Part 3. JVM, GC

2017년 12월 26일 화요일 오후 8:47

자바 가상 머신(Java Virtual Machine, JVM)

- 자바 어플리케이션을 클래스 로더를 통해 읽어 들여 자바 API와 함께 실행하는 것
- 가상 머신이란 프로그램을 실행하기 위해 물리적 머신과 유사한 머신을 소프트웨어로 구현할 것
- 자바 클래스로더(Java Classloader)는 자바 클래스를 자바 가상 머신(JVM)으로 동적 로드하는 자바 런타임 환경(JRE)의 일부이다.
- 자바는 동적으로 클래스를 읽어오며, 런타임에 모든 코드가 JVM에 링크된다. 모든 클래스는 그 클래스가 참조되는 순간 동적으로 JVM에게 링크되며, 메모리에 로딩된다.
- JVM은 JAVA와 OS 사이에서 중계자 역할을 수행하며 JAVA가 OS에 구애받지 않고 재사용이 가능하다.
- 메모리 관리와 Garbage Collection을 수행한다.
- JVM은 스택 기반의 가상머신이다.

자바 가상 머신을 알아야 하는 이유

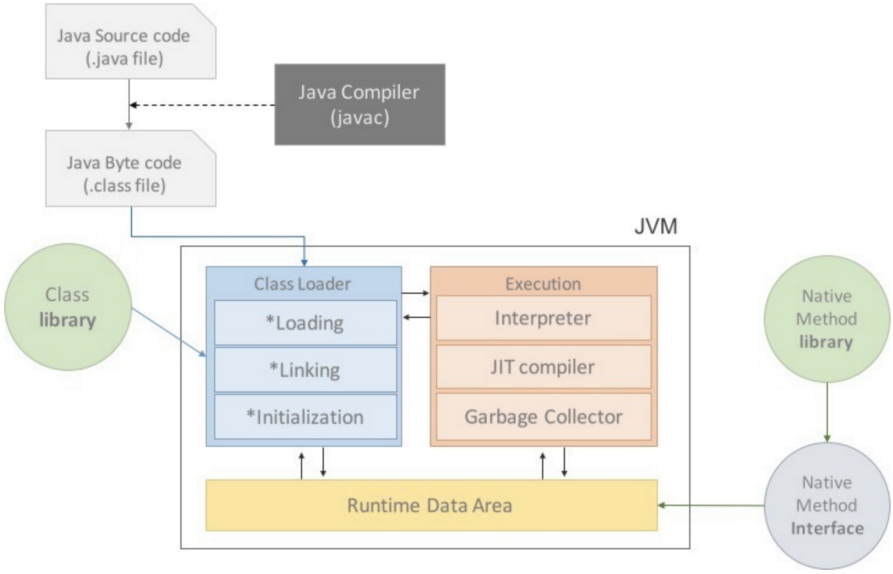
- 동일한 프로그램이라도 성능은 매우 중요
- 한정된 메모리를 효율적으로 사용하여 최고의 성능을 내기 위해서
- 메모리 효율성을 위해 메모리 구조에 대한 이해 필요
- 메모리 관리를 통한 속도 저하 현상 또는 튕김 현상 방지 등의 효율성 증대를 위해

자바 프로그램 실행 과정

- 프로그램이 실행되면 JVM은 OS로부터 이 프로그램이 필요로 하는 메모리를 할당받는다.
  JVM이 메모리를 용도에 따라 여러 영역으로 나누어 관리한다.
- 자바 컴파일러(Javac)가 자바 소스코드(.java)를 읽어들여 자바 바이트코드(.class)로 변환시킨다.
- Class Loader를 통해 class파일을 JVM으로 로딩한다.
- 로딩된 class파일은 Execution Engine을 통해 해석된다.
- 해석된 바이트코드는 Runtime Data Areas에 배치되어 실질적인 수행이 이루어진다.
- 이러한 과정 속에서 JVM은 필요에 따라 Thread Synchronization과 GC 같은 관리 작업을 수행한다.

JVM>>

-



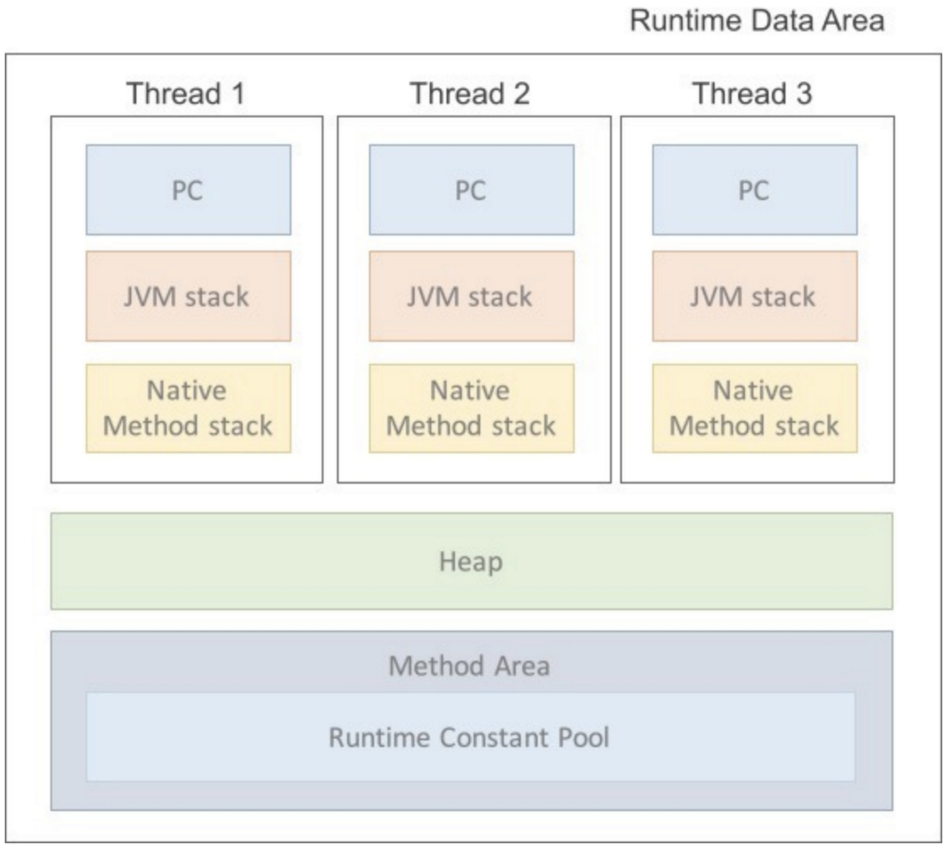
JVM 구성 요소

- Class Loader (클래스 로더)
  - JVM 내로 클래스(.class 파일)를 로드하고, 링크를 통해 배치하는 작업을 수행하는 모듈
  - Runtime 시, 동적으로 클래스 로드
  - jar 파일 내에 저장된 클래스들을 JVM 위에 탑재하고 사용하지 않는 클래스들은 메모리에서 삭제
  - 클래스를 처음 참조할 때, 해당 클래스를 로등하고 링크한다.
- Execution Engine (실행 엔진)
  - 클래스를 실행시키는 역할
  - 클래스 로더가 JVM 내의 런타임 데이터 영역에 바이트 코드를 배치시키면, 이를 실행 엔진이 실행시킨다.
  - 바이트 코드는 기계가 바로 수행할 수 없는 비교적 인간이 보기 편한 형태이다. 실행 엔진은 이를 실제 JVM 내부에서 기계가 실행할 수 있는 형태로 변경한다. JVM은 이를 두 가지 방식으로 변경한다.
    - Interpreter(인터프리터): 실행 엔진이 자바 바이트 코드를 명령어 단위로 읽어서 실행한다.
    - JIT(Just-In-Time): 인터프리터 방식의 단점을 보완하기 위해 도입된 JIT 컴파일러이다. 인터파ㅡ리터 방식으로 실행하다가 적절한 시점에서 바이트코드 전체를 컴파일하여 네이티브 코드로 변경하여 직접 실행하는 방식이다. 네이티브 코드는 캐시에 보관되기 때문에 한 번 컴파일된 코드는 빠르게 수행된다.
    - JIT 컴파일어가 컴파일하는 과정은 바이트코드를 인터프리팅하는 것보다 오래 걸리기 때문에 한 번만 실행되는 코드라면 컴파일하지 않고 인터프리팅하는 것이 유리한다. 따라서 JIT 컴파일러를 사용하는 JVM은 내부적으로 해당 메서드가 얼마나 자주 수해되는지 체크하며, 일정 정도를 넘을 경우 컴파일을 수행한다.

- Garbage Collector
    - GC를 수행하는 모듈(쓰레드)

Runtime Data Area

- 프로그램을 수행하기 위해 OS에서 할당받은 메모리 공간



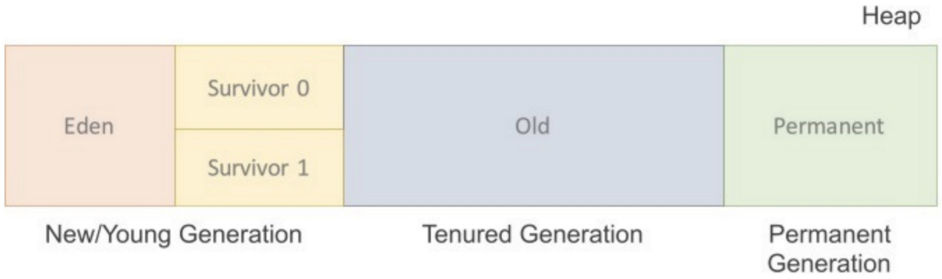
- PC Register
    - Thread가 시작될 때 생성되며, 각 스레드마다 하나씩 존재한다.
    - Thread가 어던 부분에 어떤 명령으로 실행해야 할 지에 대한 기록을 담당하며, 현재 수행중인 JVM 명령의 주소를 갖는다.

- JVM Stack (JVM 스택 영역)
    - 프로그램 실행 과정에서 임시로 할당되었다가 메소드를 빠져나가면 바로 소멸되는 특성의 데이터를 저장하기 위한 역역
    - 각종 형태의 변수, 임시 데이터, 스레드, 메소드 정보등을 저장한다.
    - 메소드 호출 시 각 스택 프레임(메서드만의 공간)이 생성되며 메서드 수행이 끝나면 프레임 단위로 삭제된다.
    - 메서드 안에서 사용되는 값(Local Variable)을 저장한다.
    - 호출된 메서드의 매개 변수, 지역 변수, 반혼 값 및 연산 시 일어나는 값을 임시로 저장한다.

- Native Method Stack
    - 자바 프로그램이 컴파일되어 생성되는 바이트 코드가 아닌 실제 실행할 수 있는 기계어로 작성된 프로그램을 실행시키는 영역
    - Java Native Interface를 통해 바이트 코드로 전환하여 저장된다.

    - 일반 프로그램처럼 커널이 스택을 잡아 독자적으로 프로그램을 실행시키는 영역

- Method Area (Class Area, Static Area)
    - 클래스 정보를 처음 메모리 공간에 올릴 때 초기화 되는 대상을 저장하기 위한 메모리 공간
    - 컴파일된 바이트코드의 대부분이 메소드 바이트코드이기 때문에 거의 모든 바이트코드가 올라간다.
    - 내부에 Runtime Constant Pool이라는 별도의 관리 영역이 존재하며, 이는 상수 자료형을 저장하며 참조하는 중복을 막는 역할을 수행한다.
    - 올라가는 정보의 종류는 세 가지로 나뉜다.
        - Field Information: 멤버 변수의 이름, 데이터 타입, 접근 제어자에 대한 정보
        - Method Information: 메소드의 이름, 리턴 타입, 매개 변수, 접근 제어자에 대한 정보
        - Type Information: class / interface 구분 저장, Type 속성, 전체 이름, super class의 정보
    - Method Area는 클래스를 위한 공간이며, Heap 영역은 객체를 위한 공간이다.

- Heap Area
    - 객체를 저장하는 가상 메모리 공간
    - new 연사자로 생성된 객체와 배열이 지정되며, Class Area 영역에서 올라온 클래스들만 객체로 생성할 수 있다.
    - 힙은 세 부분으로 나뉜다.
    - 

    - Permanent Generation
        - 생성된 객체 정보의 주소 값이 저장되는 공간
        - Class Loader에 의해 로드되는 Class, Method 등의 Meta 정보가 저장되는 공간으로, JVM에 의해 사용된다.
        - Reflection을 사용하여 동적으로 클래스가 로딩되는 경우에 사용된다.

    - New/Young Generation
        - Eden: 객체가 최초로 생성되는 공간
        - Survivor 0/1: Eden에서 참조되는 객체들이 저장되는 공간

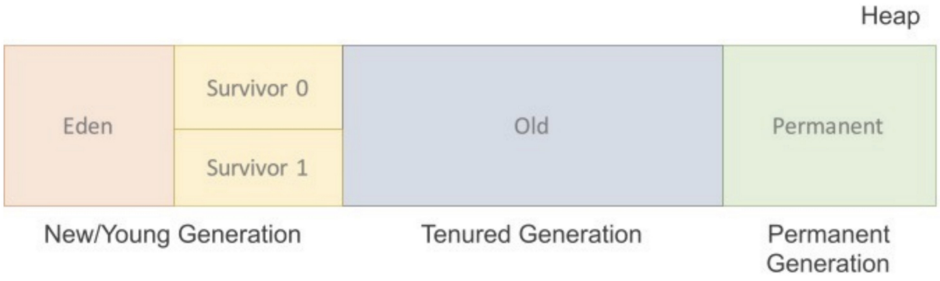
    - Tenured Generation (Old)
        - New Area에서 일정 시간 참조되고 있는, 살아남은 객체들이 저장되는 공간.
        - Eden 영역에 객체가 가득차게 되면 첫 번째 GC(Minor GC)가 발생한다.

Eden 영역에 있는 값을 Survivor 1영역에 복사하고 이 영역을 제외한 나머지 영역의 객체를 삭제한다.

- 인스턴스는 소멸 방법과 소멸 시점이 지역 변수와는 다르기 때문에 힙이라는 별도의 영역에 할당된다. 자바 가상머신은 합리적으로 인스턴스를 소멸시키며, 더 이상 인스턴스의 존재 이유가 없을 경우 소멸시킨다.

가비지 컬렉션(Garbage Collection)

-



- Minor GC
  - 새로 생선된 객체(Instance)는 Eden 영역에 위치하며 이 영역에서 GC가 하번 발생한 후 살아남은 객체를 Survivor 영역 중 하나로 이동한다.
  - 이 과정을 반복하다가 계속해서 살아남아 있는 객체는 일정 시간 참조디고 있다는 것을 의미함으로 Old 영역으로 이동시킨다.

- Major GC
  - Old 영역에 있는 모든 객체들을 검사해서 참조되지 않은객체들을 한꺼번에 삭제한다.
  - 시간이 오래 걸리고 실행 중인 프로세스가 정지되며, 이를 "Stop The World"라 지칭한다.
  - Major GC가 발생하면 GC를 실행하는 스레드를 제외한 나머지 스레드는 모두 작업을 멈춘다. GC 작업이 완료한 이후에야 중단됐던 작업을 다시 시작한다.

가비지 컬렉션의 소멸 대상 선정 원리

- Garbage Collector는 힙 내의 객체 중에서 가비지(Garbage)를 찾아내고 찾아낸 가비지를 처리해서 힙의 메모리를 회수한다.
- 참조되고 있지 않은 객체(Instance)를 가비지라 하며 객체가 가비지인지에 대한 유무를 판단하기 위해 reachability라는 개념을 사용한다.
- 어떤 힙 영역에 할당된 객체가 유효한 참조가 있으면 reachability, 없다면 unreachability로 판단한다.
- 하나의 객체는 다른 객체를 참조하고, 다른 객체는 또 다른 객체를 참조할 수 있기 때문에 참조 사슬이 형성되는데. 이 참조 사실 중 최초에 참조한 것을 Root Set이라 지칭한다.
- 힙 영역에 있는 객체들은 총 4 가지 경우에 대한 참조를 하게 된다.
  - 1: 힙 내의 다른 객체에 의한 참조

    - 2: Java 스택, Java 메서드 실행 시에 사용하는 지역 변수와 파라미터들에 의한 참조
    - 3: 네이티브 스택(JNI, Java Native Interface)에 의해 생성된 객체 참조
    - 4: 메서드 영역의 정적 변수 참조
    - 2, 3, 4를 Root Set이라 하며, 참조 사실 중 최초에 참조한 것을 의미한다.
- 비번한 가비지 컬렉션의 실행은 시스템에 부담이 될 수 있기 때문에 성능에 영향을 미치지 않도록 가비지 컬렉션 실행 타이밍은 별도의 알고리즘을 기반으로 계산된다.