자바 메모리 구조 보고서

-JVM의 메모리 구조에 관하여-

선린 인터넷 고등학교

소프트웨어과

10514

나정휘

목차

1. 도입 & 메모리

2. 자바 프로그램의 실행 구조

가. JVM이란?

나. JVM 구조

3. 메모리 영역

가. Class Area

A. Runtime Constant Pool

나. Stack Area

다. Heap Area

라. Native Method Stack Area

마. PC Register

4. 가비지 컬렉션

가. Minor Garbage Collection

나. Major Garbage Collection

다. GC 알고리즘

A. Serial Collector

B. Parallel Collector

C. Parallel Compacting Collector

5. 결론

**1. 도입 & 메모리**

프로그램을 실행하기 위한, 또는 실행하는 도중의 데이터, 명령어 등을 저장하는 공간이다.

프로그램의 성능을 올리기 위해서는 공간 복잡도(공간을 얼마나 차지하는지)는 시간 복잡도(계산 시간을 얼마나 소요하는지)와 더불어 가장 중요하게 고려해야 하는 부분이기 때문에 메모리의 구조와 메모리 사용량, 즉 공간 복잡도를 비롯한 복잡도 이론을 공부해야 한다. 또한, 메모리를 관리하지 않으면 속도가 떨어지거나 프로그램이 강제 종료가 될 수도 있기 때문에 중요도는 더욱 올라간다. 그래서 이 보고서에서는 메모리 구조와 쓰레기 객체를 다루는 기술을 알아볼 것이다.

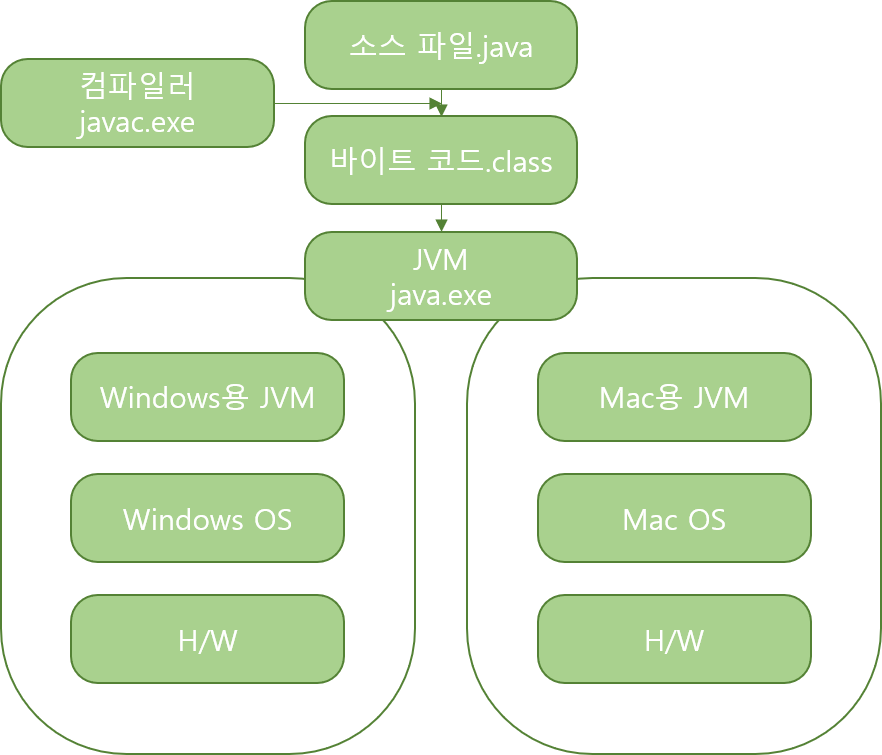
**2. 자바 프로그램의 실행 구조**

**가. JVM이란?**

자바 프로그램은 완전한 기계어가 아니기 때문에 운영 체제에서 바로 실행할 수 없다. 대신 기계어의 전 단계라고 볼 수 있는 바이트 코드로 구성되어 있기 때문에 이것을 해석하고 실행할 수 있는 가상의 운영 체제인 JVM(Java Virtual Machine, 자바 가상 머신)에서 실행할 수 있다. 대부분의 프로그램들은 운영 체제가 제어하고 있는 시스템의 리소스 중 일부인 메모리를 제어해야 하는데, 이 때문에 OS에 종속되어 있다. 하지만 자바 프로그램은 운영 체제마다 JVM이 구현되어 있고, JVM을 설치만 한다면 동일한 프로그램으로 다른 운영 체제에서도 실행이 가능하다. 다만, 운영 체제에 직접적으로 접근하는 것이 아닌 JVM을 거쳐서 접근하므로 속도는 약간 느려질 수 있다.

**나. JVM의 구조**

바이트 코드는 JVM환경에서는 동일한 결과를 보장하며 JVM에 종속되어 있고, JVM은 운영 체제에 종속되어 있다. 바이트 코드는 운영 체제가 이해할 수 있는 기계어로 바뀌어야 하기 때문에 운영 체제에 맞는 JVM을 설치해야 한다.



자바 프로그램을 제작할 때에는 먼저 확장자가 .java인 소스 파일을 작성을 한다. 이 파일을 컴파일러(javac.exe)로 컴파일을 하면 확장자가 .class인 바이트 코드 파일이 된다. 바이트 코드는 java.exe에 의해 JVM에서 해석되고 기계어로 번역되어 실행된다.

Write once, run anywhere.

자바의 장점인 이 문구는 자바의 매우 매력적인 점이다. 한 번만 코드를 작성을 하면 JVM을 통해서 어디에서도 실행이 가능하다. 하지만 기계어로 번역되어 실행이 되기 때문에 C/C++와 같이 컴파일 단계에서 기계어로 번역되는 언어보다는 느리다. 그러나 JVM 내부에 JIT컴파일러를 통해 최적화를 수행하여 속도의 격차는 많이 줄어들고 있다.

**3. 메모리 영역**

**가. Class Area**

Class Area은 Method Area, Code Area, Static Area로 불리기도 한다. Class Area에는 바이트 코드가 로드 된다. JVM에서 프로그램을 실행하기 위해서는 먼저 바이트 코드들이 메모리에 올라와야 한다. 그 바이트 코드를 Class Area에 저장한다.

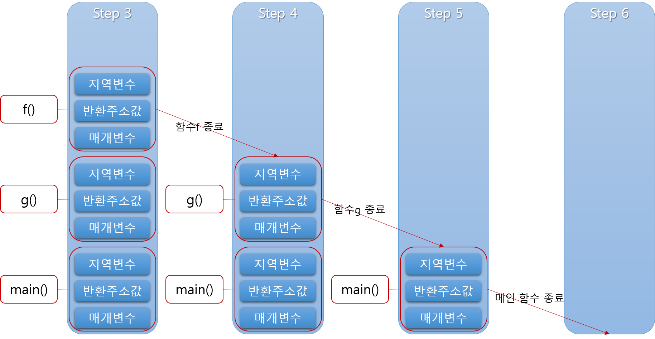
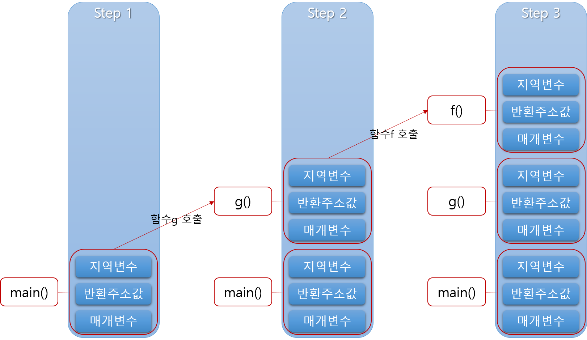
Field Information(멤버 변수의 이름, 타입, 접근 제어자), Method Information(메소드의 이름, 리턴 타입, 매개변수, 접근 제어자), Type Information(Type의 속성이 class인지 interface인지), Static 변수 등이 저장된다.

**A. Runtime Constant Pool**

Runtime Constant Pool은 Class Area에 포함되지만, 독자적인 중요성이 있다.

이름에 나와있듯이 final 변수, 클래스/인터페이스 상수와 같은 상수를 저장한다. 또한 메소드와 필드에 대한 모든 레퍼런스를 저장하고 있어, JVM은 Runtime Constant Pool을 통해 해당 메소드나 필드의 실제 메모리 상 주소를 찾아간다.

**나. Stack Area**



C언어 메모리 구조에서의 스택 영역과 동일하다. 메소드가 호출될 때마다 스택 프레임이 생성이 된다. 스택 프레임은 호출된 메소드 만을 위한 공간이며, 메소드 안에서 사용되는 값, 매개변수, 지역변수, 리턴 값과 같은 메소드 안에서 작용하는 값들을 저장한다. 메소드의 수행이 끝나면 프레임이 삭제된다.

아래 코드는 스택 영역의 예제 코드이다.

**import** **java.util.\***;

**import** **java.lang.\***;

**import** **java.io.\***;

**public** **class** **StackArea** {

**public** **static** void func1(int a) {

func2(++a);

System.out.printf("func1() : %d\n", a);

}

**public** **static** void func2(int a) {

func3(++a);

System.out.printf("func2() : %d\n", a);

}

**public** **static** void func3(int a) {

++a;

System.out.printf("func3() : %d\n", a);

}

**public** **static** void main (String [] args) {

int a = 0;

func1(a);

System.out.printf("main () : %d\n", a);

}

}

*/\**

*[[output]]*

*func3() : 3*

*func2() : 2*

*func1() : 1*

*main () : 0*

*\*/*

스택 구조는 먼저 들어간 것이 나중에 나오기 때문에 이런 현상이 발생한다.

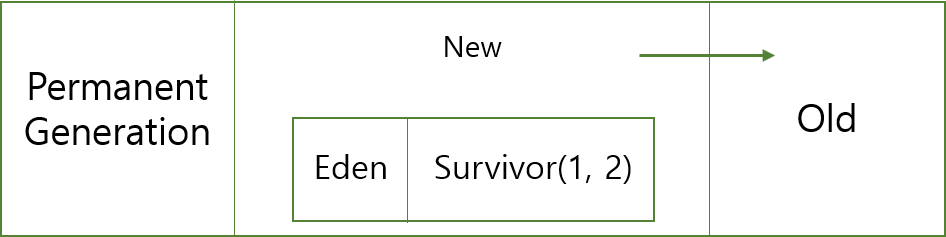
코드의 실행 흐름을 조금 더 자세히 써보면 이렇게 된다.

|  |  |  |
| --- | --- | --- |
| **실행 코드** | **스택 메모리** | **출력** |
| main () {  int a = 0;  func1(a);  //something  } | main () : a= 0 |  |
| func1(a) {  func2(++a);  //something  } | func1() : a = 1  main () : a = 0 |  |
| func2(a) {  func3(++a);  //something  } | func2() : a = 2  func1() : a = 1  main () : a = 0 |  |
| func3(a) {  ++a;  //something  } | func3() : a = 3  func2() : a = 2  func1() : a = 1  main () : a = 0 |  |
| func3(a) {  //something  print(a)  } | func3() : a = 3  func2() : a = 2  func1() : a = 1  main () : a = 0 | func3() : 3 |
| func2(a) {  //something  print(a)  } | func2() : a = 2  func1() : a = 1  main () : a = 0 | func3() : 3  func2() : 2 |
| func1(a) {  //something  print(a)  } | func1() : a = 1  main () : a = 0 | func3() : 3  func2() : 2  func1() : 1 |
| main(a) {  //something  print(a)  } | main () : a = 0 | func3() : 3  func2() : 2  func1() : 1  main () : 0 |

**다. Heap Area**

동적으로 생성된, 즉 new 연산자로 생성된 객체와 배열을 저장하는 공간이다. Stack Area와는 다르게 사용이 끝나도 사라지지 않는다. 지워지는 경우는 두 가지가 있다. 주소를 잃어버려 Garbage가 되어 Garbage Collector에 의해 지워지거나 JVM이 종료될 때 지워진다.

힙 영역을 그림으로 나타내면 다음과 같다.



permanent generation에는 생성된 객체들의 정보가 기록되어 있는 주소가 저장되어 있다.

New Area는 Eden과 Survivor(1, 2), 두 가지로 나누어져 있다. Eden은 객체들이 최초로 생성되는 공간이고, Survivor은 Eden에서 참조되는 객체들이 저장되는 공간이다.

Old Area에는 New Area에서 일정시간 이상 참조되고 있는 객체들이 저장된다.

아래 코드는 힙 영역의 예제 코드이다

**import** **java.util.\***;

**import** **java.lang.\***;

**import** **java.io.\***;

**class** **HeapArea** {

**public** **static** int [] m1(int a) {

int [] arr = m2(a+1);

arr [2] = a;

**return** arr;

}

**public** **static** int [] m2(int a) {

int [] arr = m3(a+1);

arr [1] = a;

**return** arr;

}

**public** **static** int [] m3(int a) {

int [] arr = **new** int [3];

arr [0] = a;

**return** arr;

}

**public** **static** void main (String [] args) {

int [] arr = m1(100);

**for** (int i=0; i<arr. length; i++) {

System.out.printf("%d = %d\n", i, arr[i]);

}

}

}

*/\**

*[[output]]*

*0 = 102*

*1 = 101*

*2 = 100*

*\*/*

실행 흐름은 다음과 같다.

|  |  |  |  |
| --- | --- | --- | --- |
| **실행 코드** | **스택 메모리** | **힙 메모리** | **출력** |
| main () {  //something  } | main () |  |  |
| main () {  int [] arr = m1(100)  //something  } | m1() : a = 100  main () |  |  |
| m1(a) {  int [] arr = m2(a+1)  //something  } | m2() : a = 101  m1() : a = 100  main () |  |  |
| m2(a) {  int [] arr = m3(a+1)  //something  } | m3() : a = 102  m2() : a = 101  m1() : a = 100  main () |  |  |
| m3(a) {  int [] arr = new int [3];  //something  } | m3() : a = 102, arr = 7000번지  m2() : a = 101  m1() : a = 100  main () | 7000번지  arr [0] :  arr [1] :  arr [2] : |  |
| m3(a) {  //something  arr [0] = a;  //something  } | m3() : a = 102, arr = 7000번지  m2() : a = 101  m1() : a = 100  main () | 7000번지  arr [0] : 102  arr [1] :  arr [2] : |  |
| m2(a) {  int [] arr = m3(a+1);  //something  } | m2() : a = 101, arr = 7000번지  m1() : a = 100  main () | 7000번지  arr [0] : 102  arr [1] :  arr [2] : |  |
| m2(a) {  //something  arr [1] = a;  //something  } | m2() : a = 101, arr = 7000번지  m1() : a = 100  main () | 7000번지  arr [0] : 102  arr [1] : 101  arr [2] : |  |
| m1(a) {  int [] arr = m2(a+1);  //something  } | m1() : a = 100, arr = 7000번지  main () | 7000번지  arr [0] : 102  arr [1] : 101  arr [2] : |  |
| m1(a) {  //something  arr [2] = a;  //something  } | m1() : a = 100, arr = 7000번지  main () | 7000번지  arr [0] : 102  arr [1] : 101  arr [2] : 100 |  |
| main () {  int [] arr = m1(100);  //something  } | main(), arr = 7000번지 | 7000번지  arr [0] : 102  arr [1] : 101  arr [2] : 100 |  |
| 출력 | main(), arr = 7000번지 | 7000번지  arr [0] : 102  arr [1] : 101  arr [2] : 100 | 0 = 102  1 = 101  2 = 100 |

**라. Native Method Stack Area**

자바 외의 다른 언어에서 제공하는 메소드들이 저장되는 공간이고, 각 스레드마다 하나씩 생성된다. 스레드가 Native Method를 호출하면 Java Stack Area를 벗어나게 되며, 이 때 Native Method에 의해 Native Method Stack Area를 사용하게 된다. Native Method에서 다시 Java Method를 호출하면 Java Stack Area로 돌아간다.

\* Native Method는 C/C++같은 언어로 구현된 메소드이며, 주로 하드웨어 접근용 메소드이다.

**마. PC Register**

스레드가 생성될 때마다 생성되는 공간이며, 스레드가 어떤 부분을 어떤 명령으로 실행할 지에 대해 기록한다. 현재 스레드에서 실행할 명령의 주소를 가지고 있다.

**4. 가비지 컬렉션**

자바에서는 메모리를 GC(Garbage Collector)라는 알고리즘을 통하여 관리하기 때문에, 개발자가 메모리를 처리하기 위한 로직을 만들 필요가 없고, 절대로 만들어서는 안된다.

자바에서는 쓰레기도 객체이다. 하나의 객체는 메모리를 차지하고, 필요하지 않으면 메모리에서 해제되어야 한다. 쓰레기를 메모리에서 효과적으로 제거해주는 작업을 GC라고 한다.

**가. Minor Garbage Collection**

New영역에서 발생하는 GC를 의미합니다.

**나. Major Garbage Collection**

Old와 Perm영역에서 발생하는 GC를 의미합니다.

**다. GC 알고리즘**

**A. Serial Collector (시리얼 콜렉터)**

New영역과 Old영역이 연속적으로 처리되며, 하나의 CPU를 사용한다. Stop-the-World라고 표현한다. 다시 말해, 콜렉터가 수행될 때 프로그램의 수행이 정지된다.

작동 방식은,

1. 일단 살아있는 객체는 Eden영역에 넣는다.

2. Eden영역이 꽉 차면 To Survivor영역으로 살아 있는 객체가 이동한다. 이 때 Survivor영역으로 가기에 너무 큰 객체는 바로 Old영역으로 이동한다. 그리고, From Survivor영역에 있는 살아 있는 객체는 To Survivor영역으로 이동한다.

3. To Survivor영역이 꽉 찼을 경우, Eden영역이나 From Survivor영역에 남아있는 객체들을 Old영역으로 이동한다.

4. 이 이후에 Old나 Perm영역에 있는 객체들은 Mark-Sweep-Compact Collection 알고리즘을 따른다. 이 알고리즘은 다음 과정을 거쳐 진행된다.

1. Old영역으로 이동된 객체 중 살아있는 객체를 식별한다. (Mark)

2. Old영역의 객체들을 훑는 작업을 수행하여 쓰레기 객체를 식별한다. (Sweep)

3. 필요 없는 객체들을 지우고 살아 있는 객체들을 한 곳으로 모은다. (Compact)

**B. Parallel Collector (병렬 콜렉터)**

이 방식은 throughput collector로도 잘 알려진 방식이다. 이 방식의 목표는 다른 CPU가 대기 상태로 남아있는 것을 최소화하는 것이다. 시리얼 콜렉터와 달리 New영역에서의 컬렉션을 병렬로 처리한다. 많은 CPU를 사용하기 때문에 GC의 부하를 줄이고 프로그램의 처리량을 증가시킬 수 있다.

Old영역의 GC는 시리얼 콜렉터와 마찬가지로 Mark-Sweep-Compact Collection 알고리즘을 사용한다.

**C. Parallel Compacting Collector (병렬 컴팩팅 콜렉터)**

병렬 콜렉터와 다른 점은 Old영역 GC에서 새로운 알고리즘을 사용한다는 점이다. 그러므로 New영역에 대한 GC는 병렬 콜렉터와 동일하지만, Old영역의 GC는 다음의 3단계를 거친다.

1. Mark단계 : 살아있는 객체를 식별해서 표시하는 단계

2. Sweep단계 : 이전에 GC를 수행하여 컴팩션된 영역에 살아 있는 객체의 위치를 조사하는 단계

3. Compact단계 : 컴팩션을 수행하는 단계. 수행 이후에는 컴팩션된 영역과 비어 있는 영역으로 나뉜다.

**5. 결론**

평소에 알고리즘 공부를 하면서 공간 복잡도를 계산하는 일은 있었지만, 메모리 구조에 대해 이렇게 자세히 알아본 적도, 자동으로 실행되는 Garbage Collecting의 원리도 알아 본적이 없었지만 이번 보고서를 작성하면서 알게 되었다. 앞으로도 여러 분야에 대해 보고서를 작성하면서 넓은 지식을 갖고 싶다.

참고 문서

http://huelet.tistory.com/entry/JVM-%EB%A9%94%EB%AA%A8%EB%A6%AC%EA%B5%AC%EC%A1%B0

http://wanzargen.tistory.com/16

http://hoonmaro.tistory.com/19

http://wanzargen.tistory.com/17

http://haneulnoon.tistory.com/23

http://12bme.tistory.com/57