Oracle MOOC: JVM Troubleshooting

Lab 3: Diagnostic Data Collection and Analysis Tools

In this lab, you'll run several test programs and analyze their memory leakage. Your goal is to diagnose and pinpoint possible leak sources. Begin by unzipping `Lab3_Files.zip`. You must also have Eclipse MAT installed on your machine

---

Part 1: Memory Usage Growth in the Java Heap Space

1. Compile and run the Java program `JavaHeapMemoryLeak`.
    a. Open a command prompt or terminal.
    b. Compile the program by typing: `javac JavaHeapMemoryLeak.java`
    c. Run the program with `HeapDumpOnOutOfMemoryError` so as to collect a heap dump in case the program fails with OutOfMemoryError exception. This can be done by typing:

```
java -Xmx512m -XX:+HeapDumpOnOutOfMemoryError
JavaHeapMemoryLeak
```

2. Collect the heap dump.
    a. Notice the program fails with the exception `OutOfMemoryError: Java heap space`.
    b. Locate the heap dump in the current working directory. The JVM creates a heap dump with the `.hprof` file extension.

3. Analyze the heap dump in Eclipse MAT.
    a. Launch Eclipse MAT and open the newly created heap dump.
    b. Open the **Histogram** view and find the top objects occupying the largest portion of the heap.
    c. Right-click on the top most object type which is `byte[]` and click **List objects >>With incoming reference**. This will bring up the **list_objects** view, which lists all the objects for that particular class.
    d. Right click on any of these instances and click **Merge Shortest Paths to GC Roots >>Exclude weak references'** to list reference chains excluding weak references from that instance to any of the GC roots. This will bring up the **merge_shortest_paths** window view. This shows the reference paths through which that particular instance is kept alive in the Java Heap.
    e. From the reference paths, find the object that is created in the `JavaHeapMemoryLeak` program and is responsible for holding on to that `byte[]` instance.
    f. Repeat steps d and e for some other instances in the **list_objects** view mentioned in step c.

Oracle MOOC: JVM Troubleshooting

Note: After this exercise, we can see that the LinkedList `arrObjects` created in `JavaHeapMemoryLeak` is holding on to these `byte[]` instances.

4. Look at the source code of this program to understand how and why these `byte[]` instances are held in the heap. Also, determine why these instances aren't collected by the garbage collector.

5. Explore possible solutions how you can avoid this memory leak in the Java Heap space.

Part 2: Memory Usage Growth in the Metaspace

1. Compile and run the Java program `MetaspaceMemoryLeak` with JDK 8 or 9.
   a. Open a command prompt or terminal.
   b. Compile the remaining java class files you extracted from `Lab3_Files.zip`. You can compile all the files at once by typing: `javac *.java`
   c. Run the program with `HeapDumpOnOutOfMemoryError` so as to collect a heap dump in case the program fails with an `OutOfMemoryError` exception:

```
Java -XX:MaxMetaspaceSize=300m -
XX:+HeapDumpOnOutOfMemoryError MetaspaceMemoryLeak
```

2. Collect the heap dump
   a. Notice the program fails with the exception `java.lang.OutOfMemoryError: Metaspace`.
   b. Locate the heap dump in the current working directory. The JVM creates a heap dump with the `.hprof` file extension.

3. Analyze the heap dump in Eclipse MAT.
   a. Launch Eclipse MAT and open the created heap dump in it.
   b. Open the **Class Loader Explorer** view by clicking on **Open Query Browser >>Java Basics >>Class Loader Explorer**. Explore different classloaders and the classes they load.
   c. Open the **Duplicate Classes** view and explore the various duplicate classes loaded by multiple classloaders.
   d. Note that there are numerous `TemplateClassLoader` instances, and there are duplicate classes loaded by these classloader instances.
   e. Right-click on any of these classes and click **List objects >>With incoming references**. This will bring up the **list_objects** view, which lists all the loaded instances of that particular class.
   f. Right click on any of these class instances, and click **Merge Shortest Paths to GC Roots >>Exclude weak references** to list reference chains excluding weak

Oracle MOOC: JVM Troubleshooting

references from that class to any of the GC roots. This will bring up the **merge_shortest_paths** window view. This shows the reference paths through which a particular class is kept alive in the class metadata space

g. From the reference paths, find the object that is created in the `MetaspaceMemoryLeak` program and is responsible for holding on to those duplicate classes.

h. Repeat steps f and g for some other classes in the **list_objects** view mentioned in step e.

Note: After this exercise, we can see that the List `classesCache` created in `MetaspaceMemoryLeak` is holding on to these class instances.

4. Look at the source code of this program to understand how and why these classes are loaded multiple times. Also, determine why these classes are not collected by the garbage collector.

5. Explore possible solutions on how you can avoid this memory leak in the Metaspace.