PermGen vs. Metaspace runtime comparison

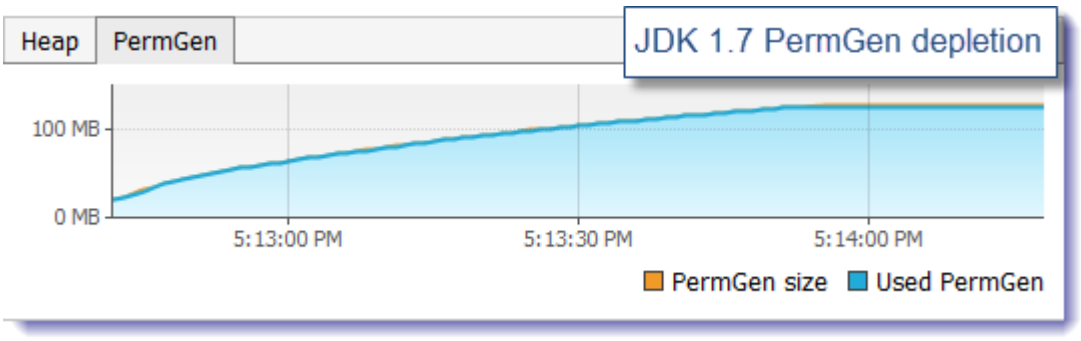
In order to better understand the runtime behavior of the new Metaspace memory space, we created a class metadata leaking Java program. You can download the source here.

The following scenarios will be tested:

- Run the Java program using JDK 1.7 in order to monitor & deplete the PermGen memory space set at 128 MB.
- Run the Java program using JDK 1.8 (b75) in order to monitor the dynamic increase and garbage collection of the new Metaspace memory space.
- Run the Java program using JDK 1.8 (b75) in order to simulate the depletion of the Metaspace by setting the MaxMetaspaceSize value at 128 MB.

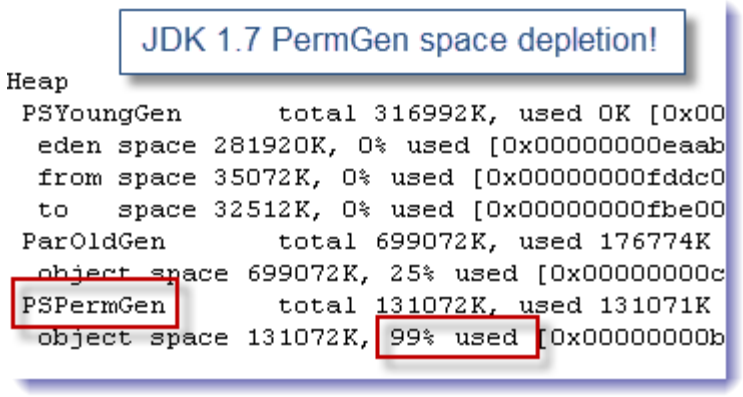
JDK 1.7 @64-bit – PermGen depletion

- Java program with 50K configured iterations
- Java heap space of 1024 MB
- Java PermGen space of 128 MB (-XX:MaxPermSize=128m)



As you can see form JVisualVM, the PermGen depletion was reached after loading about 30K+ classes. We can also see this depletion from the program and GC output.

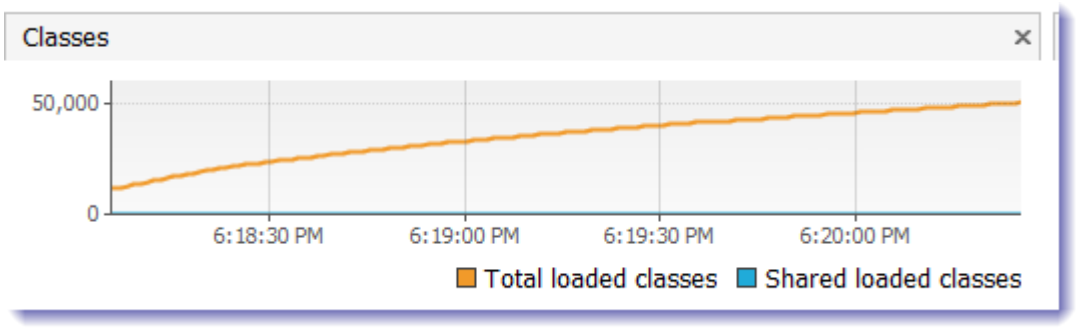
```
Class metadata leak simulator
Author: Pierre-Hugues Charbonneau
http://javaeesupportpatterns.blogspot.com
ERROR: java.lang.OutOfMemoryError: PermGen space
```



Now let's execute the program using the HotSpot JDK 1.8 JRE.

JDK 1.8 @64-bit – Metaspace dynamic re-size

- Java program with 50K configured iterations
- Java heap space of 1024 MB
- Java Metaspace space: unbounded (default)




```
3.162: [GC (Metadata GC Threshold) [PSYoungGen: 199286K->29344K(305856K
3.219: [Full GC (Metadata GC Threshold) [PSYoungGen: 29344K->0K(305856K
6.153: [GC (Metadata GC Threshold) [PSYoungGen: 155324K->22688K(305856K
6.220: [Full GC (Metadata GC Threshold) [PSYoungGen: 22688K->0K(305856K
13.777: [GC (Metadata GC Threshold) [PSYoungGen:                        6
13.881: [Full GC (Metadata GC Threshold) [PSYoun                        6
26.105: [GC (Allocation Failure) [PSYoungGen: 26                        ]
36.925: [GC (Metadata GC Threshold) [PSYoungGen:                        6
37.101: [Full GC (Metadata GC Threshold) [PSYoun                        6
57.717: [GC (Allocation Failure) [PSYoungGen: 26                        ]
78.448: [GC (Allocation Failure) [PSYoungGen: 263904K->62848K(286656K)]
101.297: [GC (Metadata GC Threshold) [PSYoungGen: 277959K->62848K(22150
101.573: [Full GC (Metadata GC Threshold) [PSYoungGen: 62848K->0K(22150
121.502: [GC (Allocation Failure) [PSYoungGen: 158656K->24288K(254080K)
142.407: [GC (Allocation Failure) [PSYoungGen: 182944K->49824K(258176K)
Heap
 PSYoungGen      total 258176K, used 72641K [0x00000000eaab0000, 0x0000
  eden space 162880K, 14% used [0x00000000eaab0000,0x00000000ec0f8738,0
  from space 95296K, 52% used [0x00000000fa2f0000,0x00000000fd398030,0x
  to   space 91328K, 0% used [0x00000000f49c0000,0x00000000f49c0000,0x0
 ParOldGen       total 699072K, used 262735K [0x00000000c0000000, 0x000
  object space 699072K, 37% used [0x00000000c0000000,0x00000000d0093c10
 Metaspace total 304492K, used 191055K, reserved 335872K
  data space     204104K, used 160283K, reserved 233472K
  class space    100388K, used 30771K, reserved 102400K
```

JDK 1.8
Metaspace dynamic
re-size

from **20 MB**...**328 MB**

As you can see from the verbose GC output, the JVM Metaspace did expand dynamically from 20 MB up to 328 MB of reserved native memory in order to honor the increased class metadata memory footprint from our Java program. We could also observe garbage collection events in the attempt by the JVM to destroy any dead class or classloader object. Since our Java program is leaking, the JVM had no choice but to dynamically expand the Metaspace memory space.

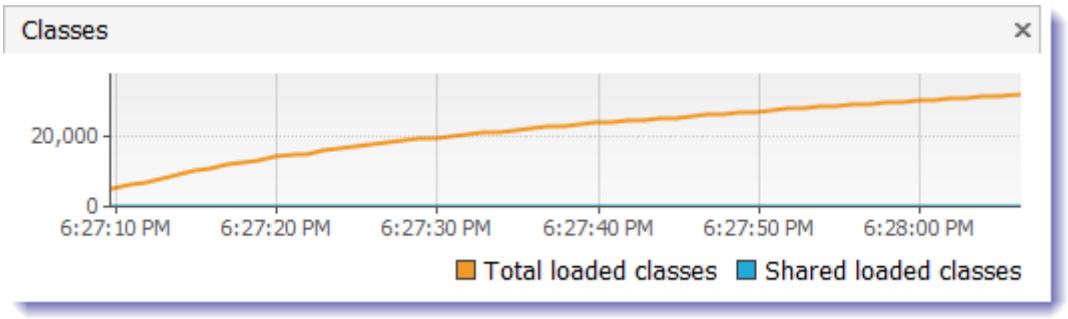
The program was able to run its 50K of iterations with no OOM event and loaded 50K+ Classes.

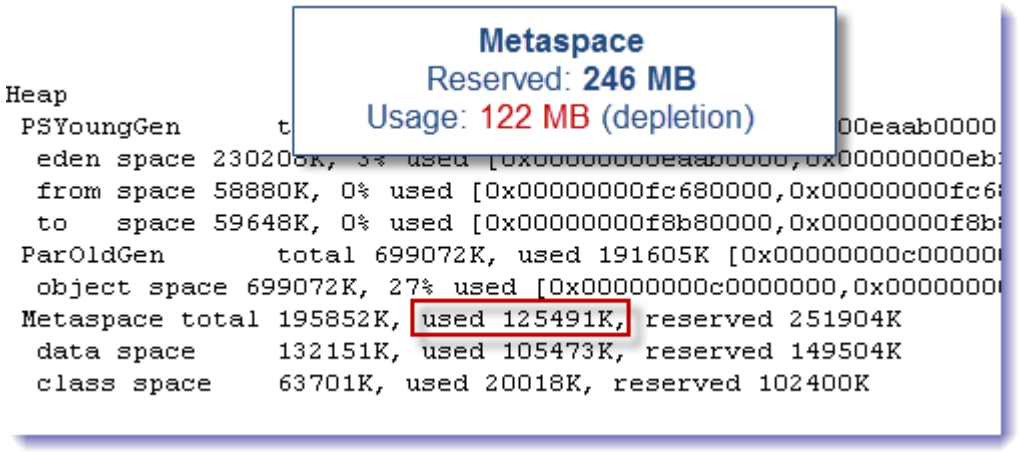
Let's move to our last testing scenario.

JDK 1.8 @64-bit – Metaspace depletion

- Java program with 50K configured iterations
- Java heap space of 1024 MB

- Java Metaspace space: 128 MB (-XX:MaxMetaspaceSize=128m)





As you can see form JVisualVM, the Metaspace depletion was reached after loading about 30K+ classes; very similar to the run with the JDK 1.7. We can also see this from the program and GC output. Another interesting observation is that the native memory footprint reserved was twice as much as the maximum size specified. This may indicate some opportunities to fine tune the Metaspace re-size policy, if possible, in order to avoid native memory waste.

Now find below the Exception we got from the Java program output.

```
Class metadata leak simulator
Author: Pierre-Hugues Charbonneau
http://javaeesupportpatterns.blogspot.com
ERROR: java.lang.OutOfMemoryError: Metadata space
Done!
```

As expected, capping the Metaspace at 128 MB like we did for the baseline run with JDK 1.7 did not allow us to complete the 50K iterations of our program. A new OOM error was thrown by the JVM. The above OOM event was thrown by the JVM from the Metaspace following a memory allocation failure.

#metaspace.cpp

```
if (result == NULL) {
  // Try to clean out some memory and retry.
  result =
    Universe::heap()->collector_policy()->satisfy_failed_metadata_alloc
      loader_data, word_size, mdtype);

  // If result is still null, we are out of memory.
  if (result == NULL) {
    if (Verbose && TraceMetadataChunkAllocation) {
      gclog_or_tty->print_cr("Metaspace allocation failed for size "
        SIZE_FORMAT, word_size);
      if (loader_data->metaspace_or_null() != NULL) loader_data->metasp
      MetaspaceAux::dump(gclog_or_tty);
    }
    // -XX:+HeapDumpOnOutOfMemoryError and -XX:OnOutOfMemoryError suppo
    report_java_out_of_memory("Metadata space");

    if (JvmtiExport::should_post_resource_exhausted()) {
      JvmtiExport::post_resource_exhausted(
          JVMTI_RESOURCE_EXHAUSTED_OOM_ERROR,
          "Metadata space");
    }
```

Metaspace Depletion

Final words

I hope you appreciated this early analysis and experiment with the new Java 8 Metaspace. The current observations definitely indicate that proper monitoring & tuning will be required in order to stay away from problems such as excessive Metaspace GC or OOM conditions triggered from our last testing scenario. Future articles may include performance comparisons in order to identify potential performance improvements associated with this new feature.

Please feel free to provide any comment.