collect 된 메모리 양이다.

이 로그파일은 excel 등을 이용하여 그래프등으로 변환해서 보면 좀더 다각적인 분석이 가능해진다.

※ JDK 1.4 에서부터는 -XX:+PrintGCDetails 옵션이 추가되어서 좀더 자세한 GC 정보를 수집할 수 있다.

※ HP JVM 의 GC Log 수집

HP JVM 은 전체 heap 뿐 아니라 -Xverbosegc 옵션을 통해서 Perm,Eden,Old 등의 모든 영역에 대한 GC 정보를 좀더 정확하게 수집할 수 있다.

Example) java -verbosegc -Xverbosegc MyApplication ← (HP JVM Only)

HP JVM 의 GC 정보는 18 개의 필드를 제공하는데 그 내용을 정리해보면 <표 4.>와 같다.

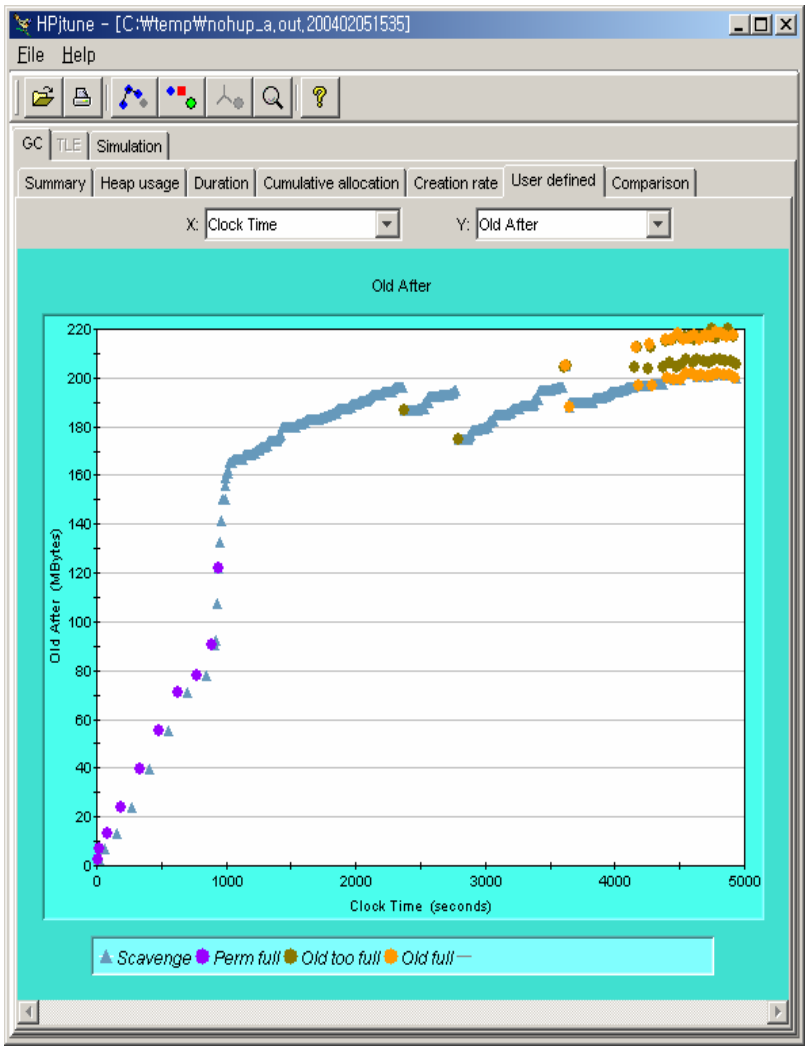
<GC : %1 %2 %3 %4 %5 %6 %7 %8 %9 %10 %11 %12 %13 %14 %15 %16 %17 %18 >

필드	의미
%1	GC 가 일어난 이유를 나타냄 -1:Minor GC 0~6 : Full GC (0:Call to System.gc, 1:Old Generation Full,2: Perm Generation Full,3:Train Generation Full,4:Old generation expanded on last scavenge,5:Old generation tool full to scavenge,6:FullGCAlot)
%2	GC 가 시작된 시간 (Application 이 시작된 시간을 기준으로 sec)
%3	GC invocation. Counts of Scavenge and Full GCs are maintained separately
%4	Size of the object allocation request that forced the GC in bytes
%5	Tenuring threshold – determines how long the new born object remains the New Generation
%06|%07|%08	Eden Before | After | Capacity
%09|%10|%11	Survivor Generation Before | After | Capacity
%12|%13|%14	Old Generation Before | After | Capacity
%15|%16|%17	Perm Generation Before | After | Capacity
%18	Time taken in seconds to finish the gc

<표 4. HP JVM GC 로그 필드별 의미 >

이 로그를 직접 보면서 분석하기는 쉽지가 않다. 그래서, HP 에서는 좀더 Visual 한 환경에서 분석이 가능하도록 HPJtune 이라는 툴을 제공한다. 다음 URL 에서 다운로드 받을 수 있다.

http://www.hp.com/products1/unix/java/java2/hpjtune/index.html



<그림 9. HP Jtune 을 이용해서 GC 후 Old 영역의 변화 추이를 모니터링하는 화면 >

6. GC 관련 Parameter

GC 관련 설정값을 보기전에 앞서서 -X 와 -XX 옵션에 대해서 먼저 언급하자. 이 옵션들은 표준 옵션이 아니라, 벤더별 JVM 에서 따로 제공하는 옵션이기 때문에, 예고 없이 변경되거나 없어질 수 있기 때문에, 사용전에 미리 JVM 벤더 홈페이지를 통해서 검증한다음에 사용해야한다.

1) 전체 Heap Size 조정 옵션

전체 Heap size 는 -ms 와 -mx 로 Heap 사이즈의 영역을 조정할 수 있다. 예를 들어 -ms512m -mx 1024m 로 설정하면 JVM 은 전체 Heap size 를 application 의 상황에 따라서 512m~1024m byte 사이에서 사용하게 된다. 그림 2 의 Total heap size

메모리가 부족할때는 heap 을 늘리고, 남을때는 heap 을 줄이는 heap growing 과 shirinking 작업을 수행하는데, 메모리 변화량이 큰 애플리케이션이 아니라면 이 min heap size 와 max heap size 는 동일하게 설정하는 것이 좋다. 일반적으로 1GB 까지의 Heap 을 설정하는데에는 문제가 없으나, 1GB 가 넘는 대용량 메모리를 설정하고자 할 경우에는 별도의 JVM 옵션이 필요한 경우가 있기때문에 미리 자료를 참고할 필요가 있다.

※ IBM AIX JVM 의 경우
%export LDR_CNTRL=MAXDATA=0x10000000
%java -Xms1500m -Xmx1500m MyApplication

2) Perm size 조정 옵션

Perm Size 는 앞에서도 설명했듯이, Java Application 자체(Java class etc..)가 로딩되는 영역이다. J2EE application 의 경우에는 application 자체의 크기가 큰 편에 속하기 때문에, Default 로 설정된 Perm Size 로는 application class 가 loading 되기에 부족한 경우가 대부분이기 때문에, WAS start 초기나, 가동 초기에 Out Of Memory 에러를 유발하는 경우가 많다.

PermSize 는 -XX:MaxPermSize=128m 식으로 지정할 수 있다.
일반적으로 WAS 에서 PermSize 는 64~256m 사이가 적절하다.

3) New 영역과 Old 영역의 조정

New 영역은 -XX:NewRatio=2 에 의해서 조정이 된다.
NewRatio Old/New Size 의 값이다. 전체 Heap Size 가 768 일때, NewRatio=2 이면
New 영역이 256m, Old 영역이 512m 로 설정이 된다.
JVM 1.4.X 에서는 -XX:NewSize=128m 옵션을 이용해서 직접 New 영역의 크기를 지정하는 것이 가능하다.

4) Survivor 영역 조정 옵션

-XX:SurvivorRatio=64 (eden/survivor 의 비율) :64 이면 eden 이 128m 일때, survivor 영역은 2m 가 된다.

5) -server 와 -client 옵션

JVM 에는 일반적으로 server 와 client 두가지 옵션을 제공한다.
결론만 말하면 server 옵션은 WAS 와 같은 Server 환경에 최적화된 옵션이고, client 옵션은 워드프로세서와 같은 client application 에 최적화된 옵션이다. 그냥 언뜻 보기에는 단순한 옵션 하나로보일 수 있지만, 내부에서 돌아가는 hotspot compiler 에 대한 최적화 방법과 메모리 구조자체가 아예 틀리다.

○ -server 옵션

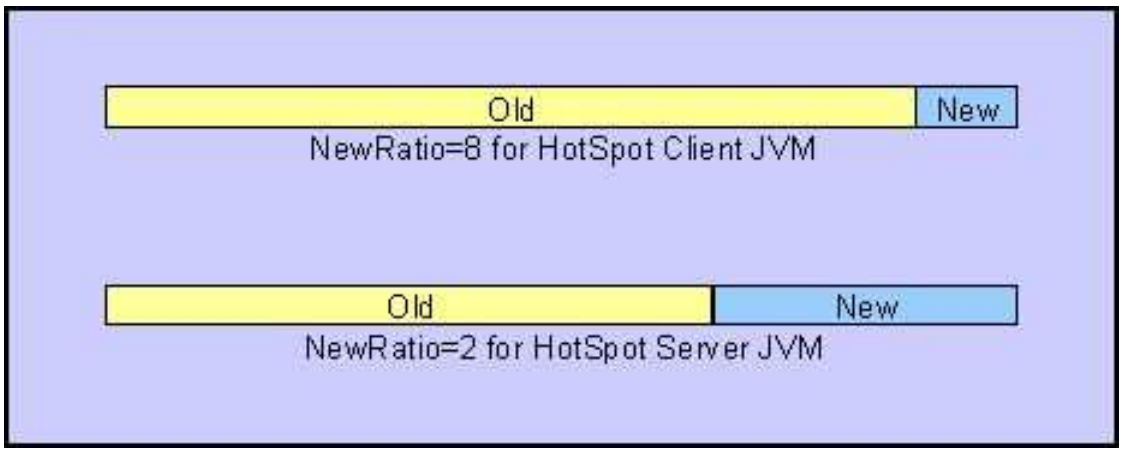
server 용 application 에 최적화된 옵션이다. Server application 은 boot up 시간 보다는 user 에 대한 response time 이 중요하고, 많은 사용자가 동시에 사용하기 때문에 session 등의

user data 를 다루는게 일반적이다. 그래서 server 옵션으로 제공되는 hotspot compiler 는 java application 을 최적화 해서 빠른 response time 을 내는데 집중되어 있다.

또한 메모리 모델 역시, 서버의 경우에는 특정 사용자가 서버 운영시간동안 계속 서버를 사용하는게 아니기 때문에 (Login 하고, 사용한 후에는 Logout 되기 때문에..) 사용자에 관련된 객체들이 오래 지속되는 경우가 드물다. 그래서 상대적으로 Old 영역이 작고 New 영역이 크게 배정된다. <그림 10. 참조 >

○ -client 옵션

client application 은 워드프로세서 처럼 혼자 사용하는 application 이다. 그래서 client application 은 response time 보다는 빨리 기동되는데에 최적화가 되어 있다. 또한 대부분의 client application 을 구성하는 object 는 GUI Component 와 같이 application 이 종료될때까지 남아있는 object 의 비중이 높기 때문에 상대적으로 Old 영역의 비율이 높다.



< 그림 10. -server 와 -client 옵션에 따른 JVM Old 와 New 영역 >

이 두 옵션은 가장 간단한 옵션이지만, JVM 의 최적화에 아주 큰부분을 차지하고 있는 옵션이기 때문에, 반드시 Application 의 성격에 맞춰서 적용하기 바란다.
(※ 참고로, SUN JVM 은 default 가 client, HPJVM 는 default 가 server 로 세팅되어 있다.)

○ GC 방식에 대한 옵션

GC 방식에 대한 옵션은 앞에서도 설명했지만, 일반적인 GC 방식이외에, Concurrent GC,Parallel GC,Inceremental GC 와 같이 추가적인 GC Algorithm 이 존재한다. 옵션과 내용은 앞장에서 설명한 “다양한 GC 알고리즘” 을 참고하기 바란다.

7.JVM GC 튜닝

그러면 이제부터 지금까지 설명한 내용을 기반으로 실제로 JVM 튜닝을 어떻게 하는지 알아보도록 하자.

STEP 1. Application 의 종류와 튜닝목표값을 결정한다.

JVM 튜닝을 하기위해서 가장 중요한것은 JVM 튜닝의 목표를 설정하는것이다. 메모리를 적게 쓰는것이 목표인지, GC 횟수를 줄이는것이 목표인지, GC 에 소요되는시간이 목표인지, Application 의 성능(Throughput or response time) 향상인지를 먼저 정의한후에. 그 목표치에 근접하도록 JVM Parameter 를 조정하는것이 필요하다.

STEP 2. Heap size 와 Perm size 를 설정한다.

-ms 와 -mx 옵션을 이용해서 Heap Size 를 정한다. 일반적으로 server application 인 경우에는 ms 와 mx 사이즈를 같게 하는것이 Memory 의 growing 과 shrinking 에 의한 불필요한 로드를 막을 수 있어서 권장할만하다.

ms 와 mx 사이즈를 다르게 하는 경우는 Application 의 시간대별 memory 사용량이 급격하게 변화가 있는 Application 에 효과적이다.
PermSize 는 JVM vendor 에 따라 다소 차이가 있으나 일반적으로 16m 정도이다. Client application 의 경우에는 문제가 없을 수 있지만, J2EE Server Application 의 경우 64~128m 사이로 사용이 된다.

Heap Size 와 Perm Size 는 아래 과정을 통해서 적정 수치를 얻어가야한다.

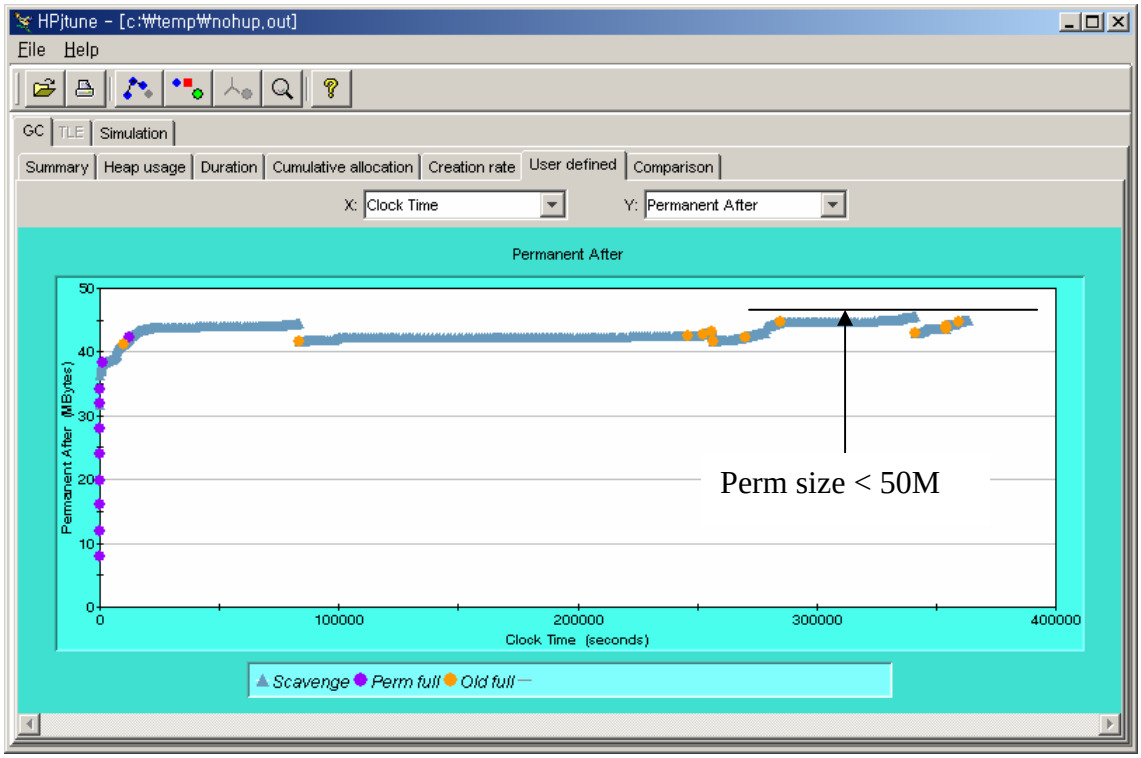
STEP 3. 테스트 & 로그 분석.

JVM Option 에 GC 로그를 수집하기 위한 -verbosegc 옵션을 적용한다. (HP 의 경우 -Xverbosegc 옵션을 적용한다.)

LoadRunner 나 MS Stress(무료로 MS 社의 홈페이지에서 다운로드 받을 수 있다.)와 같은 Stress Test 툴을 통해서 Application 에 Stress 를 줘서. 그 log 를 수집한다. 튜닝에서 있어서 가장 중요한것은 목표산정이지만, 그만큼이나 중요한것은 실제 Tuning 한 Parameter 가 Application 에 어떤 영향을 주는지를 테스트하는 방법이 매우 중요하다. 그런 의미에서 적절한 Stress Tool 의 선정과, Stress Test 시나리오는 정확한 Tuning 을 위해서 매우 중요한 요인이다.

○ Perm size 조정

아래 그림 11.은 HP JVM 에서 -Xverbosegc 옵션으로 수집한 GC log 를 HP Jtune 을 통해서 graph 로 나타낸 그래프이다. 그림을 보면 Application 이 startup 되었을때 Perm 영역이 40m 에서. 시간이 지난후에도 50m 이하로 유지되는것을 볼 수 있다. 특별하게 동적 classloading 등이 수십 M byte 가 일어나지 않는등의 큰 변화요인이 없을때, 이 application 의 적정 Perm 영역은 64m 로 판단할 수 있다.



<그림 11. GC 결과중 Perm 영역 그래프 >

○ GC Time 수행 시간 분석

다음은 GC 에 걸린 시간을 분석해보자. 앞에 강좌 내용에서도 설명햇듯이. GC Tuning 에서 중요한 부분중 하나가 GC 에 소요되는 시간 특히 Full GC 시간이다.

지금부터 볼 Log 는 모社의 물류 시스템의 WAS 시스템 GC Log 이다. HP JVM 을 사용하며, -server -ms512m -mx512m 옵션으로 기동되는 시스템이다.

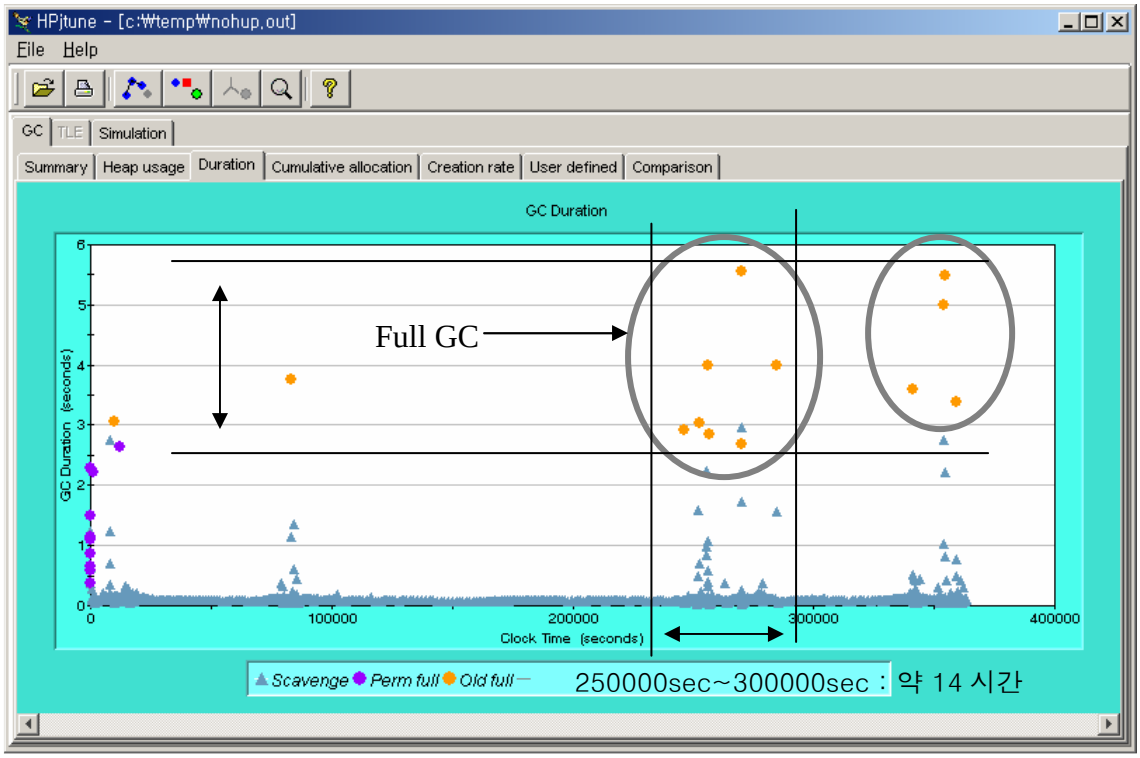
그림 9 를 보면 Peak 시간 (첫번째 동그라미) 14 시간동안에 Full GC(동그란점)가 7 번일어난것을 볼 수 있다. 각각에 걸린 시간은 2.5~6sec 사이이다.
여기서 STEP 1.에서 설정한 AP Tuning 의 목표치를 참고해야하는데.

Full GC 가 길게 일어나서 Full GC 에 수행되는 시간을 줄이고자 한다면 Old 영역을 줄이면 Full GC 가 일어나는 횟수는 늘어나고, 반대로 Full GC 가 일어나는 시간을 줄어들것이다.

반대로 Full GC 가 일어나는 횟수가 많다면, Old 영역을 늘려주면 Full GC 가 일어나는 횟수는 상대적으로 줄어들것이고 반대로 Full GC 수행시간이 늘어날 것이다.

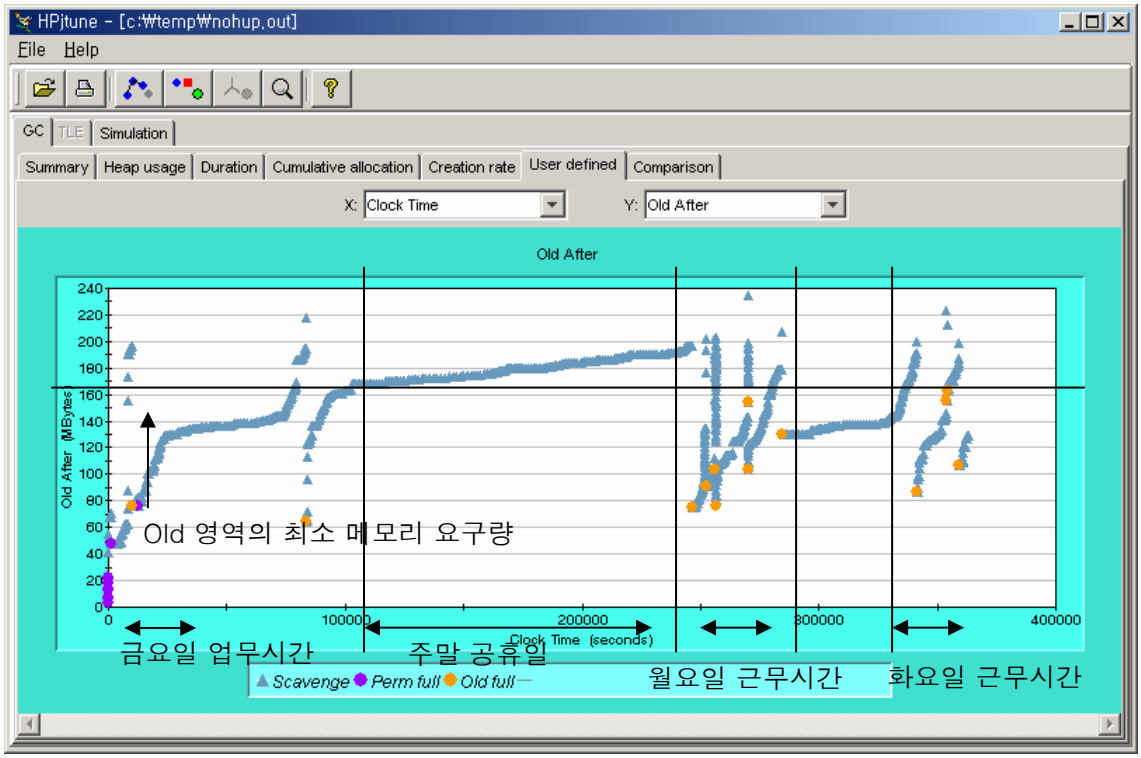
특히 Server Application 의 경우 Full GC 가 일어날때는 JVM 자체가 멈춰버리기 때문에, 그림 12 의 instance 는 14 시간동안 총 7 번 시스템이 멈추고, 그때마다 2.5~6sec 가량 시스템이 response 를 못하는 상태가 된것이다. 그래서 멈춘 시간이 고객이 납득할만한 시간인지를 판단해야 하고, 거기에 적절한 Tuning 을 해야한다.

Server Application 에서 Full GC 를 적게일어나게하고, Full GC 시간을 양쪽다 줄이기 위해서는 Old 영역을 적게한후에, 여러개의 Instance 를 동시에 뛰어서 Load Balancing 을 해주면, Load 가 분산되기 때문에 Full GC 가 일어나는 횟수가 줄어들테고, Old 영역을 줄였기 때문에, Full GC 에 드는 시간도 줄어들것이다. 또한 각각의 FullGC 가 일어나는동안 하나의 서버 instance 가 멈춰져 있어도, Load Balancing 이 되는 다른 서버가 response 를 하고 있기때문에, Full GC 로 인한 Application 이 멈추는것에 의한 영향을 최소화할 수 있다.



<그림 12. GC 소요시간 >

데이타에 따라서 GC Tuning 을 진행한후에는 다시 Stress Test 를 진행해서 응답시간과 TPS(Throughput Per Second)를 체크해서 어떤 변화를 주었는지를 반드시 체크해봐야한다.



<그림 13. GC 후의 Old 영역>

그림 13 은 GC 후에 Old 영역의 메모리 변화량을 나타낸다

금요일 업무시간에 메모리 사용량이 올라가다가. 주말에가서 완만한 곡선을 그리는것을 볼 수 있다. 월요일 근무시간에 메모리 사용량이 매우 많고, 화요일에도 어느정도 메모리 사용량이 있는것을 볼 수 있다. 월요일에 메모리 사용량이 많은것을 볼때, 이 시스템의 사용자들이 월요일에 시스템 사용량이 많을 수 있다고 생각할 수 있고, 또는 다른 주의 로그를 분석해봤을때 이 주만 월요일 사용량이 많았다면, 특별한 요인이나 Application 변경등이 있었는지를 고려해봐야할것이다.

이 그래프만을 봤을때 Full GC 가 일어난후에도 월요일 근무시간을 보면 Old 영역이 180M 를 유지하고 있는것을 볼 수 있다. 이 시스템의 Full GC 후의 Old 영역은 80M~180M 를 유지하는것을 볼 수 있다. 그래서 이 시스템은 최소 180M 이상의 Old 영역을 필요로하는것으로 판단할 수 있다.

STEP 4. Parameter 변경.

STEP 3 에서 구한 각 영역의 허용 범위를 기준으로 Old 영역과 New 영역을 적절하게 조절한다. PermSize 와 New 영역의 배분 (Eden,Survivor)영역등을 조정한다.
PermSize 는 대부분 Log 에서 명확하게 나타나기 때문에, 크게 조정이 필요가 없고 New 영역내의 Eden 과 Survivor 는 거의 조정하지 않는다. 가장 중요한것은 Old 영역과 New 영역의 비율을 어떻게 조정하는가가 관건이다.

이 비율을 결정하면서, STEP1 에서 세운 튜닝 목표에 따라서 JVM 의 GC Algorithm 을 적용한다. GC Algorithm 을 결정하는 기본적인 판단 내용은 아래와 같다.

향상 포인트	GC Algorithm	비고

Performance(속도) 중시	Parallel GC	JVM 1.4.2 이상 4CPU 이상
Responsiveness(응답성) 중시	Concurrent GC	JVM 1.4.2 이상 4CPU 이상
Responsiveness(응답성) 중시	Incremental GC	JVM 1.4.2 이하 또는 4CPU 미만
일반	Default GC	JVM 1.4.2 이하 또는 4CPU 미만

이렇게 Parameter 를 변경하면서 테스트를 진행하고, 다시 변경하고 테스트를 진행하는 과정을 거쳐서 최적의 Parameter 와 GC Algorithm 을 찾아내는것이 JVM 의 메모리 튜닝의 이상적인 절차이다.

지금까지 JVM 의 메모리 구조와 GC 모델 그리고 GC 튜닝에 대해서 알아보았다.
정리하자면 GC 튜닝은 Application 의 구조나 성격 그리고, 사용자의 이용 Pattern 에 따라서 크게 좌우 되기때문에, 얼마만큼의 Parameter 를 많이 아느냐 보다는 얼마만큼의 테스트와 로그를 통해서 목표 값에 접근하느냐가 가장 중요하다.

참고자료

Sun 社 JVM Hotspot Options http://java.sun.com/docs/hotspot/VMOptions.html
Tuning Garbage Collection with the Java 1.4.2 http://java.sun.com/docs/hotspot/gc1.4.2/
Java Performance Tuning 2nd ed – Oreilly