java System 구조

- Java 의 Runtime 시 동작하는 시스템의 구조에 대해 이해하고, 시스템 구조로 인하여 발생 가능한 이슈들에 대해 파악할수 있게 매커니즘의 이해 , 그리고, 발생하는 이슈에 대하여 트러블 슈팅 방법에 대해 학습한다.

1. java 의 특성

    - java 는 실행시 jvm (java virtual machine) 상에서 기동이 된다.
    - jvm 은 host machine(Window, linux, unix등) 과 상관 없이 동작 할수 있게 해주어서 platform independent하게 java 가 동작 할수 있도록 해준다.
    - java의 개발은 .java 파일로 개발하며, 컴파일시 .class 파일 (byte code)이 생성되어 jvm 상에서 동작 된다.
    - java 는 maven, gradle 과 같은 dependency 관리 툴을 사용한다.
    - java 의 메모리는 jvm 의 GC(Garvage Collector)에 의해 관리 된다.

2. java runtime 환경

    1. 사용자 소스의 main 구문이 실행 된다.

        - main 은 Static 으로 선언되어 instance화 없이 수행이 가능 하다.

    2. 수행에 필요한 dependency는 pom파일이나, gradle 파일로 관리되며, runtime시 class path에 관련 dependent 한 jar 파일들이 추가되어 application class loader에 의해 로딩된다.

    3. class loading은 계층적으로 수행이 되며, java jre의 기본 클래스 -> 응용 시스템 클래스 순으로 로딩이 되며, was 의 경우는 jre 기본 -> was classloader -> web application classloader 순으로 수행이 되어, ap에서 필요한 class 를 was classloader에서 먼저 있는 경우는 ap가 원하는 class 대신 was 의 class 가 먼저 loading되어 문제가 발생 할수 있다.

    4. class들은 perm 영역의 method Area에 적재된다.

    5. 수행중 new 명령어로 instance 화 되면, heap 영역에 메모리가 생성이 되어 수행이 된다.

    6. heap 영역에 생성된 메모리는 gc 설정에 따라 young 영역에 저장되며, young영역에서 일정 thredhold 이상까지 존재하고 있으면, old 영역으로 위치를 이동한다.

    7. old 영역에서 메모리가 많이 사용될때 gc 에 의해 full gc 가 수행이 되며, full gc는 메모리 확보를 위해 ap code의 수행을 멈추고 메모리 확보 까지 gc를 수행하게 된다.

3. java memory 관리

    - java 는 instance 화 된 메모리를 heap 으로 관리하며, heap 에 생성된 메모리의 생성/회수는 gc를 통해서 관리된다.
    - gc가 자동으로 관리되는 만큼 gc의 설정에 따라 시스템이 full gc 로 인해 시스템 중단이 발생 할수 있다.
    - java gc 설정은 용량, old, new, eden survivor, perm, gc 종류 설정 등이 있으며, 이는 gc 매커니즘을 이해하고, 시스템의 메모리 사용에 따라 적절히 설정 해야 한다.
    - java gc 설정에는 추후 문제 분석을 위한 log 설정 및 heapdump 생성 옵션등도 있다.

4. 발생할수 있는 이슈

    1. class loading 이슈

1. ap 개발자 들은 ap의 pom 파일의 내용을 가지고 개발을 하기에 실제 runtime시 실행되는 class loader의 차이에 따라 예상치 못한 이슈가 발생 할수 있다.
    - 예 : jboss was를 사용하고 있으며, ap에서는

```
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-web</artifactId>
        <version>6.0.2</version>
    </dependency>
```

    을 참조하여 개발을 했으나 jboss was의 class path에 다른버젼이 존재 할 경우에는 class loading순서에 따라 jboss 의 jar파일이 로딩되고, ap의 jar파일의 class는 무시된다.

2. Memory leak 이슈
    1. ap에서 memory를 과다하게 사용할 경우 oom(Out of memory)이 발생하게 된다.
        1. 일반적으로 spring을 통해 개발을 할 경우는 dependency injection에 따라 singleton registry로 동작을 하기 때문에 jvm 안에 하나의 instance된 클래스로 동작을 하게 되어 중복 사용이 없다.
    2. Sample ap

```
import java.util.List;

public class MemoryLeakExample {

static List<byte[]> leak = new ArrayList<>();

public static void main(String[] args) {
    while (true) {
    byte[] bytes = new byte[1048576];
    leak.add(bytes);
    }
}
}
```

        1. 샘플 프로그램을 구동할 경우 java 에서 설정한 heap memory를 모두 다 써버리고 oome 가 발생하여 ap 가 중단된다.
        2. Ap 구동시 vm option 으로 heap memory dump를 위한 설정을 해준다.

```
-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/Users/mac
```

        3. 구동시 ap 가 중지되면서 heap dump파일을 생성한다.

```

```

```
-rw-------    1 mac  staff  2149070181  3 15 11:23
java_pid11076.hprof
```

4. 생성된 heap dump파일을 mat(memory analyze tool)을 통해 분석리포트를 본다.

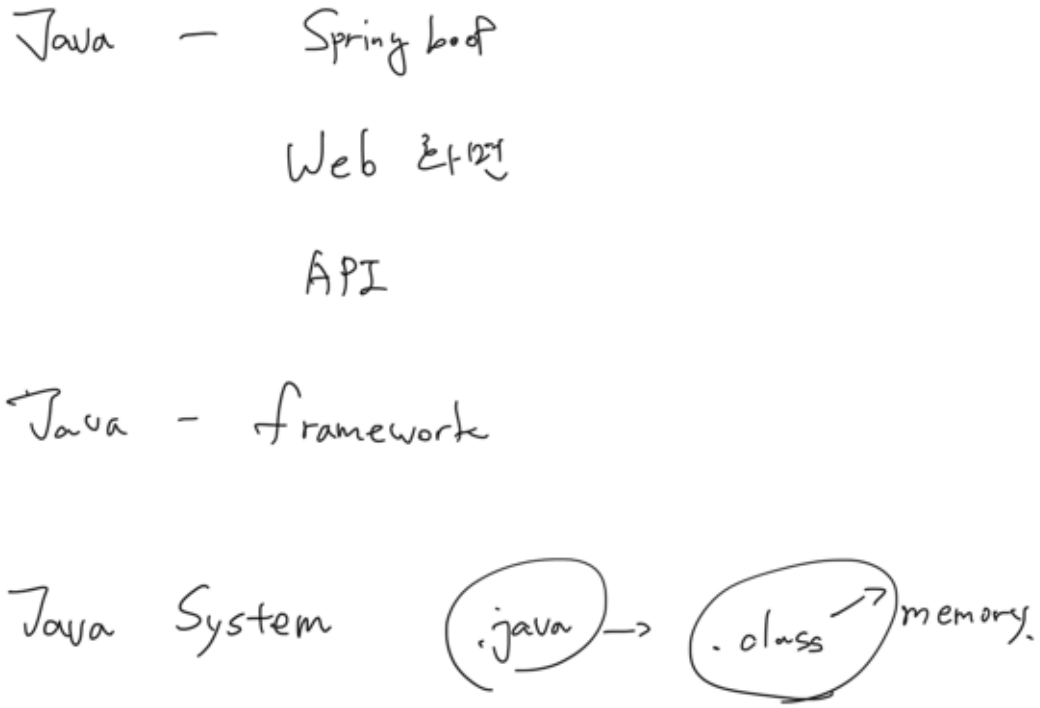
```
The class com.example.demo.MemoryLeakExample, loaded by
jdk.internal.loader.ClassLoaders$AppClassLoader @
0x70821c2f8, occupies 2,144,381,792 (99.91%) bytes. The
memory is accumulated in one instance of java.lang.Object[],
loaded by <system class loader>, which occupies
2,144,381,760 (99.91%) bytes.

Keywords

com.example.demo.MemoryLeakExample
jdk.internal.loader.ClassLoaders$AppClassLoader @
0x70821c2f8
java.lang.Object[]
```

분석 내용으로 'com.example.demo.MemoryLeakExample' 에서 20g 분량의 heap을 사용하고 있음을 확인 할수 있다.

3. Deadlock 이슈

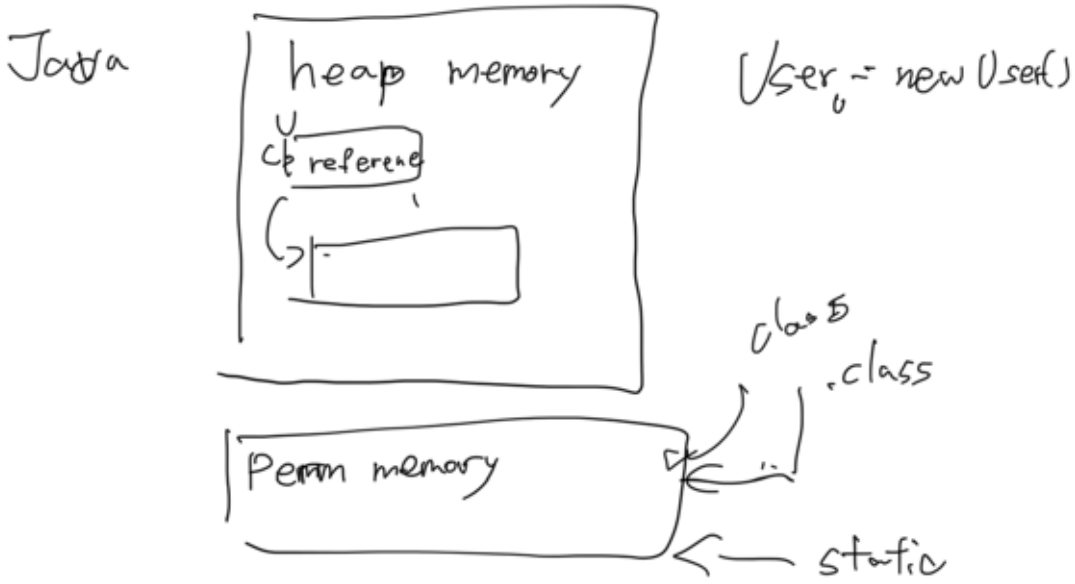
- to be continue

강의 스케치



→ JVM → Memory

Ap
↓
JVM
↓
Kernel System call



com.ktds.User

→ 변수생성 수 있는 이유

→ jvm code → 어떻게 동작

demo

Spring 3.04 → 2.01

org.---.SpringBootApplication

pom.

dependent maven

groovy

node dependent

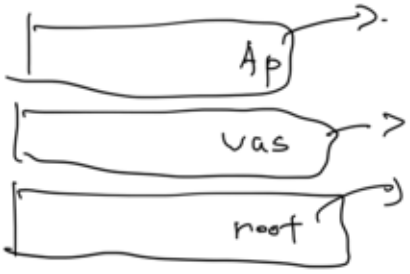
python dependent

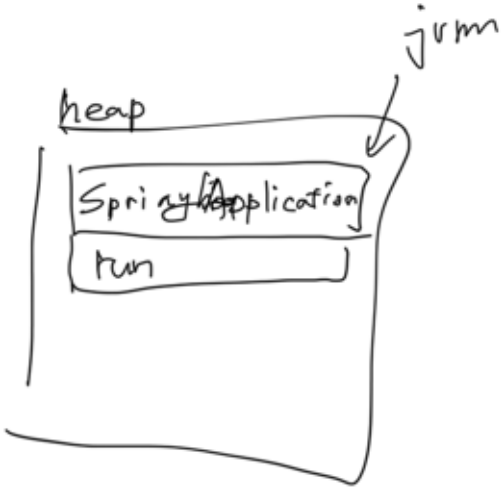
Was App classloader

System class

Java class loading . root ~~System~~ class loader

WAS jboss





class loader

↳ jar

classpath ~ jar 4G

-> 메모리 누수 있는 애플리케이션

-> class loading issue

-> Memory leak Out of memory heap dump

-> deadlock issue thread dump