Java Class Metadata: A User Guide

Andrew Dinn: Red Hat

Part 1: Class Metadata and Other Native Memory

What is Java Class Metadata?

As you are probably well aware Java applications are executed by a program called a Java Virtual Machine (JVM). Instead of compiling your Java code to an executable program -- as you might do with a language like C or C++ -- you compile them to bytecode stored either in single class files or jars containing a suite of class files. When you run the java command you point the JVM at a compiled class containing the program main method. The JVM loads this main class, and any other classes it needs, as it creates objects and executes methods of the classes described in the bytecode.

The OpenJDK Java Virtual Machine (JVM) records many details of the loaded Java class base in memory as it executes the application, and this information is referred to as Class Metadata. Metadata is necessary to support powerful Java features like (lazy) dynamic class loading, JIT compilation, reflection and agent transformation. Different JVMs may record more or less information in memory in order to make a trade-off between lower metadata storage costs or faster execution. OpenJDK uses a relatively rich metadata model in order to get the fastest possible performance.

This article is the first in a series about OpenJDK Class Metadata. It explains why it is needed and shows you how to measure the total amount of storage allocated for metadata in a running Java application. Depending on the size of your code base and the complexity of the design, metadata costs may vary from a few megabytes to several 100s of megabytes. So, it can be very useful to know where your design might incur specific metadata costs and, perhaps, to take that into account when developing or tuning an application.

Later articles will address the question of what you can do to minimize metadata costs when you design and implement your Java app to run on OpenJDK. This will include:

- how to obtain statistics which report per-class metadata storage costs i.e. exactly how many bytes of memory are used to store metadata for any given loaded class
- how these per-class costs can be subdivided and correlated with the size and number of (C++) objects used internally by the JVM to model your class
- how different design choices can lead to greater or smaller metadata costs

Case studies will be presented to show how specific savings may be achieved.

Why Class Metadata?

A managed runtime like the JVM needs to retain a lot more runtime information about the code being executed than would appear in a program that gets compiled offline to an executable binary. There are several reasons for this. Firstly, unlike normal compiled programs, the code base is not bounded. Classes can be discovered and loaded at runtime without the need for a direct reference to the class to exist in the code loaded at program start. Indeed, some classes may even be synthesised at runtime using tools like ASM or Javassist or may be redefined at runtime by a JVMTI agent.

So, the JVM must keep track of which classes exist in the runtime, what methods and fields they contain, and be able to dynamically resolve references from one class to another during link-loading. Linking of classes also needs to respect class visibility and accessibility. Class metadata is associated with classloaders whose delegation paths are used to determine visibility. Class permissions and packages are recorded in metadata in order to determine accessibility.

Secondly, Java programs need either to be interpreted or have their bytecode JIT compiled to native machine code. The interpreter and JIT compiler need to know details of the class hierarchy (supers, implemented interfaces, subclasses), class properties and layout (accessibility, abstract vs static, field and method types/signatures) and method implementation (bytecode, exceptions, stack layouts) to, respectively, interpret bytecode correctly or generate valid machine code. Indeed, JITted code sometimes needs to be able to refer to class structure in order to operate correctly. For example, code which implements a checkcast must be able to locate the instance being tested in the class hierarchy. Again, both the interpreter and JITted code need to collect profile statistics and associate them with methods during execution in order to guide adaptive (re)compilation.

Thirdly, the Java language supports reflection, allowing classes and class structure, including fields and methods, to be reified as Java objects. Reflection must support indirect access to fields and execution of methods via these reified Field or Method objects. (and in recent Java releases MethodHandles provide an alternative model for indirect execution). This requirement reinforces the need to retain details of class structure in the runtime and expands the amount and type of information that needs to be retained.

Finally, the JVM supports the use of agents to transform existing classes at runtime. As with reflection, this capability is only possible if the runtime retains information about the original class structure. In some circumstances the use of agent transformation imposes the need to retain extra information that would otherwise be dropped.

Why not just use bytecode?

All of the information needed by the JVM to execute Java code is available in the class bytecode. In theory, this information needs only to be supplemented with runtime-generated information such as profile data, the address(es) of compiled code buffers etc. Indeed, it would be possible for a JVM simply to base all its internal operations on in-memory bytecode, or perhaps even, bytecode re-loaded on demand when it is needed.

In practice, the JVM maintains its own internal representation of the currently loaded class base and this internal class model is what is referred to as class metadata. Metadata is *derived* from the bytecode loaded by the bootstrap, system and application classloaders. It replicates essentially the same *information* as is contained in the class file byte array but unpacks it into a tree of C++ instances and compresses it in the process, for example by deduplicating text shared across different class files.

This internal model of the class file contents is needed for several reasons. Firstly, although a class's bytecode is packed into a relatively compact format (a byte[]), that format does not allow the JVM to access the data it contains quickly or efficiently. Secondly, it does not allow the data to be easily annotated or, in a few rare cases, overwritten with runtime-derived data (the JVM supplements the class model with a variety of extra data to improve performance -- for example, a cache listing all of a class's supers and interfaces is used to speed up type-based operations). Finally, as mentioned earlier, the JVM can save space by eliding common data embedded in many different class files.

That last point is worth amplifying on. Bytecode files contain large numbers of (UTF) text strings all of which are referenced (using a bytecode offset) from a table called the constant pool: these include package, class & interface names, method & field names, method signature strings, String literal text, etc. The use of a constant pool in the class file format already achieves a certain level of compression in the method bytecode. For example, a get_field instruction needs to identify the owning class and field names. Each get_field bytecode is followed by a two byte operand which indexes the constant pool (referred to as a CP index). The associated pool slot contains a pair of CP indices. The first index recursively identifies the owner class name slot, which embeds the UTF text for the class name. The second identifies the field name which, likewise, embeds the field name. So, repeat mentions of class or member names only require an index to be replicated in the bytecode. UTF text strings only occur once per class file. Symbol text constitutes the vast bulk of the bytes in rt.jar (~90%) so anything which helps avoid duplication of this data can save a lot of space.

Even greater savings are possible when converting class bytes to metadata, by eliding the *same* symbols found in *different* class files. Many names occur with very high frequency; for example, consider how often a name like "Ljava/lang/String;" appears as the type of a field, method parameter, owner type for an invoked method, etc. In consequence, the JVM uses a global Symbol Dictionary to record text strings found in bytecode. Repeat occurrences of the (textually) same symbol found in successive class constant pools end up being stored in the associated metadata as pointers to the same unique Symbol instance. Similarly, String objects occurring in more than one class file are created once on the Java heap and re-used.

Metadata Memory Management

The JVM impements its own memory management for storage of Class Metadata. It carves out a Metaspace memory region from the available virtual memory range. This is a distinct memory area, separate from the heap area used to store Java instance data. Pages are committed as needed up to the limit configured using flag -XX:MaxMetaspace. If command line option -XX+UseCompressedClassPointers (default true) is configured then this region may be split into two subregions referred to as the class and data (sic) metadata regions. The data region can be allocated anywhere. The JVM attempts to map the class metadata region in the low end of memory. However, it can still profit from splitting the two regions even if the class metadata region has to be allocated at some other address.

The class metadata region is used to allocate instances of (C++) class Klass that the VM uses to store the primary details of a loaded class. The data region is used to allocate instances of other C++ metadata classes, holding details of Symbol text, methods, annotations, execution profile stats etc. The reason for this partitioning of metadata is that a Java object on the Java heap needs to have a Klass pointer embedded in its header. Normally a 64-bit pointer is needed. However, if the class metaspace region is configured smaller than a few GB (which is normally the case) then it is possible to embed a 32 bit offset from the start of the class region into the header saving heap space. Even better, if the region sits in the low 32-bits of the address space then the 32 bit header word can contain a direct (32-bit) pointer rather than an offset.

The JVM uses its own (very efficient -- at around 80/90% utilization) region and chunk management algorithms to partition metadata space between classloaders. Almost all the metadata associated with any given class is carved out of chunks that ‘belong’ to its own classloader. This makes it easy to reclaim metadata when a class loader is dropped. Essentially, freeing the metadata involves releasing all the classloader's chunks back into the free chunk pool with no need to free the smaller blocks carved out of each chunk. The one exception is Symbol metadata. As mentioned earlier, Symbols may be common to different classes (and hence class loaders), so the JVM maintains them in separately managed Symbol storage. The System Dictionary maintains a reference count for each Symbol. So, when a classloader is dropped the Symbols from each of its classes must be counted off, allowing them to be reclaimed when the count is zero.

Native Memory Tracking and Metadata

The JVMs’ Native Memory Tracking (NMT) diagnostic commands can be used to provide both summary and detailed reports of a variety of different aspects of JVM memory use, including memory used for Class Metadata and Symbols. Summary reports distinguish memory allocated using the JVM’s internal memory manager from the (usually) small amount of memory allocated using the underlying operating system malloc and free routines. The internal manager report distinguishes mapped virtual memory ranges from committed physical page counts (a recent update to these stats breaks down committed pages into actively used space, unused space including blocks and chunks restored to free lists and space wasted in headers and fragments too small to be of use). Detailed reports complement the summary output with details of all memory allocation and free operations, including a stack backtrace showing the originator of each such call.

NMT stats can be obtained from a running JVM using the VM.native_memory option of the jcmd tool. However, use of this option is only possible if the JVM is started with the relevant command line argument:

```
$ java -XX:NativeMemoryTracking=summary HelloWait
Hello, world!
enter <cr>:
```

n.b. HelloWait is a variant of Hello which prints the usual message then waits for input.

The command line argument requests the JVM to collect summary statistics recording allocation and release of native (i.e. non-Java heap) memory, including metadata memory. This alternative option collects detailed statistics

```
$ java -XX:NativeMemoryTracking=detail HelloWait
. . .
```

n.b.b. Detailed native memory tracking adds a small extra overhead to JVM metadata footprint in order to record some details of allocation and free call sites. It should normally only be used in production deployments for trouble-shooting and problem diagnosis.

While the program is still running it is possible to gather NMT statistics using jcmd

```
$ jcmd -l
5553 HelloWait
5577 sun.tools.jcmd.JCmd -l
$ jcmd 5553 VM.native_memory summary
5553

Native Memory Tracking:

Total: reserved=5526215KB, committed=430227KB
 - Java Heap (reserved=4038656KB, committed=253952KB)
             (mmap: reserved=4038656KB, committed=253952KB)

 -     Class (reserved=1061997KB, committed=10093KB)
             (classes #389)
             (malloc=5229KB #139)
             (mmap: reserved=1056768KB, committed=4864KB)
  . . .
 -    Symbol (reserved=1348KB, committed=1348KB)
             (malloc=892KB #67)
             (arena=456KB #1)
  . . .
```

The output show details of memory use for Class Metadata under the Class section, listing how many classes have been loaded, how much virtual memory has been reserved and how much of that address space has been committed to physical memory. Note that Symbol memory use is detailed separately from Class data. Symbol storage makes use of an alternative to mmap-based Block/Chunk manager called an Arena. Other stats not shown above report memory use for Thread Stacks, GC, Compiler, JITted code etc.

NMT is very useful for identifying the overall cost of Class Metadata storage but it provides little help when it comes to understanding how these costs relate to the design and implementation of the deployed application and JDK runtime code. Ideally, it would be useful to be able to identify each class's contribution to the in use count and break down this contribution further into different blocks of storage used to model different structural properties of the class and its methods. That will be the topic of the next article in this series.