G1 가비지 컬렉터 시작하기

Overview

개요

목적

이 튜토리얼에서는 G1 가비지 수집기를 사용하는 방법과 핫스팟 JVM과 함께 사용하는 방법에 대한 기본 사항을 다룹니다. G1 수집기가 내부적으로 작동하는 방식, G1을 사용하기 주요 명령줄 스위치, 작동 로깅 옵션 등을 배우게 됩니다.

완료 시간 약 1시간 소개

이 OBE에서는 Java 가상 머신(JVM) G1 가비지 컬렉션(GC)의 기본 사항을 다룹니다. 첫 번째 파트에서는
OBE에서는 가비지 컬렉션 및 성능에 대한 소개와 함께 JVM에 대한 개요가 제공됩니다. 그 다음에는 CMS 수집기가 핫스팟 JVM에서 어떻게 작동하는지에 대한 검토가 제공됩니다. 다음으로, 핫스팟 JVM과 함께 G1 가비지 컬렉션을 사용할 때 가비지 컬렉션이 어떻게 작동하는지에 대한 단계별 가이드가 제공됩니다. 그 다음에는 G1 가비지 수집기에서 사용할 수 있는 가비지 수집 명령줄 옵션을 다루는 섹션이 제공됩니다. 마지막으로 G1 수집기와 함께 사용할 수 있는 로깅 옵션에 대해 알아봅니다.

하드웨어 및 소프트웨어 요구 사항

다음은 하드웨어 및 소프트웨어 요구 사항 목록입니다:

- Windows XP 이상, Mac OS X 또는 Linux를 실행하는 PC. 실습은 Windows 7에서 수행되었으며 모든 플랫폼에서 테스트되지는 않았습니다. 그러나 OS X 또는 Linux에서는 모든 것이 정상적으로 작동합니다. 또한 코어가 두 개 이상인 컴퓨터가 좋습니다.
- Java 7 업데이트 9 이상
- 최신 Java 7 데모 및 샘플 Zip 파일

전제 조건

이 튜토리얼을 시작하기 전에 다음을 수행해야 합니다:

- 아직 설치하지 않았다면 최신 버전의 Java JDK(JDK 7 u9 이상)를 다운로드하여 설치하세요. Java 7 JDK 다운로드
- 같은 위치에서 데모 및 샘플 zip 파일을 다운로드하여 설치합니다. 파일의 풀고 내용을 디렉토리에 넣습니다. 예를 들어 `C:\javademos`

Java 기술 및 JVM

Java Technology and the JVM

Java 개요

Java는 1995년 썬마이크로시스템즈에서 처음 출시한 프로그래밍 언어이자 컴퓨팅 플랫폼입니다. 유틸리티, 게임, 비즈니스 애플리케이션을 비롯한 Java 프로그램의 기반이 되는 기술입니다. Java는 전 세계 8억 5천만 대 이상의 개인용 컴퓨터와 모바일 및 TV 기기를 포함한 전 세계 수십억 대의 기기에서 실행되고 있습니다. Java는 전체적으로 Java 플랫폼을 구성하는 여러 가지 주요 구성 요소로 이루어져 있습니다.

Java 런타임 에디션

Java를 다운로드하면 Java 런타임 환경(JRE)이 제공됩니다. JRE는 Java 가상 머신(JVM), Java 플랫폼 핵심 클래스 및 지원 Java 플랫폼 라이브러리로 구성됩니다. 컴퓨터에서 Java 애플리케이션을 실행하려면 이 세 모두 필요합니다. Java 7에서 Java 애플리케이션은 운영 체제에서 데스크톱 애플리케이션으로 실행되거나, 데스크톱 애플리케이션이지만 Java Web Start를 사용하여 웹에서 설치되거나, 브라우저에서 웹 임베디드 애플리케이션(JavaFX 사용)으로 실행됩니다.

Java 프로그래밍 언어

Java는 다음과 같은 기능을 포함하는 객체 지향 프로그래밍 언어입니다.

플랫폼 독립성 - Java 애플리케이션은 클래스 파일에 저장된 *바이트코드로* 컴파일되어 JVM에서 로드됩니다. 애플리케이션은 JVM에서 실행되므로 다양한 운영 체제 및 기기에서 실행할 수 있습니다.

- 객체 지향 - Java는 C와 C++의 많은 기능을 가져와 이를 개선한 객체 지향 언어입니다.

- 자동 가비지 컬렉션 - Java는 자동으로 메모리를 할당하고 할당 해제하여 프로그램이 해당 작업에 부담을 느끼지 않도록 합니다.
- 풍부한 표준 라이브러리 - Java에는 입력/출력, 네트워킹, 날짜 조작과 같은 작업을 수행하는 데 사용할 수 있는 수많은 사전 제작된 객체가 포함되어 있습니다.

Java 개발 키트

JDK(Java 개발 키트)는 Java 애플리케이션을 개발하기 위한 도구 모음입니다. JDK를 사용하면 Java 프로그래밍 언어로 작성된 프로그램을 컴파일하고 JVM에서 실행할 수 있습니다. 또한 JDK는 애플리케이션을 패키징하고 배포하기 위한 도구도 제공합니다.

JDK와 JRE는 Java 애플리케이션 프로그래밍 인터페이스(Java API)를 공유합니다. Java API는 개발자가 Java 애플리케이션을 만드는 데 사용하는 사전 패키지 라이브러리 모음입니다. Java API는 문자열 조작, 날짜/시간 처리, 네트워킹, 데이터 구조(예: 목록, 맵, 스택, 큐) 구현 등 많은 일반적인 프로그래밍 작업을 완료할 수 있는 도구를 제공하여 개발을 더 쉽게 만들어 줍니다.

Java 가상 머신

자바 가상 머신(JVM)은 추상적인 컴퓨팅 머신입니다. JVM은 그 안에서 실행되도록 작성된 프로그램에게 기계처럼 보이는 프로그램입니다. 이러한 방식으로 Java 프로그램은 동일한 인터페이스 및 라이브러리 집합으로 작성됩니다. 특정 운영 체제에 대한 각 JVM 구현은 Java 프로그래밍 명령어를 로컬 운영 체제에서 실행되는 명령어와 명령으로 변환합니다. 이러한 방식으로 Java 프로그램은 플랫폼 독립성을 확보할 수 있습니다.

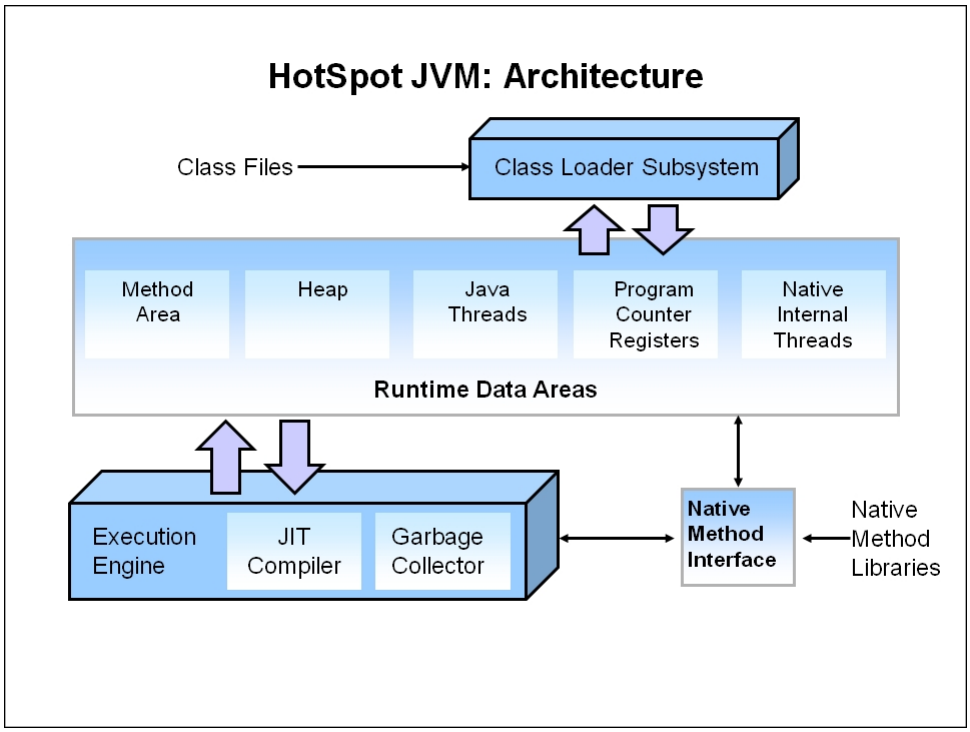
Java 가상 머신의 첫 번째 프로토타입 구현은 Sun Microsystems, Inc.에서 수행되었으며, 최신 개인용 디지털 비서(PDA)와 유사한 핸드헬드 장치에서 호스팅되는 소프트웨어에서 Java 가상 머신 명령어 집합을 에뮬레이트했습니다. 오라클의 현재 구현은 모바일, 데스크탑 및 서버 기기에서 Java 가상 머신을 에뮬레이트하지만 Java 가상 머신은 특정 구현 기술, 호스트 하드웨어 또는 호스트 운영 체제를 가정하지 않습니다. 본질적으로 해석되는 것이 아니라 명령어 집합을 실리콘 CPU의 명령어 집합으로 컴파일하여 구현할 수 있습니다. 또한 마이크로코드로 구현하거나 실리콘에 직접 구현할 수도 있습니다.

Java 가상 머신은 Java 프로그래밍 언어에 대해서는 전혀 알지 못하며 특정 바이너리 형식인 클래스 파일 형식만 알고 있습니다. 클래스 파일에는 Java 가상 머신 명령어(또는 바이트코드와 심볼 테이블 및 기타 보조 정보가 포함되어 있습니다.

보안을 위해 Java 가상 머신은 클래스 파일의 코드에 강력한 구문 및 구조적 제약을 부과합니다. 그러나 유효한 클래스 파일로 표현할 수 있는 기능을 가진 모든 언어는 Java 가상 머신에서 호스팅할 수 있습니다. 일반적으로 사용 가능한 기계 독립적인 플랫폼에 매력을 느낀 다른 언어의 구현자는 Java 가상 머신을 해당 언어의 전달 수단으로 사용할 수 있습니다. ((1)) (Java 가상 머신)

JVM 아키텍처 **핫스팟 아키텍처** 살펴보기

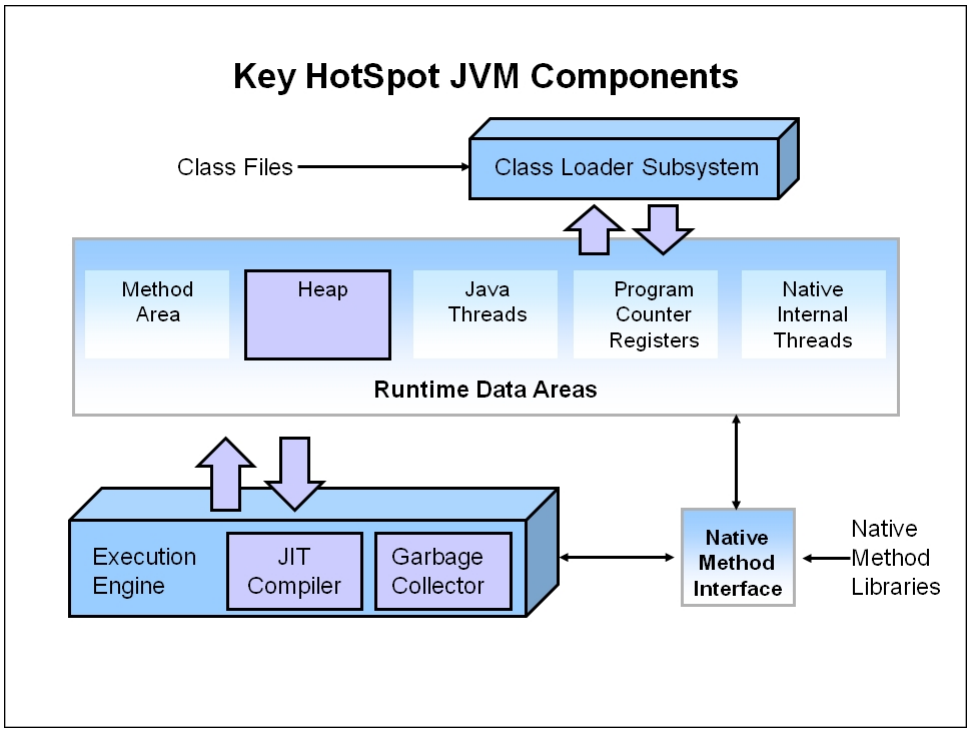
HotSpot JVM은 강력한 기능 기반을 지원하는 아키텍처를 갖추고 있으며 고성능과 대규모 확장성을 실현할 수 있는 기능을 지원합니다. 예를 들어, HotSpot JVM JIT 컴파일러는 동적 최적화를 생성합니다. 즉, Java 애플리케이션이 실행되는 동안 최적화 결정을 내리고 기본 시스템 아키텍처를 대상으로 하는 고성능 네이티브 머신 명령어를 생성합니다. 또한 런타임 환경과 멀티스레드 가비지 컬렉터의 발전과 지속적인 엔지니어링을 통해 HotSpot JVM은 사용 가능한 가장 큰 컴퓨터 시스템에서도 높은 확장성을 제공합니다.



JVM의 주요 구성 요소로는 클래스 로더, 런타임 데이터 영역, 실행 엔진이 있습니다.

주요 핫스팟 구성 요소

성능과 관련된 JVM의 주요 구성 요소는 다음 이미지에서 강조 표시되어 있습니다.



성능을 조정할 때 중점을 두는 JVM에는 세 가지 구성 요소가 있습니다. 힙은 객체 데이터가 저장되는 곳입니다. 이 영역은 시작 시 선택한 가비지 컬렉터에 의해 관리됩니다. 대부분의 튜닝 옵션은 힙의 크기를 조정하고 상황에 가장 적합한 가비지 컬렉터를 선택하는 것과 관련이 있습니다. JIT 컴파일러도 성능에 큰 영향을 미치지만 최신 버전의 JVM에서는 튜닝이 거의 필요하지 않습니다.

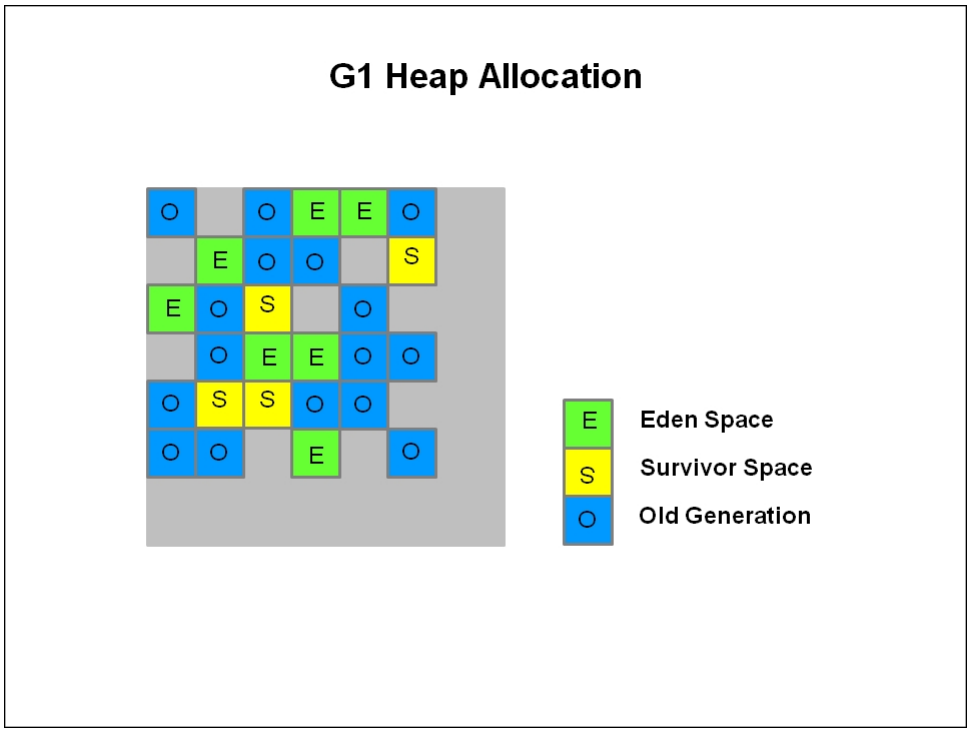
성능 기본 사항

일반적으로 Java 애플리케이션을 튜닝할 때는 응답성 또는 처리량이라는 두 가지 주요 목표 중 하나에 중점을 둡니다. 튜토리얼을 진행하면서 이러한 개념을 다시 언급하겠습니다.

G1 가비지 컬렉터

모든 메모리 객체는 이 세 섹션 중 하나에 저장됩니다. G1 수집기는 다른 접근 방식을 취합

니다.



힙은 각각 인접한 범위의 가상 메모리인 동일한 크기의 힙 영역 집합으로 분할됩니다. 특정 영역 세트에는 이전 컬렉터와 동일한 역할(eden, survivor, old)이 할당되지만 고정된 크기는 없습니다. 따라서 메모리 사용의 유연성이 향상됩니다.

가비지 컬렉션을 수행할 때 G1은 CMS 수집기와 유사한 방식으로 작동합니다. G1은 동시 글로벌 마킹 단계를 수행하여 힙 전체에서 객체의 활성도를 결정합니다. 마크 단계가 완료되면 G1은 어느 영역이 대부분 비어 있는지 파악합니다. 이 영역에서 먼저 수집하므로 일반적으로 많은 양의 여유 공간이 확보됩니다. 이 가비지 수집 방법을 가비지 우선이라고 부르는 이유입니다. 이름에서 알 수 있듯이 G1은 힙에서 회수 가능한 개체, 즉 쓰레기로 가득 차 있을 가능성이 높은 영역에 수집 및 압축 활동을 집중합니다. G1은 일시 중지 예측 모델을 사용하여 사용자가 정의한 일시 중지 시간 목표를 충족하고 지정된 일시 중지 시간 목표에 따라 수집할 영역 수를 선택합니다.

G1이 매립할 준비가 되었다고 식별한 영역은 이비전을 사용하여 수집한 가비지입니다. G1은 힙의 하나 이상의 영역에서 힙의 단일 영역으로 객체를 복사하며, 이 과정에서 메모리를 압축하고 확보합니다. 이 비우기는 일시 중지 시간을 줄이고 처리량을 높이기 위해 멀티 프로세서에서 병렬로 수행됩니다. 따라서 가비지 컬렉션을 수행할 때마다 G1은 사용자가 정의한 일시 정지 시간 내에서 작동하면서 조각화를 줄이기 위해 지속적으로 노력합니다. 이는 이전 두 가지 방법의 기능을 뛰어넘는 것입니다. CMS(동시 마크 스윕) 가비지 수집기는 압축을 수행하지 않습니다. ParallelOld 가비지 수집은 전체 힙 압축만 수행하므로 일시 중지 시간이 상당히 길어집니다.

G1은 실시간 수집기가 아니라는 점에 유의해야 합니다. 설정된 일시 정지 시간 목표를 높은 확률로 충족하지만 절대적인 확률은 아닙니다. G1은 이전 수집 데이터를 기반으로 사용자가 지정한 목표 시간 내에 얼마나 많은 영역을 수집할 수 있는지 추정합니다. 따라서 수집기는 영역 수집 비용에 대한 합리적으로 정확한 모델을 갖게 되며, 이 모델을 사용하여 일시 정지 시간 목표 내에 머무르면서 수집할 영역과 그 수를 결정합니다.

참고: G1에는 동시(애플리케이션 스레드와 함께 실행, 예: 정제, 마킹, 정리) 단계와 병렬(멀티스레드, 예: 스톱 더 월드) 단계가 모두 있습니다. 전체 가비지 컬렉션은 여전히 단일 스레드이지만 애플리케이션을 적절히 튜닝하면 전체 GC를 피할 수 있습니다.

G1 발자국

ParallelOldGC 또는 CMS 수집기에서 G1으로 마이그레이션하는 경우, JVM 프로세스 크기가 더 커질 수 있습니다. 이는 주로 기억된 세트 및 컬렉션 세트와 같은 "회계" 데이터 구조와 관련이 있습니다.

기억된 세트 또는 RSet은 지정된 영역에 대한 객체 참조를 추적합니다. 힙에는 영역당 하나의 RSet이 있습니다. RSet을 사용하면 영역을 병렬로 독립적으로 수집할 수 있습니다. RSet의 전체 풋프린트 영향은 5% 미만입니다.

수집 세트 - GC에서 수집할 영역의 집합을 설정합니다. CSet의 모든 라이브 데이터는 GC 중에 비워집니다(복사/이동). 지역 세트는 에덴, 생존자 및/또는 구세대가 될 수 있습니다. CS셋은 JVM의 크기에 1% 미만의 영향을 미칩니다.

G1의 권장 사용 사례

G1의 첫 번째 초점은 제한된 GC로 대용량 힙을 필요로 하는 애플리케이션을 실행하는 사용자를 위한 솔루션을 제공하는 것입니다.

지연 시간. 이는 약 6GB 이상의 힙 크기와 0.5초 미만의 안정적이고 예측 가능한 일시 정지 시간을 의미합니다.

현재 실행 중인 애플리케이션에 다음 특성 중 하나 이상이 있는 경우 CMS 또는 ParallelOldGC 가비지 수집기를 사용하여 G1으로 전환하는 것이 좋습니다.

- 전체 GC 시간이 너무 길거나 너무 자주 발생합니다.
- 객체 할당률 또는 프로모션 비율은 크게 달라집니다.
- 원치 않는 긴 쓰레기 수거 또는 압축 일시정지(0.5~1초 이상)

참고: CMS 또는 ParallelOldGC를 사용 중이고 애플리케이션에서 가비지 수집이 오래 일시 중지되지 않는다면 현재 수집기를 그대로 사용해도 괜찮습니다. G1 수집기로 변경하는 것이 최신 JDK를 사용하기 위한 필수 요건은 아닙니다.

CMS로 GC 검토하기

세대별 GC 및 CMS 검토

동시 마크 스윕(CMS) 수집기(동시 낮은 일시 중지 수집기라고도 함)는 테뉴어드 세대를 수집합니다. 대부분의 가비지 수집 작업을 애플리케이션 스레드와 동시에 수행하여 가비지 수집으로 인한 일시 중지를 최소화하려고 시도합니다. 일반적으로 동시 로우 일시 중지 수집기는 라이브 객체를 복사하거나 압축하지 않습니다. 가비지 컬렉션은 라이브 오브젝트를 이동하지 않고 수행됩니다. 조각화가 문제가 되는 경우 더 큰 힙을 할당하세요.

참고: 젊은 세대의 CMS 수집기는 병렬 수집기의 알고리즘과 동일한 알고리즘을 사용합니다.

CMS 수집 단계

CMS 수집기는 이전 세대의 힙에 대해 다음 단계를 수행합니다:

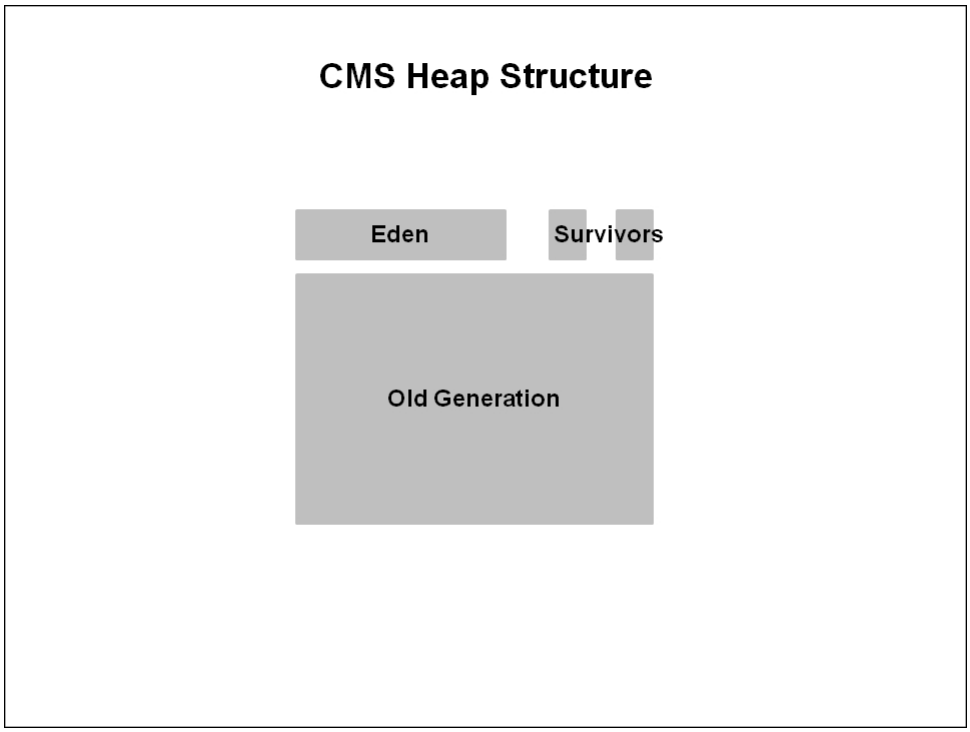
단계	설명
(1) 이니셜 마크 *(월드 이벤트 중지)*	오래된 세대의 오브젝트는 젊은 세대에서 도달할 수 있는 오브젝트를 포함하여 도달 가능한 것으로 '표시'됩니다. 일시 중지 시간은 일반적으로 사소한 수집 일시 중지 시간에 비해 상대적으로 짧습니다.
(2) 동시 표시	Java 애플리케이션 스레드가 실행되는 동안 동시에 도달 가능한 객체에 대해 테뉴어드 생성 객체 그래프를 탐색합니다. 표시된 객체에서 스캔을 시작하고 루트에서 도달 가능한 모든 객체를 일시적으로 표시합니다. 뮤테이터는 동시 2, 3, 5 단계에서 실행되며 이러한 단계에서 CMS 생성에 할당된 모든 객체(승격된 객체 포함)는 즉시 라이브 상태로 표시됩니다.
(3) 비고 *(월드 이벤트 중지)*	동시 수집기가 해당 객체 추적을 완료한 후 Java 애플리케이션 스레드가 객체를 업데이트하여 동시 마크 단계에서 놓친 객체를 찾습니다.
(4) 동시 스윕	마킹 단계에서 도달할 수 없는 것으로 식별된 개체를 수집합니다. 죽은 개체를 수집하면 나중에 할당할 수 있도록 해당 개체를 위한 공간이 여유 목록에 추가됩니다. 이 시점에서 죽은 개체가 합쳐질 수 있습니다. 살아있는 개체는 이동되지 않는다는 점에 유의하세요.
(5) 재설정	데이터 구조를 지워 다음 동시 수집을 준비하세요.

가비지 수집 단계 검토

다음으로 CMS 수집기 작업을 검토해 보겠습니다.

1. **CMS 수집기의 힙 구조**

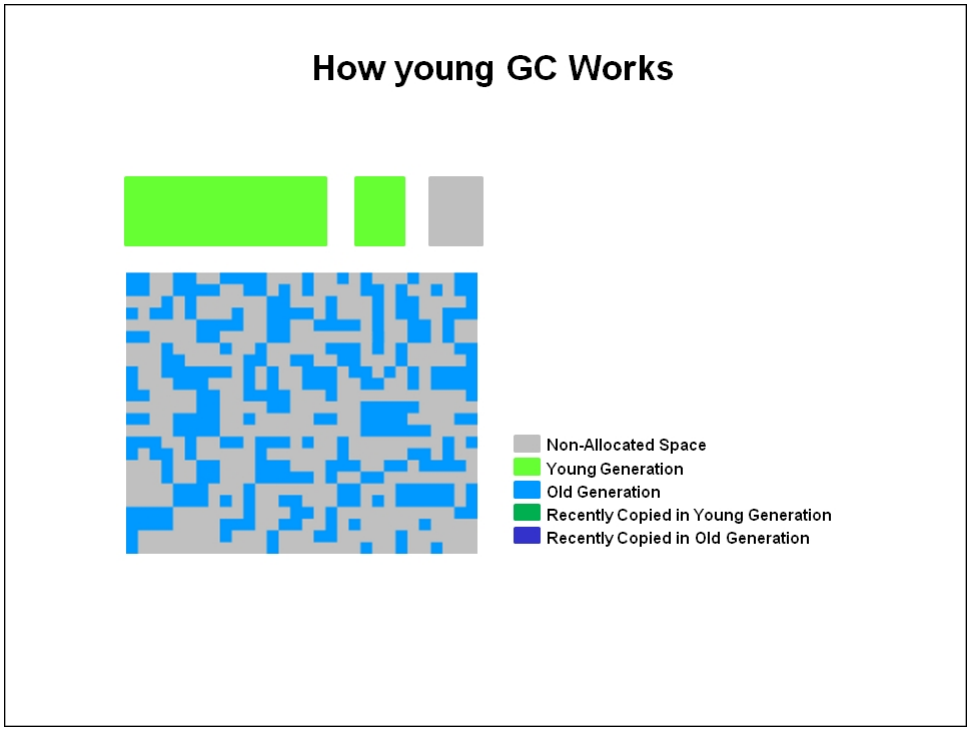
   힙은 세 개의 공간으로 나뉩니다.



젊은 세대는 에덴과 두 개의 생존자 공간으로 나뉩니다. 구세대는 하나의 연속된 공간입니다. 오브젝트 수집은 제자리에서 이루어집니다. 전체 GC가 없는 한 압축은 수행되지 않습니다.

2. Young GC가 CMS에서 작동하는 방식

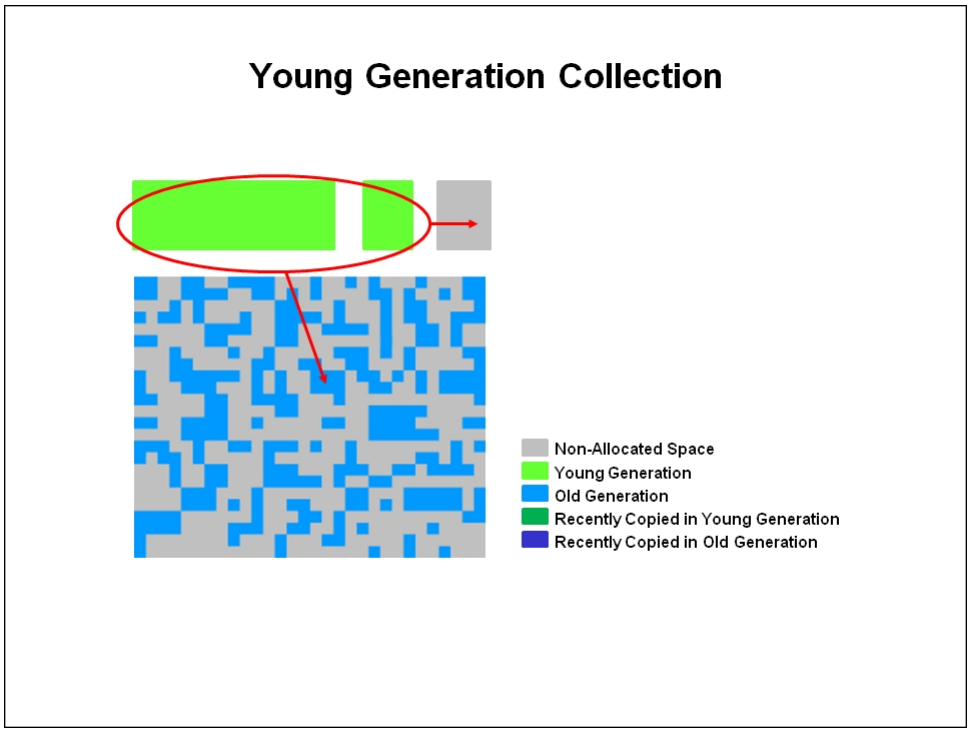
젊은 세대는 연한 녹색으로, 노년층은 파란색으로 표시됩니다. 애플리케이션이 한동안 실행된 경우 CMS가 이렇게 수 있습니다. 구세대 영역에는 오브젝트가 흩어져 있습니다.



CMS를 사용하면 이전 세대의 개체는 제자리에서 할당 해제됩니다. 않습니다. 전체 GC가 없는 한 공간이 압축되지 않습니다.

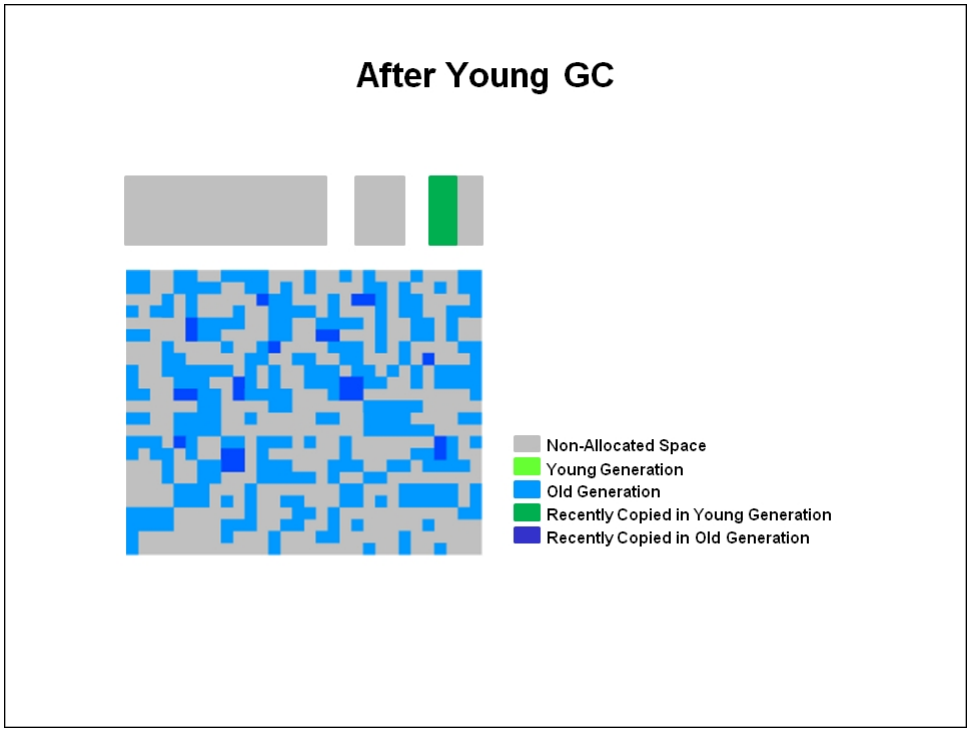
3. 젊은 세대 컬렉션

살아있는 오브젝트는 에덴 공간과 생존자 공간에서 다른 생존자 복사됩니다. 에이징 임계값에 도달한 오래된 오브젝트는 모두 오래된 세대로 승격됩니다.



4. 젊은 GC 이후

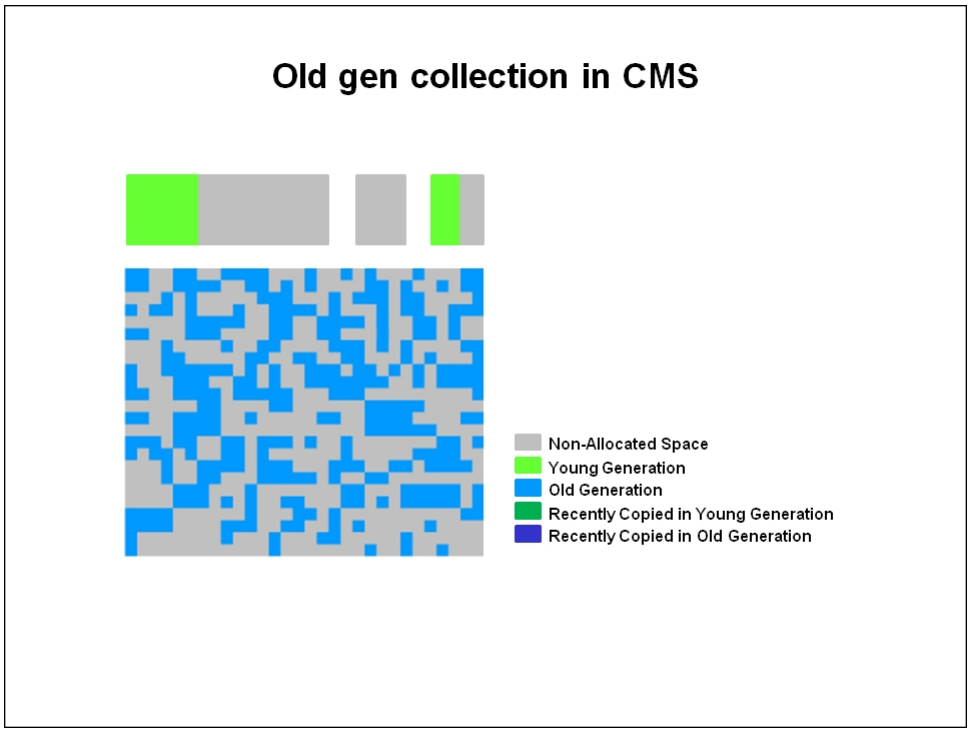
어린 GC가 죽으면 에덴 공간이 클리어되고 생존자 공간 중 하나가 클리어됩니다.



새로 승격된 개체는 다이어그램에서 진한 파란색으로 표시됩니다. 녹색 개체는 아직 구세대로 승격되지 않은 살아남은 젊은 세대 개체입니다.

5. CMS를 사용한 구세대 컬렉션

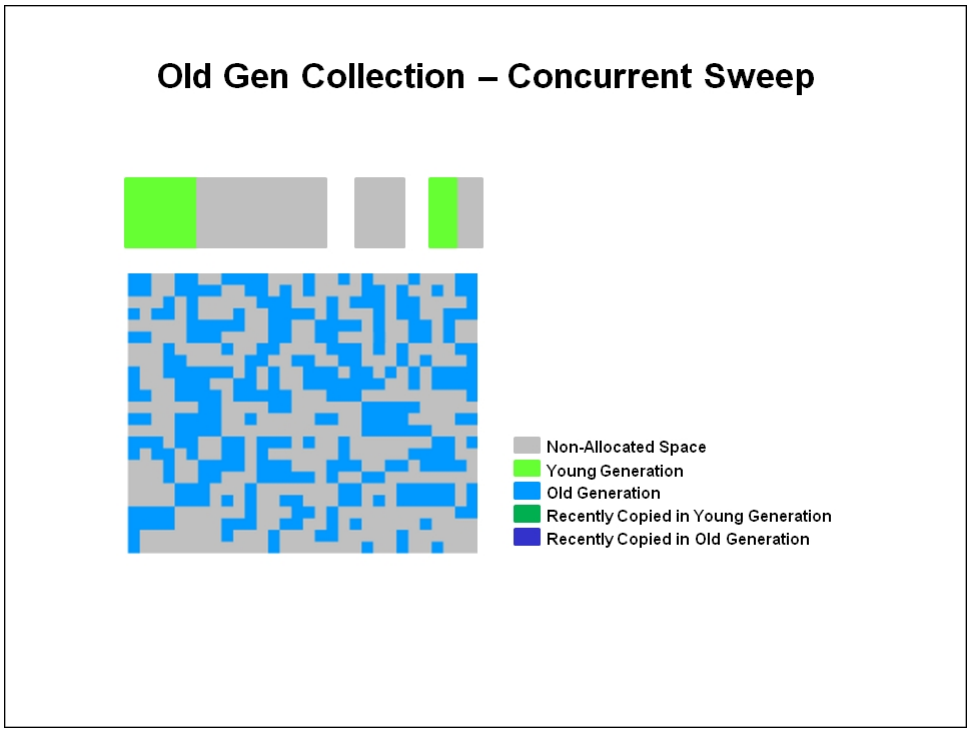
이니셜 마크와 발언이라는 두 가지 이벤트가 발생합니다. 구세대가 특정 점유율에 도달하면 CMS가 시작됩니다.



(1) 초기 표시는 라이브(도달 가능한) 개체가 표시되는 짧은 일시 정지 단계입니다. (2) 동시 마킹은 애플리케이션이 계속 실행되는 동안 라이브 개체를 찾습니다. 마지막으로 (3) 리마킹 단계에서는 이전 단계의 (2) 동시 마킹 중에 놓친 개체를 찾습니다.

6. 구세대 컬렉션 - 동시 스윕

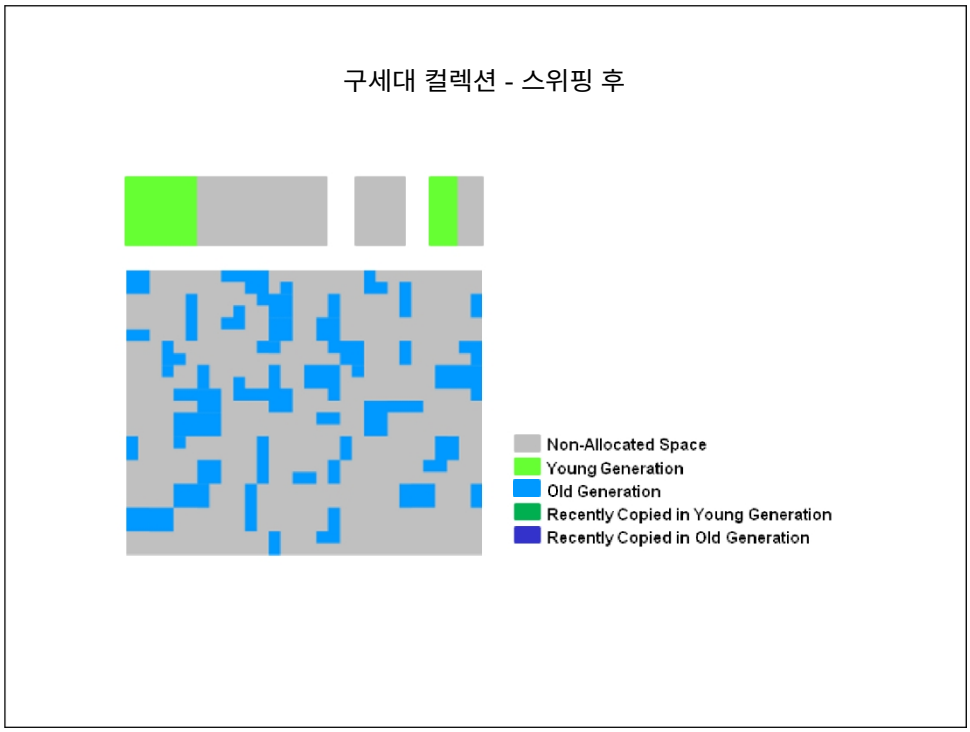
이전 단계에서 표시되지 않은 개체는 제자리에서 할당 해제됩니다. 압축이 없습니다.

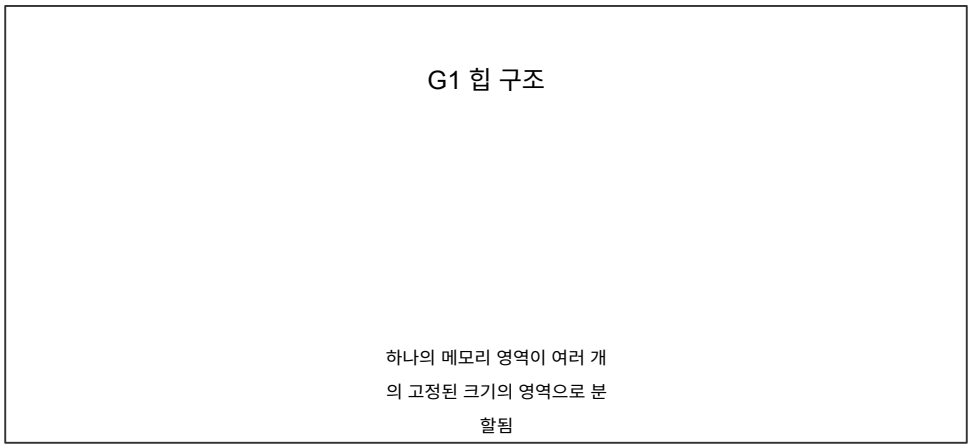


참고: 표시되지 않은 개체== 죽은 개체

7. 구세대 컬렉션 - 스위핑 후

(4) 스위핑 단계가 끝나면 많은 메모리가 확보된 것을 볼 수 있습니다. 또한 압축이 수행되지 않은 것을 알 수 있습니다.

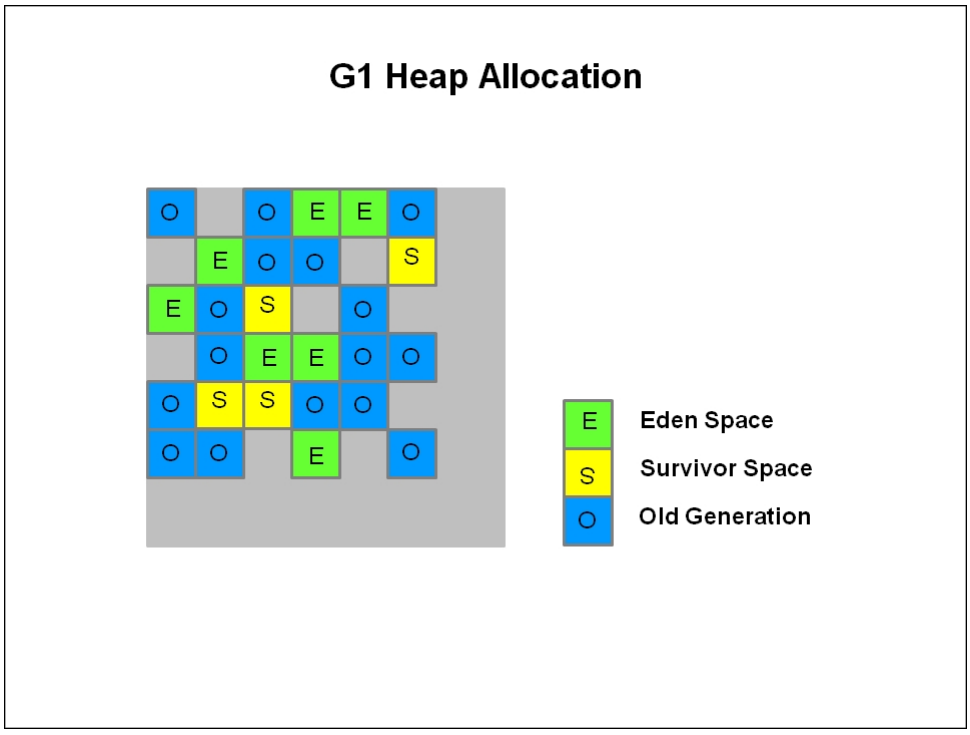




리전 크기는 시작 시 JVM이 선택합니다. JVM은 일반적으로 1~32Mb 크기의 다양한 약 2000개 리전을 대상으로 합니다.

2. G1 힙 할당

실제로 이러한 영역은 에덴, 생존자, 구세대 공간의 논리적인 표현으로 매핑됩니다.



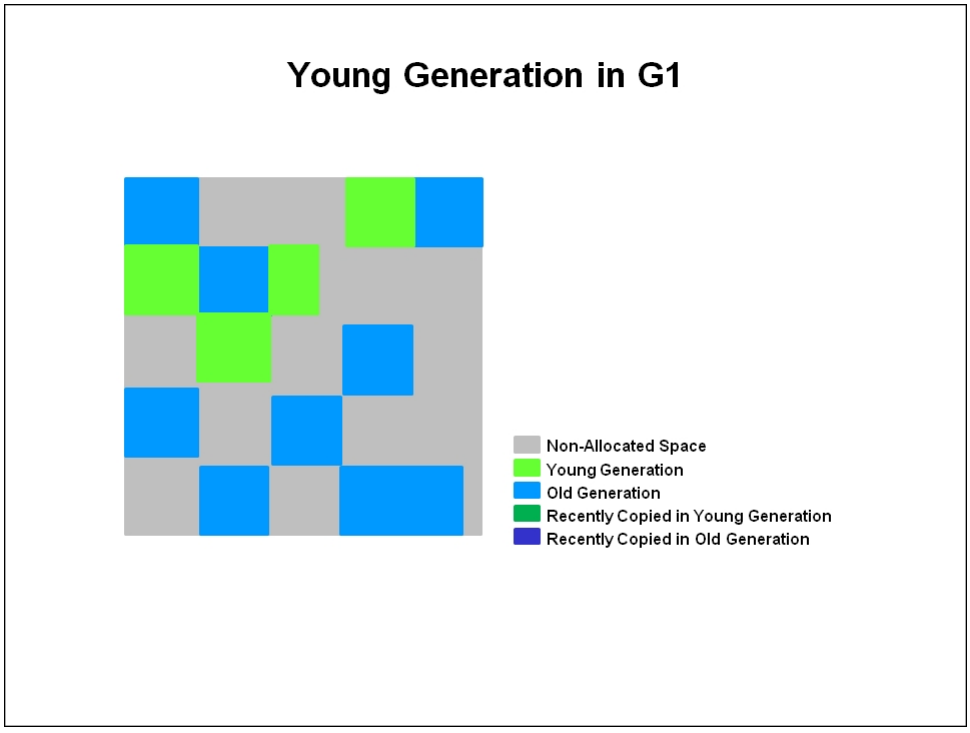
그림의 색상은 어떤 영역이 어떤 역할과 연관되어 있는지 나타냅니다. 라이브 개체는 한 영역에서 다른 영역으로 이동(즉, 복사 또는 이동)됩니다. 영역은 다른 모든 애플리케이션 스레드를 중지하거나 중지하지 않고 병렬로 수집하도록 설계되었습니다.

표시된 것처럼 지역은 에덴, 생존자, 구세대 지역으로 할당할 수 있습니다. 또한 휴먼 지역이라는 네 번째 유형의 오브젝트가 있습니다. 이 영역은 표준 영역의 50% 크기 이상의 개체를 보관할 수 있도록 설계되었습니다. 인접한 영역의 집합으로 저장됩니다. 마지막으로 마지막 유형의 영역은 힙의 미사용 영역입니다.

참고: 이 글을 작성하는 시점에서는 거대한 오브젝트를 수집하는 기능이 최적화되지 않았습니다. 따라서 이 크기의 개체는 만들지 않는 것이 좋습니다.

3. G1의 젊은 세대

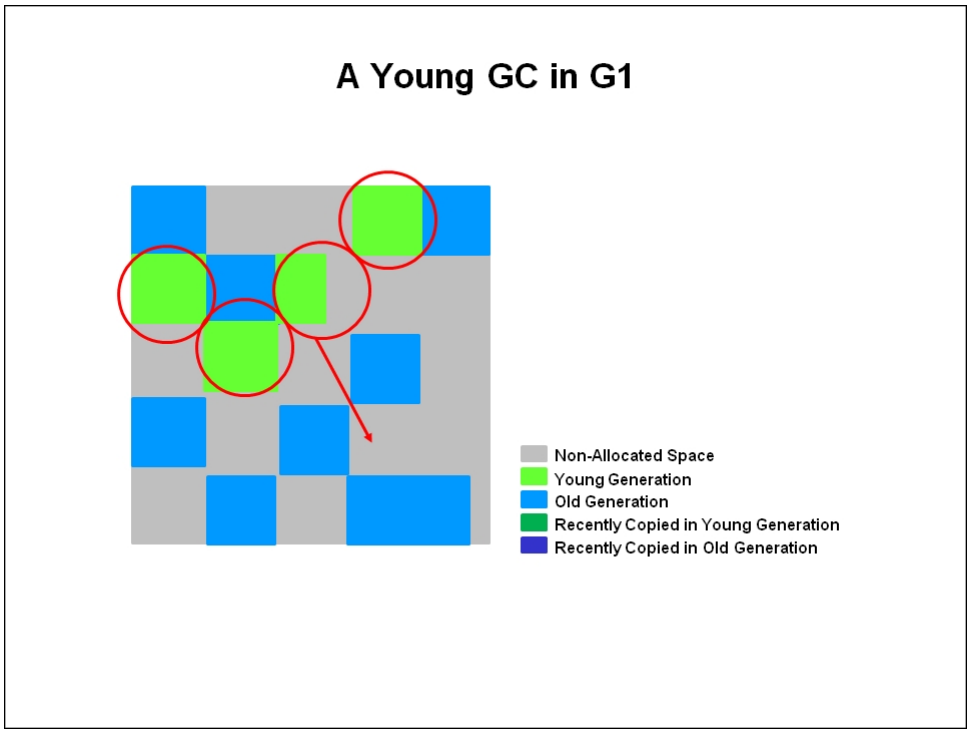
힙은 약 2000개의 영역으로 나뉩니다. 최소 크기는 1Mb, 최대 크기는 32Mb입니다. 파란색 영역은 오래된 세대 오브젝트를, 녹색 영역은 젊은 세대 오브젝트를 보관합니다.



이전 가비지 수집기처럼 영역이 연속적일 필요는 없습니다.

4. G1의 젊은 GC

라이브 오브젝트는 하나 이상의 생존자 영역으로 대피(즉, 복사 또는 이동)됩니다. 에이징 충족되면 일부 오브젝트는 이전 세대 리전으로 승격됩니다.

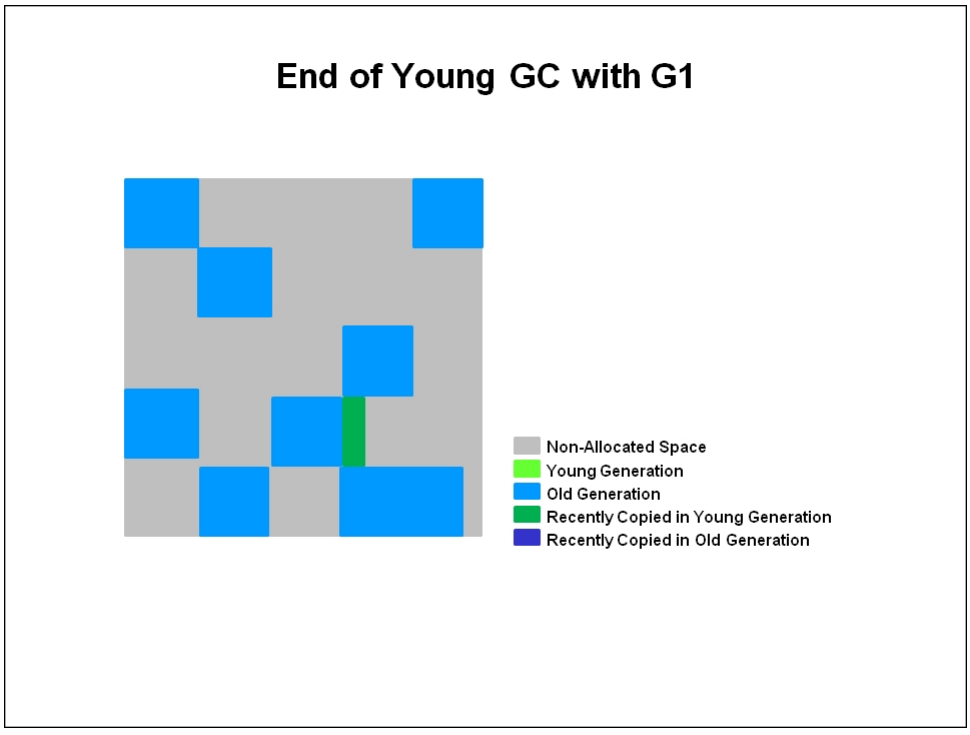


이것은 스톱 더 월드(STW) 일시정지입니다. 에덴의 크기와 생존자 크기는 다음 젊은 GC를 위해 계산됩니다. 규모 계산을 돕기 위해 회계 정보가 보관됩니다. 일시 정지 시간 목표와 같은 것들이 고려됩니다.

이 접근 방식을 사용하면 영역의 크기를 매우 쉽게 조정하여 필요에 따라 더 크게 또는 더 작게 만들 수 있습니다.

5. G1과 함께한 젊은 GC의 끝

살아있는 물체는 생존자 지역이나 구세대 지역으로 대피했습니다.



최근 승격된 개체는 진한 파란색으로 표시됩니다. 생존자 지역은 녹색으로 표시됩니다. 요약하면 G1의 젊은 세대에 대

해 다음과 같이 말할 수 있습니다:

- 힙은 영역으로 분할된 단일 메모리 공간입니다.
- 젊은 세대 메모리는 비연속적인 영역 세트로 구성됩니다. 따라서 필요할 때 크기를 쉽게 조정할 수 있습니다.
- 젊은 세대 가비지 컬렉션, 즉 젊은 GC가 월드 이벤트를 중단합니다. 모든 애플리케이션 스레드가 작업을 위해 중지됩니다.
- 젊은 GC는 여러 스레드를 사용하여 병렬로 수행됩니다.

- 라이브 오브젝트는 새로운 생존자 또는 이전 세대 영역으로 복사됩니다.

G1과 함께하는 올드 제너레이션 컬렉션

CMS 수집기와 마찬가지로 G1 수집기는 이전 세대 개체에 대한 낮은 일시 중지 수집기로 설계되었습니다. 다음 표에서는 이전 세대의 G1 수집 단계에 대해 설명합니다.

G1 수집 단계 - 동시 마킹 주기 단계

G1 수집기는 힙의 이전 세대에 대해 다음 단계를 수행합니다. 일부 단계는 젊은 세대 컬렉션의 일부라는 점에 유의하세요.

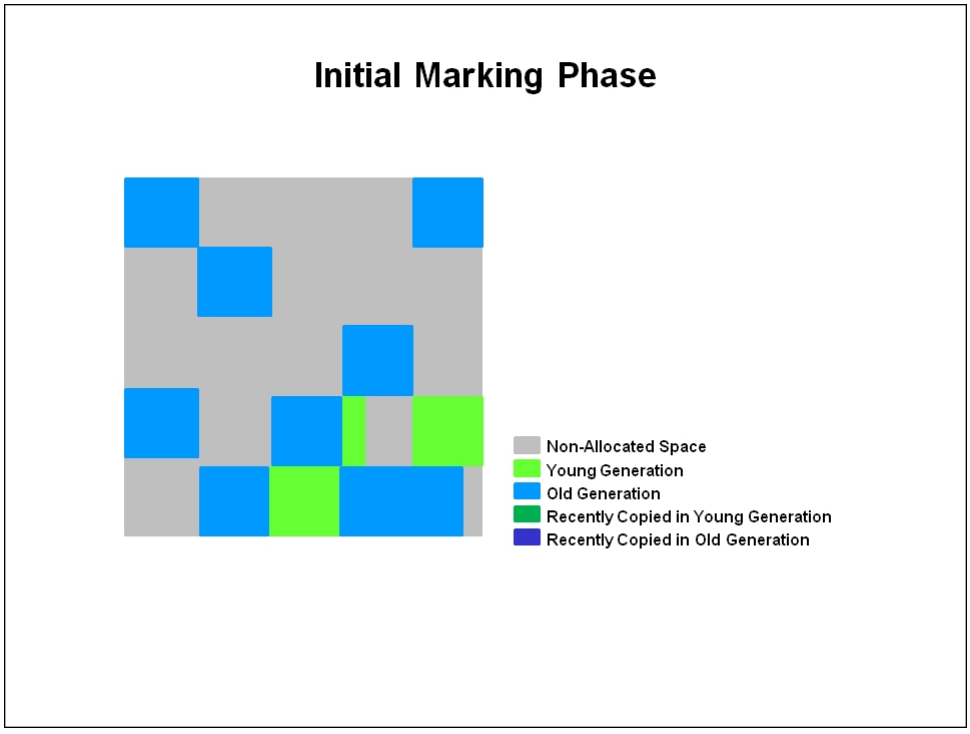
단계	설명
(1) 이니셜 마크 *(월드 이벤트 중지)*	이것은 스톱 더 월드 이벤트입니다. G1을 사용하면 일반 젊은 GC에 피기백됩니다. 이전 세대의 오브젝트에 대한 참조가 있을 수 있는 생존자 영역(루트 영역)을 표시합니다.
(2) 루트 영역 스캔	생존자 영역에서 이전 세대에 대한 참조를 검색합니다. 이 과정은 애플리케이션이 계속 실행되는 동안 발생합니다. 이 단계는 반드시 완료되어야 젊은 GC가 발생할 수 있습니다.
(3) 동시 표시	전체 힙에서 라이브 오브젝트를 찾습니다. 이 작업은 애플리케이션이 실행되는 동안 발생합니다. 이 단계는 젊은 세대 가비지 컬렉션에 의해 중단될 수 있습니다.
(4) 비고 *(월드 이벤트 중지)*	힙에 있는 라이브 오브젝트의 마킹을 완료합니다. CMS 수집기에서 사용했던 것보다 훨씬 빠른 스냅샷 시작 시점(SATB)이라는 알고리즘을 사용합니다.
(5) 정리 *(월드 이벤트 및 동시 진행 중지)*	• 라이브 오브젝트와 완전 자유 영역에 대한 회계를 수행합니다. (세계 중지) • 기억된 세트를 스크럽합니다. (세상 멈추기) • 비어 있는 영역을 재설정하고 무료 목록으로 되돌립니다. (동시)
(*) 복사 *(월드 이벤트 중지)*	사용하지 않는 새로운 리전으로 라이브 오브젝트를 대피시키거나 복사하기 위해 월드가 일시 정지하는 것입니다. 이 작업은 `[GC 일시정지(젊은)]`로 기록된 젊은 세대 영역에서 수행할 수 있습니다. 또는 `[GC 일시정지(혼합)]`로 기록되는 젊은 세대 및 구세대 영역 모두에서 가능합니다.

G1 구세대 컬렉션 단계별

단계가 정의되었으니 이제 G1 수집기에서 이전 세대와 어떻게 상호 작용하는지 살펴보겠습니다.

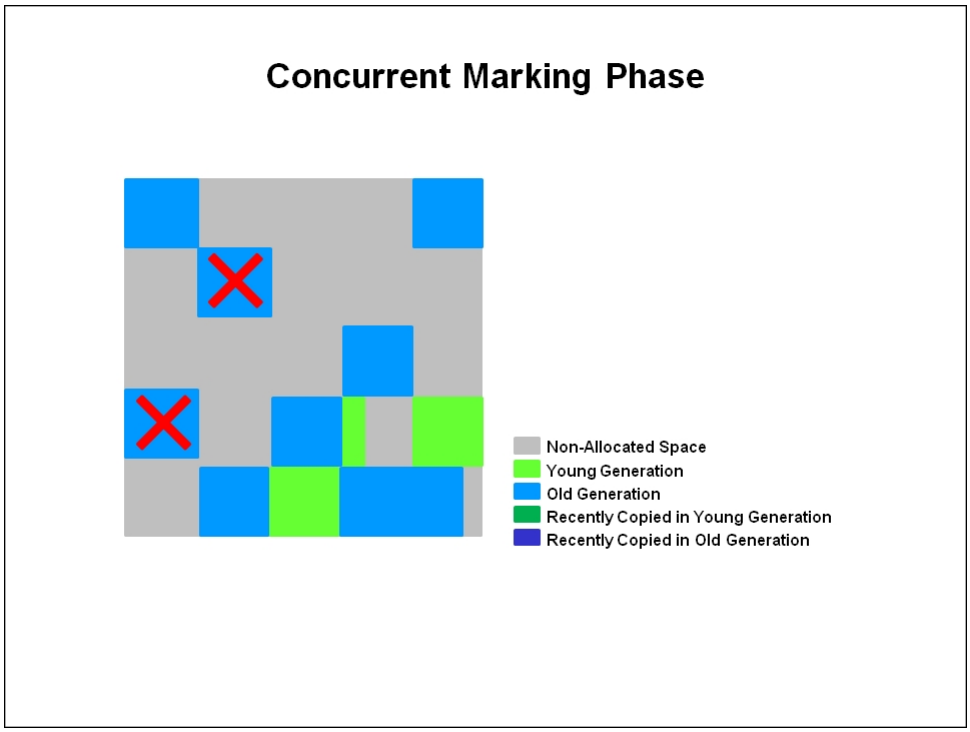
6. 초기 마킹 단계

라이브 오브젝트의 초기 마킹은 젊은 세대 가비지 컬렉션에 피기백됩니다. 로그에는 `GC 일시 중지(젊은)(초기 마크)`로 표시됩니다.



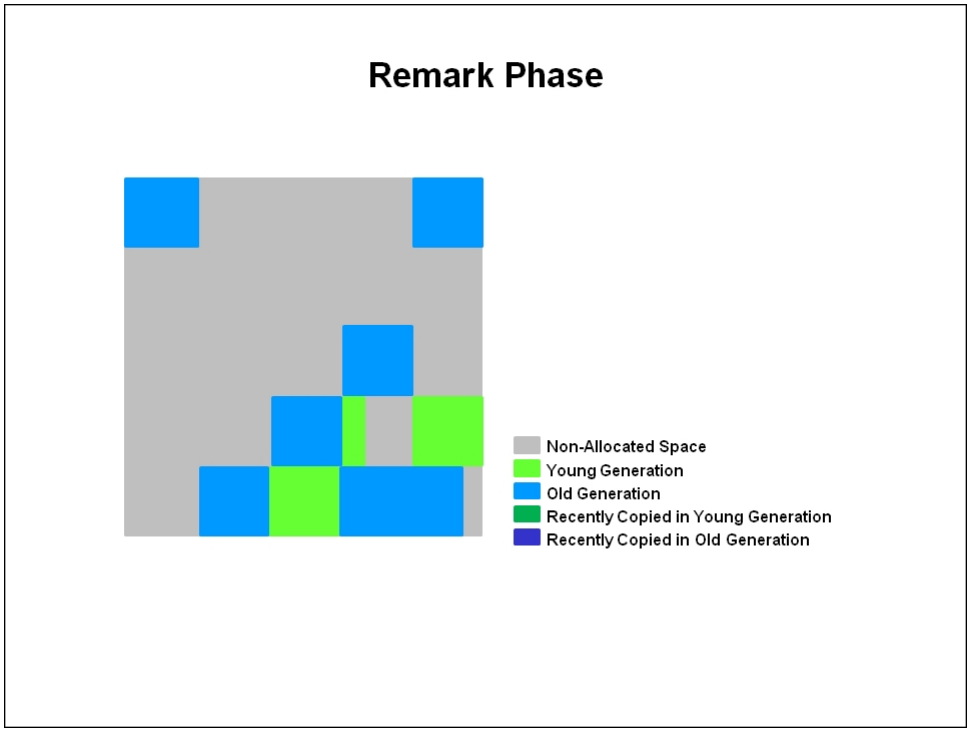
7. 동시 마킹 단계

비어 있는 영역이 발견되면('X'로 표시됨) 비고 단계에서 즉시 제거됩니다. 또한 활력을 결정하는 '회계' 정보도 계산됩니다.



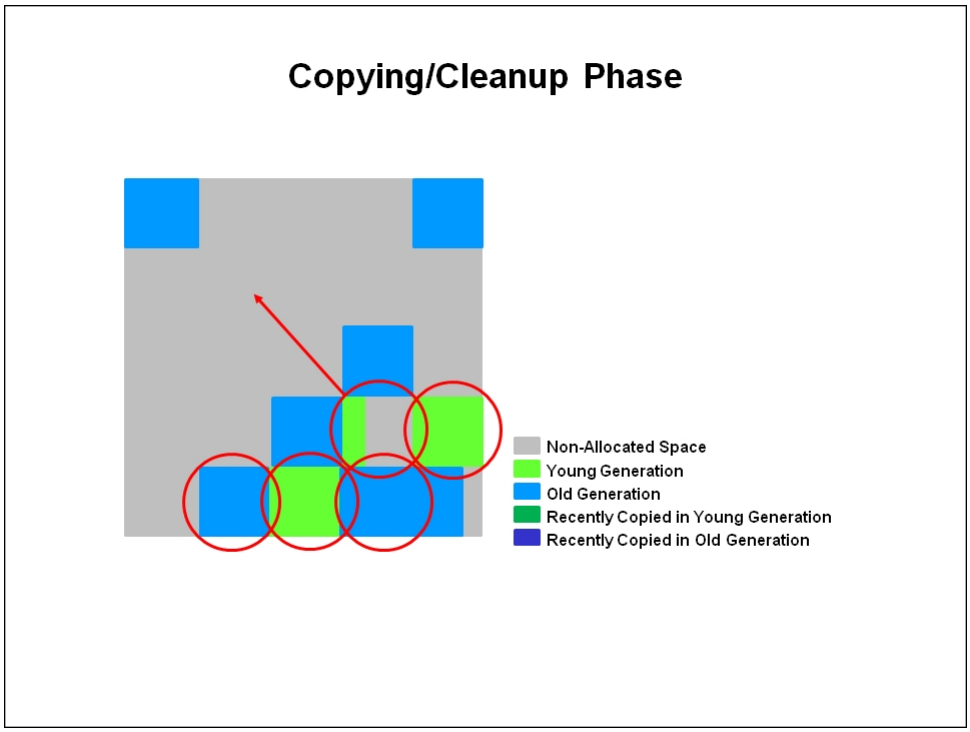
8. 비고 단계

비어 있는 지역은 제거되었다가 다시 채워집니다. 이제 모든 지역에 대해 지역 활력이 계산됩니다.



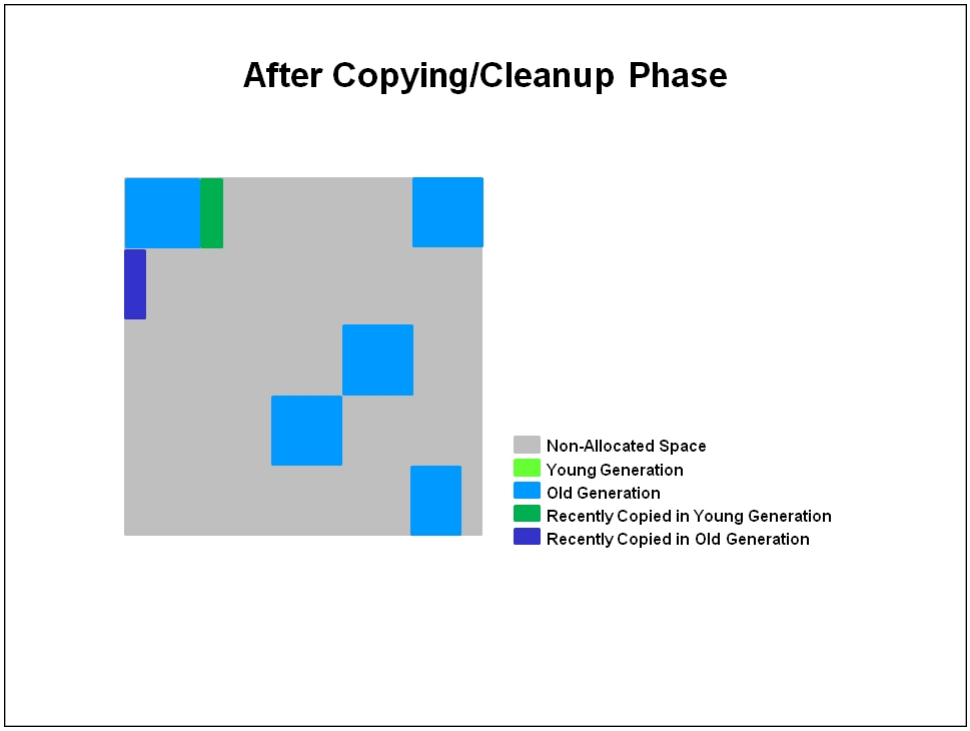
9. 복사/정리 단계

G1은 '활력도'가 가장 낮은 지역, 즉 가장 빠르게 수집할 수 있는 지역을 선택합니다. 그런 다음 해당 영역은 젊은 GC와 동시에 수집됩니다. 이는 로그에 `[GC 일시 중지(혼합)]`로 표시됩니다. 따라서 젊은 세대와 오래된 세대가 동시에 수집됩니다.



10. 복사/정리 단계 후

선택한 영역이 수집되어 다이어그램에 표시된 진한 파란색 영역과 진한 녹색 영역으로 압축되었습니다.



구세대 GC 요약

요약하면, 이전 세대의 G1 가비지 컬렉션에 대해 몇 가지 핵심 사항을 확인할 수 있습니다.

- 동시 마킹 단계
  - 활력도 정보는 애플리케이션이 실행되는 동안 동시에 계산됩니다.
  - 이 생존자 정보를 통해 대피 일시정지 기간 동안 어느 지역을 복구하는 것이 가장 좋은지 파악할 수 있습니다.
  - CMS와 같은 스위핑 단계는 없습니다.
- 비고 단계
  - CMS에서 사용하던 것보다 훨씬 빠른 SATB(Snapshot-at-the-Beginning) 알고리즘을 사용합니다.
  - 완전히 비어 있는 지역은 재생됩니다.
- 복사/정리 단계
  - 젊은 세대와 노년 세대가 동시에 재생됩니다.

- 구세대 지역은 활기를 기준으로 선정됩니다.

명령줄 옵션 및 모범 사례

명령줄 옵션 및 모범 사례

이 섹션에서는 G1의 다양한 명령줄 옵션을 살펴보겠습니다.

기본 명령줄

G1 수집기를 사용하려면 `-XX:+UseG1GC`

다음은 JDK 데모 및 샘플 다운로드에 포함된 Java2Demo를 시작하기 위한 샘플 명령줄입니다: **`java -Xmx50m -Xms50m -XX:+UseG1GC -XX:MaxGCPauseMillis=200 -jar c:\javademos\demo\jfc\Java2D\Java2demo.jar`**.

주요 명령줄 스위치

`-XX:+UseG1GC` - JVM에 G1 가비지 컬렉터를 사용하도록 지시합니다.

-XX:MaxGCPauseMillis=200 - 최대 GC 일시 중지 시간 목표를 설정합니다. 이는 소프트 목표이며, JVM은 이를 달성하기 위해 최선을 다합니다. 따라서 일시 중지 시간 목표가 않는 경우가 있습니다. 기본값은 200밀리초입니다.

-XX:InitiatingHeapOccupancyPercent=45 - 동시 GC 사이클을 시작할 (전체) 힙 점유율의 백분율입니다. 이 값은 G1에서 세대 하나가 아닌 전체 힙의 점유율을 기반으로 동시 GC 주기를 트리거하는 데 사용됩니다. 값이 0이면 '일정한 GC 주기를 수행'함을 나타냅니다. 기본값은 45(즉, 45% 가득 차거나 점유됨)입니다.

모범 사례

G1을 사용할 때 따라야 할 몇 가지 모범 사례가 있습니다.

젊은 세대 크기 설정 안 함

`Xmn`을 통해 명시적으로 젊은 세대 크기를 설정하면 G1 수집기의 기본 동작에 간섭하게 됩니다.

- G1은 더 이상 컬렉션의 일시 중지 시간 목표를 준수하지 않습니다. 따라서 본질적으로 젊은 세대 크기를 설정하면 일시 중지 시간 목표가 비활성화됩니다.
- G1은 더 이상 필요에 따라 젊은 세대 공간을 확장하거나 축소할 수 없습니다. 크기가 고정되어 있으므로 크기를 변경할 수 없습니다.

응답 시간 메트릭

평균 응답 시간(ART)을 메트릭으로 사용하여 `XX:MaxGCPauseMillis=<N>`을 설정하는 대신 목표를 90% 이상 충족하는 값을 설정하는 것을 고려하세요. 즉, 요청을 하는 사용자의 목표보다 높은 응답 시간을 경험하지 않게 됩니다. 일시 정지 시간은 목표이며 항상 충족된다는 보장은 없다는 점을 기억하세요.

대피 실패란 무엇인가요?

생존자 및 승격된 객체에 대한 GC 중에 JVM의 힙 영역이 부족할 때 발생하는 승격 실패입니다. 힙이 이미 최대에 도달했기 때문에 힙을 확장할 수 없습니다. 이는 `-XX:+PrintGCDetails`를 사용할 때 GC 로그에 **공간 오버플로로** 표시됩니다. 비용이 많이 듭니다!

- GC는 여전히 계속되어야 하므로 공간을 합니다.
- 복사하지 못한 객체는 제자리에 유지되어야 합니다.
- CSet에 있는 지역의 RSet에 대한 모든 업데이트는 다시 생성해야 합니다.
- 이 모든 단계는 비용이 많이 듭니다.

대피 실패를 피하는 방법

대피 실패를 방지하려면 다음 옵션을 고려하세요.

- 힙 크기 늘리기
  - **`XX:G1ReservePercent=n을`** 증가시키면 기본값은 10입니다.
  - G1은 더 많은 '투 스페이스'가 필요할 경우를 대비해 예비 메모리를 비워두는 방식으로 잘못된 상한선을 생성합니다.
- 마킹 주기를 더 일찍 시작하기
- **`-XX:ConcGCThreads=n`** 옵션을 사용하여 마킹 스레드 수를 늘립니다.

G1 GC 스위치 전체 목록

G1 GC 스위치의 전체 목록입니다. 위에서 설명한 모범 사례를 사용해야 합니다.

옵션 및 기본값	설명
-XX:+UseG1GC	가비지 우선(G1) 수집기 사용

-XX:MaxGCPauseMillis=n	최대 GC 일시 중지 시간의 목표를 설정합니다. 이는 소프트 목표이며 이를 달성하기 위해 최선을 다할 것입니다.

-XX:InitiatingHeapOccupancyPercent=n	동시 GC 사이클을 시작하기 위한 (전체) 힙 점유율의 백분율입니다. 세대 중 하나(예: G1)가 아닌 전체 힙의 점유율을 기반으로 동시 GC 주기를 트리거하는 GC에서 사용됩니다. 값이 0이면 '일정한 GC 주기를 수행'함을 나타냅니다. 기본값은 45입니다.
-XX:NewRatio=n	신세대/구세대 크기의 비율입니다. 기본값은 2입니다.
-XX:생존자 비율=n	에덴/서바이버 공간 크기의 비율입니다. 기본값은 8입니다.
-XX:MaxTenuringThreshold=n	최대값입니다. 기본값은 15입니다.
-XX:ParallelGCThreads=n	가비지 수집기의 병렬 단계에서 사용되는 스레드 수를 설정합니다. 기본값은 JVM이 실행되는 플랫폼에 따라 다릅니다.
-XX:ConcGCThreads=n	동시 가비지 수집기가 사용할 스레드 수입니다. 기본값은 JVM이 실행되는 플랫폼에 따라 다릅니다.
-XX:G1ReservePercent=n	프로모션 실패 가능성을 줄이기 위해 잘못된 상한으로 예약되는 힙의 양을 설정합니다. 기본값은 10입니다.
-XX:G1HeapRegionSize=n	G1을 사용하면 Java 힙이 균일한 크기의 영역으로 세분화됩니다. 이것은 개별 하위 분할의 크기를 설정합니다. 이 매개변수의 기본값은 힙 크기에 따라 인체공학적으로 결정됩니다. 최소값은 1Mb이고 최대값은 32Mb입니다.

G1로 GC 로깅

G1로 GC 로깅

마지막으로 다루어야 할 주제는 로깅 정보를 사용하여 G1 수집기로 성능을 분석하는 것입니다. 이 섹션에서는 데이터를 수집하는 데 사용할 수 있는 스위치와 로그에 인쇄되는 정보에 대한 간략한 개요를 제공합니다.

로그 세부 정보 설정

세 가지 세부 수준으로 디테일을 설정할 수 있습니다.

(1) **`-verbosegc`**(**`-XX:+PrintGC`와** 동일)는 로그의 상세 수준을 *fine로* 설정합니다.

샘플 출력

```
[GC 일시정지(G1 거대 할당) (영) (초기 표시) 24M- >21M(64M), 0.2349730초]
[GC 일시 정지(G1 대피 일시 정지) (혼합) 66M->21M(236M), 0.1625268초]
```

(2) **`-XX:+PrintGCDetails`는** 세부 수준을 *더 세밀하게* 설정합니다. 이 옵션은 다음과 같은 정보를 표시합니다:

- 각 단계에 대해 평균, 최소 및 최대 시간이 표시됩니다.
- 루트 스캔, RSet 업데이트처리된 버퍼 정보 포함), RSet 스캔, 개체 복사, 종료(시도 횟수 포함).
- 또한 CSet 선택, 참조 처리, 참조 대기열 및 CSet 해제에 소요된 시간 등 '기타' 시간도 표시됩니다.
- 에덴, 생존자, 총 힙 점유량을 표시합니다.

샘플 출력

```
[확장 루트 스캔(ms): 평균: 1.7 분: 0.0 최대: 3.7 차이: 3.7]
[에덴: 818만(818만)->0억(714만) 생존자: 0억->104만 힙: 836만(4096만)->409만(4096만)]
```

(3) **`-XX:+UnlockExperimentalVMOptions -XX:`**G1LogLevel=finest는 세부 수준을 *가장 세밀하게* 설정합니다. 세밀함과 비슷하지만 개별 워커 스레드 정보를 포함합니다.

```
[외부 루트 스캔(ms): 2.1 2.4 2.0 0.0
                평균: 1.6분 0.0 최대: 2.4 차이: 2.3]
          [RS 업데이트(ms) 0.4 .2 0.4 .0
                평균: 0.2 분 0.0 최대: 0.4 차이: 0.4]
                [처리된 버퍼 : 5 1 10 0
                합계: 16, 평균: 4, 최소: 0, 최대: 10, 차이: 10]
```

시간 결정

몇 가지 스위치에 따라 GC 로그에 시간이 표시되는 방식이 결정됩니다.

(1) **`-XX:+PrintGCTimeStamps`** - JVM이 시작된 후 경과된 시간을 표시합니다.

샘플 출력

```
            1.729: [GC 일시정지(영) 46M->35M(1332M), 0.0310029초]
```

(2) `-XX:+PrintGCDateStamps` - 각 항목에 시간 접두사를 추가합니다.

```
2012-05-02T11:16:32.057+0200: [GC 일시정지(영) 46M->35M(1332M), 0.0317225초]
```

G1 로그 이해

이 섹션에서는 로그를 이해하기 위해 실제 GC 로그 출력을 사용하여 여러 용어를 정의합니다. 다음 예제에서는 로그의 출력과 함께 로그에서 찾을 수 있는 용어 및 값에 대한 설명을 보여줍니다.

참고: 자세한 내용은 G1 GC 로그에 대한 Poonam Bajaj의 블로그 게시물을 확인하세요.

G1 로깅 용어 색인

CT CSet 지우기
외부 루트 스캔 무료 CSet
GC 작업자 종료 GC 작업
자 기타 개체 복사 기타
평행 시간 참조 Eng
참조 프로시저
기억된 세트 스캔 종료 시간
기억된 세트 워커 시작 업데이트

평행 시간

```
414.557: [GC 일시 정지(영), 0.03039600 초] [병렬 시간: 22.9ms]
[GC 워커 시작(ms) 7096.0 .0 .1 7096.1 706.1 7096.1 7096.1 7096.1 7096.2
7096.2 .2 7096.2
          평균: 7096.1, 최소: 7096.0, 최대: 7096.2, 차이: 0.2]
```

병렬 시간 - 일시 정지의 주요 병렬 부분의 전체 경과 시간 워커 시작 - 워커가 시작되는 타임스탬프입니다.
참고: 로그는 스레드 ID를 기준으로 정렬되며 각 항목에서 일관성이 유지됩니다.

외부 루트 스캔

```
[익스트림 루트 스캔(ms): 3.1 3.4 3.4 3.0 4.2 2.0 3.6 3.2 3.4 7.7 3.7 4.4
       평균: 3.8, 최소: 2.0, 최대: 7.7, 차이: 5.7]
```

외부 루트 스캔 - 외부 루트(예: 힙을 가리키는 시스템 사전과 같은 것)를 스캔하는 데 걸리는 시간입니다.

기억된 세트 업데이트

```
[RS 업데이트(ms): 0.1 0.0 0.0 .0 0.0 0.0 0.0 0.0 .0 0.0 .0 평균: 0.0, 최소: 0.0,
최대: 0.1, 차이: 0.1]
              [처리된 버퍼 : 26 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
              합계: 26, 평균: 2, 최소: 0, 최대: 26, 차이: 26]
```

기억된 세트 업데이트 - 일시 정지가 시작되기 전에 동시 정제 스레드에서 완료되었지만 아직 처리되지 않은 버퍼는 모두 업데이트해야 합니다. 시간은 카드의 밀도에 따라 달라집니다. 카드가 많을수록 시간이 더 오래 걸립니다.

기억된 세트 스캔

```
[스캔 RS(ms): 0.4 0.2 0.1 0.3 0.0 .0 0.1 0.2 0.0 .1 0.0 0.0 .0 평균: 0.1, 최소: 0.0, 최대
0.4, Diff: 0.3]F
```

기억된 세트 스캔하기 - 컬렉션 세트를 가리키는 포인터를 찾습니다.

개체 복사

```
[오브젝트 복사(ms) 16.7 16.7 16.7 16.9 16.0 18.1 16.5 16.8 16.7 12.3 16.4 15.7 평균(%)
16.3, 최소: 12.3, 최대: 18.1, 차이: 5.8]
```

객체 복사 - 각 개별 스레드가 객체를 복사하고 비우는 데 소요한 시간입니다.

종료 시간

```
[종료(ms): 0.0 0.0 0.0 0.0 .0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 평균: 0.0, 최소: 0.0, 최대: 0.0, 차이: 0.0] [종료 시도 : 1 1 1 1 1 1 1 1 1 1 1
1 1 1 합계: 12, 평균: 1, 최소: 1, 최대: 1, 차이: 0]
```

종료 시간 - 작업자 스레드가 복사 및 스캔할 특정 개체 집합을 완료하면 종료 프로토콜로 들어갑니다. 훔칠 작업을 찾고 해당 작업이 완료되면 다시 종료 프로토콜로 들어갑니다. 종료 시도는 작업을 훔치려는 모든 시도를 계산합니다.

GC 작업자 끝

```
[GC 작업자 종료(ms) 7116.4 .3 7116.4 7116.3 7116.4 7116.3 71164 7116.4 7116.4
7116.4 7116.3 7116.3
      평균: 7116.4, 최소: 7116.3, 최대: 7116.4, 차이:                    0.1]
[GC 워커(ms) 20.4 20.3 20.3 20.2 20.3 20.2 20.2 20.2 .3 20.2 201 20.1
       평균: 20.2, 최소: 20.1, 최대: 20.4, 차이: 0.3]
```

GC 워커 종료 시간 - 개별 GC 워커가 종료되는 타임스탬프입니다. GC 워커 시간 - 개별 GC 워커 스레드에서 소

요된 시간입니다.

GC 작업자 기타

```
[GC 작업자 기타(ms): 2.6 2.6 2.7 2.7 2.7 2.7 2.7 28 2.8 2.8 2.8 2.8
      평균: 2.7, 최소: 2.6, 최대: 2.8, 차이: 0.2]
```

GC 워커 기타 - 앞서 나열된 워커 단계에 귀속시킬 수 없는 시간(각 GC 스레드에 대해)입니다. 상당히 낮아야 합니다. 과거에는 지나치게 높은 값을 보았으며, 이는 JVM의 다른 부분(예: 계층화된 코드 캐시 점유 증가)에서 병목 현상이 발생했기 때문이었습니다.

클리어 CT

```
[클리어 CT: 0.6ms]
```

RSet 스캔 메타데이터의 카드 테이블을 지우는 데 걸리는 시간

기타

```
[기타: 6.8ms]
```

GC 일시 중지의 다른 여러 순차 단계에 걸리는 시간입니다.

CSet

```
[CSet: 0.1ms 선택]
```

수집할 지역 세트를 완성하는 데 걸리는 시간입니다. 일반적으로 매우 짧지만 오래된 지역을 선택해야 하는 경우 약간 더 길어집니다.

참조 프로시저

```
[참조 프로시저: 4.4ms]
```

GC의 이전 단계에서 지연된 소프트, 약한 등의 참조를 처리하는 데 소요되는 시간입니다.

참조 문의

```
[참조 대기 시간: 0.1ms]
```

소프트, 약함 등의 참조를 보류 목록에 올리는 데 소요되는 시간입니다.

무료 CSet

```
[무료 CSet: 2.0ms]
```

기억된 세트를 포함하여 방금 수집한 지역 세트를 해제하는 데 소요되는 시간입니다.

Summary

요약

이번 OBE에서는 Java JVM에 포함된 G1 가비지 컬렉터에 대한 개요를 살펴봤습니다. 먼저 힙과 가비지 컬렉터가 Java JVM의 핵심 부분이라는 것을 배웠습니다. 다음으로 CMS 수집기와 G1 수집기를 사용하여 가비지 수집이 어떻게 작동하는지 검토했습니다. 다음으로 G1 명령줄 스위치와 이를 사용하기 위한 모범 사례에 대해 배웠습니다. 마지막으로 GC 로그에 포함된 개체 및 데이터 로깅에 대해 배웠습니다.

이 튜토리얼에서는 학습 내용을 살펴보았습니다:

- Java JVM의 구성 요소 G1 개요 CMS 수집기에
- 대한 리뷰
- G1 수집기 리뷰
  명령줄 스위치 및 모범 사례 G1을 사용한 로깅

리소스

자세한 정보 및 관련 정보는 다음 사이트와 링크를 참조하세요.

- Java 핫스팟 VM 옵션
- 가비지 퍼스트(G1) 쓰레기 수거업체 푸남 바자즈 G1 GC 블로그 포스

트

Java SE 7: 리치 클라이언트 애플리케이션 개발

- Java 성능 - 찰리 헌트와 비누 존
- 오라클 학습 라이브러리

크레딧

- 커리큘럼 개발자: 마이클 J 윌리엄스
- QA: 크리슈난자니 치타