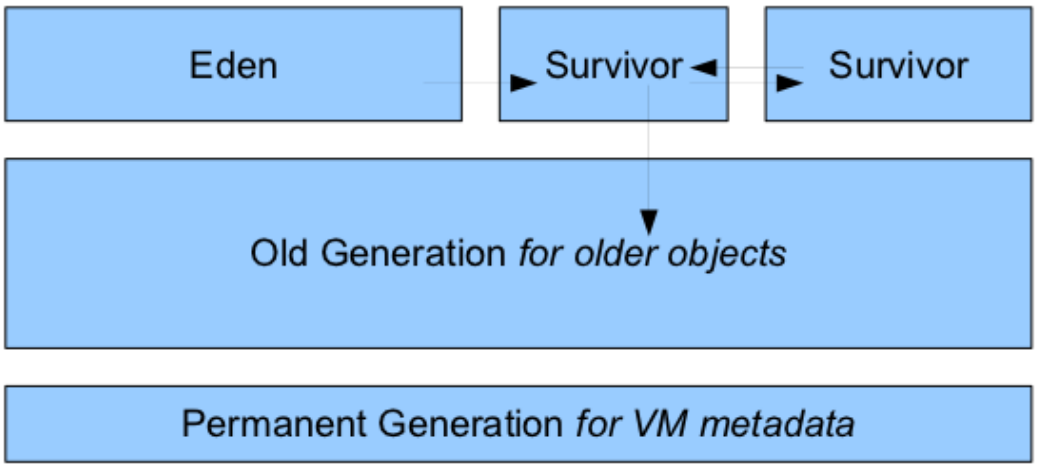
Metaspace in Java 8

This is how Heap Structure look like in Java 6



Permanent Generation

The pool containing all the reflective data of the virtual machine itself, such as class and method objects. With Java VMs that use class data sharing, this generation is divided into read-only and read-write areas.
The Permanent generation contains metadata required by the JVM to describe the classes and methods used in the application. The permanent generation is populated by the JVM at runtime based on classes in use by the application. In addition, Java SE library classes and methods may be stored here.
Classes may get collected (unloaded) if the JVM finds they are no longer needed and space may be needed for other classes. The permanent generation is included in a full garbage collection

- Region of Java Heap for JVM Class Metadata.
- Hotspot's internal representation of Java Classes.
- Class hierarchy information, fields, names
- Method compilation information and bytecodes
- Variables
- Constant pool and symbolic resolution

PermGen Size

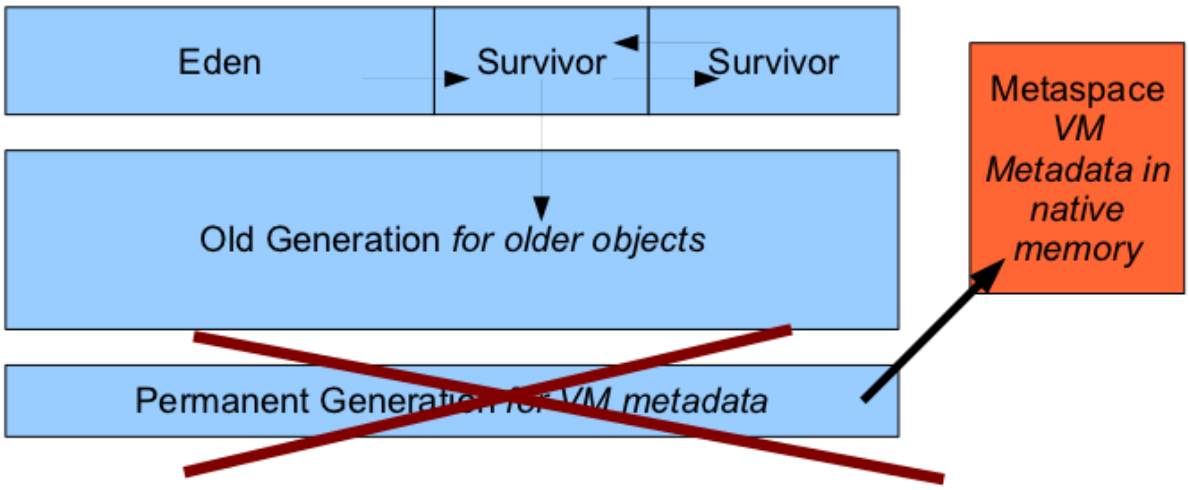
- Limited to MaxPermSize – default ~64M - 85M
- Contiguous with Java Heap : Identifying young references from old gen and permgen would be more expensive and complicated with a non-contiguous heap – card table(A kind of remembered set that records where oops have changed in a generation).
- Once exhausted throws OutOfMemoryError "PermGen space".
- Application could clear references to cause class unloading.
- Restart with larger MaxPermSize.
- Size needed depends on number of classes, size of methods, size of constant pools.

Why was PermGen Eliminated?

- Fixed size at startup – difficult to tune.
- -XX:MaxPermSize=?

- Internal Hotspot types were Java objects : Could move with full GC, opaque, not strongly typed and hard to debug, needed meta-metadata.
- Simplify full collections : Special iterators for metadata for each collector
- Want to deallocate class data concurrently and not during GC pause
- Enable future improvements that were limited by PermGen.

Where did JVM Metadata go now?



Error! Filename not specified.

Metaspace

The Permanent Generation (PermGen) space has completely been removed and is kind of replaced by a new space called Metaspace.
The consequences of the PermGen removal is that obviously the PermSize and MaxPermSize JVM arguments are ignored and you will never get ajava.lang.OutOfMemoryError: PermGen error.
The JDK 8 HotSpot JVM is now using native memory for the representation of class metadata and is called **Metaspace**.

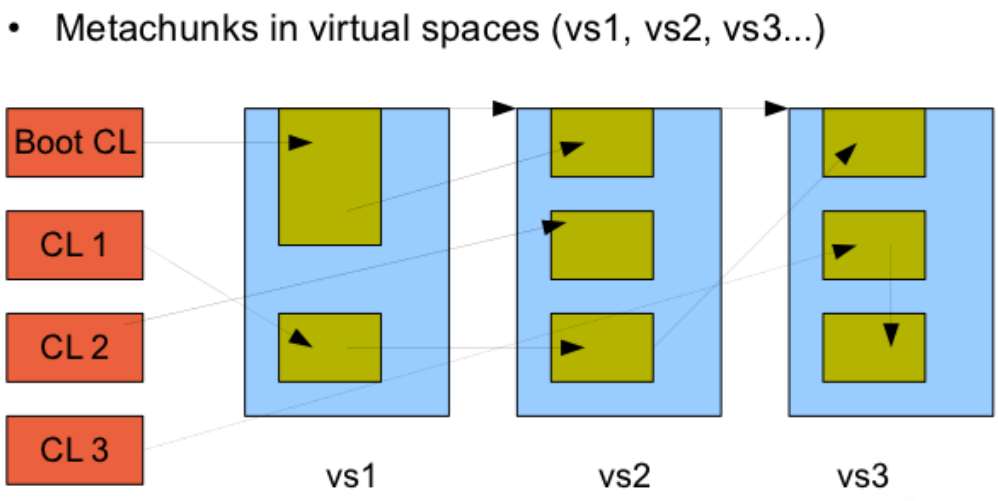
- Take advantage of Java Language Specification property : Classes and associated metadata lifetimes match class loader's.
- Per loader storage area – Metaspace
- Linear allocation only
- No individual reclamation (except for RedefineClasses and class loading failure)
- No GC scan or compaction
- No relocation for metaspace objects
- Reclamation en-masse when class loader found dead by GC

In Metaspace memory allocation model

- Most allocations for the class metadata are now allocated out of native memory.
- The classes that were used to describe class metadata have been removed.
- Multiple mapped virtual memory spaces allocated for metadata.
- Allocate per-class loader chunk lists

- Chunk sizes depend on type of class loader.
- Smaller chunks for sun/reflect/Delegating ClassLoader.
- Return chunks to free chunk lists.
- Virtual memory spaces returned when emptied.
- Strategies to minimize fragmentation.

We see how virtual memory space is allocated for metadata and how it loaded per -class loader with this picture



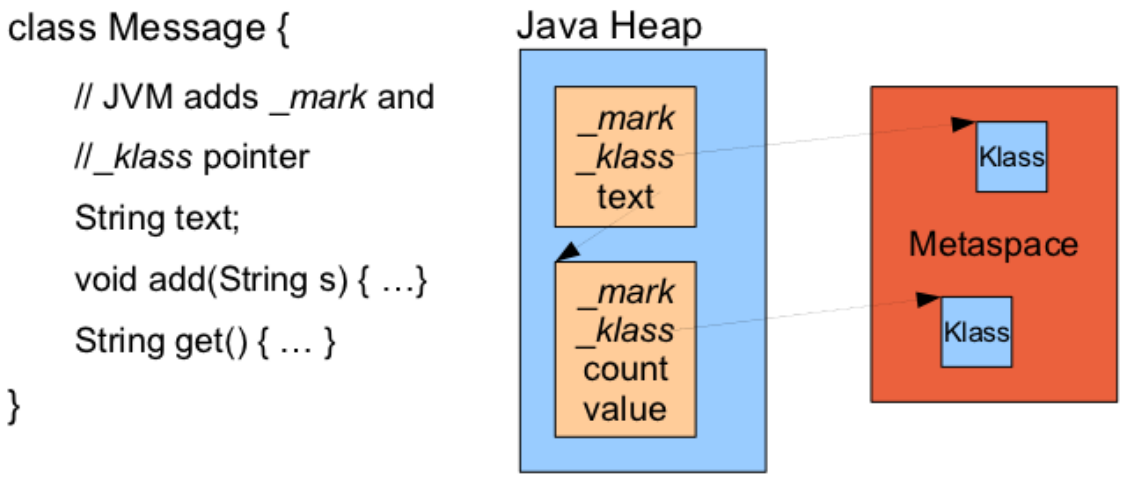
You can see how virtual memory space(vs1,vs2,vs3) allocated and how per-class loader chunk is allocated. CL - class loader

Understanding of *_mark* and *_klass* pointer
To understand the next diagram, you need to have an idea of these pointers.
In the JVM, every object has a pointer to its class, but only to its concrete class and not to its interface or abstract class.
For 32 bit JVM:
_mark : 4 byte constant
_klass : 4 byte pointer to class
The second field (_klass) in the object layout in memory (for a 32-bit JVM, the offset is 4, for a 64-bit JVM offset is 8 from the address of an object in memory) points to the class definition of object in memory.
For 64 bit JVM:
_mark : 8 byte constant
_klass : 8 byte pointer to class

For 64 bit JVM with compressed-oops:
_mark : 8 byte constant
_klass : 4 byte pointer to class

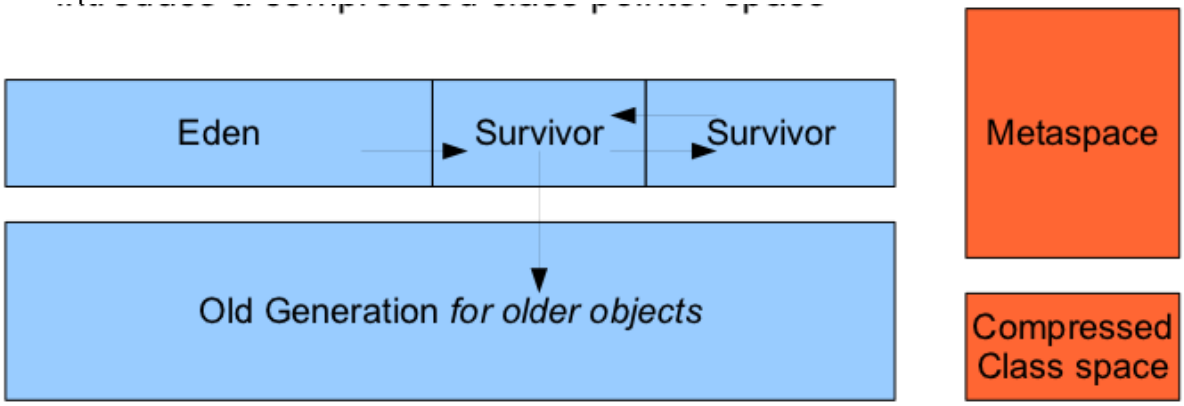
HotSpot Glossary of Terms

Java Object Memory Layout

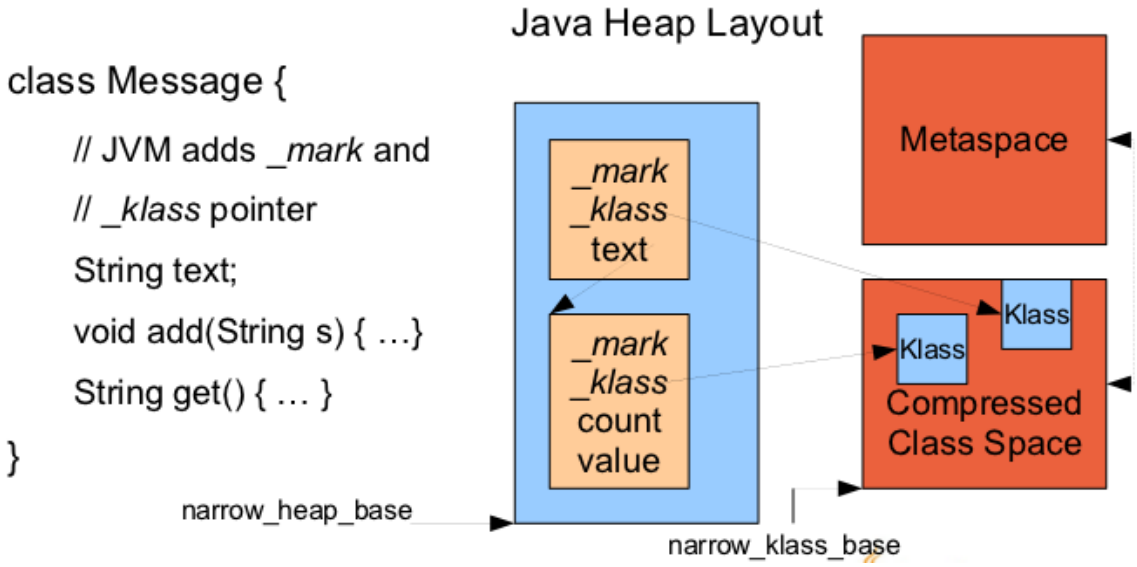


Compressed Class Pointer Space

This is case where we compressed the class pointer space and this is only for 64 bit platforms. For 64 bit platforms, to compress JVM _klass pointers in objects, introduce a compressed class pointer space



Java Object Memory Layout with Compressed Pointers



Summary of Compressed Pointers

- Default for 64 bit platforms.
- Compressed object pointers -XX:+UseCompressedOops
- "oops" are "ordinary" object pointers.
- Object pointers are compressed to 32 bits in objects in Java Heap.
- Using a heap base (or zero if Java Heap is in lower 26G memory).
- Compressed Class Pointers -XX:+UseCompressedClassPointers.
- Objects have a pointer to VM Metadata class (2nd word) compressed to 32 bits.
- Using a base to the compressed class pointer space.

Difference between Metaspace vs. Compressed Class Pointer Space

- Compressed Class Pointer Space contains only class metadata.
- InstanceKlass, ArrayKlass
- Only when UseCompressedClassPointers true.
- These include Java virtual tables for performance reasons.
- We are still shrinking this metadata type.
- Metaspace contains all other class metadata that can be large.
- Methods, Bytecodes, ConstantPool ...

Metaspace Tuning

The maximum metaspace size can be set using the -XX:MaxMetaspaceSize flag, and the default is unlimited, which means that only your system memory is the limit. The -XX:MetaspaceSize tuning flag defines the initial size of metaspace If you don't specify this flag, the Metaspace will dynamically re-size depending of the application demand at runtime.

Tuning Flags - MaxMetaspaceSize

- -XX:MaxMetaspaceSize={unlimited}
- Metaspace is limited by the amount of memory on your machine.
- Limit the memory used by class metadata before excess swapping and native allocation failure occur.
- Use if suspected class loader memory leaks.
- Use on 32 bit if address space could be exhausted.
- Initial MetaspaceSize 21 mb – GC initial high water mark for doing a full GC to collect classes.
- GC's are done to detect dead classloaders and unload classes.
- Set to a higher limit if doing too many GC's at startup.
- Possibly use same value set by PermSize to delay initial GC.
- High water mark increases with subsequent collections for a reasonable amount of head room before next Metaspace GC.
- See MinMetaspaceFreeRatio and MaxMetaspaceFreeRatio
- Interpreted similarly to analogous GC FreeRatio parameters

Tuning Flags - CompressedClassSpaceSize

- Only valid if -XX:+UseCompressedClassPointers (default on 64 bit).
- -XX:CompressedClassSpaceSize=1G.
- Since this space is fixed at startup time currently, start out with large reservation.
- Not committed until used.
- Future work is to make this space growable.
- Doesn't need to be contiguous, only reachable from the base address.
- Would rather shift more class metadata to Metaspace instead.
- In future might set ergonomically based on PredictedLoadedClassCount (experimental flag now).
- Sets size of other internal JVM data structures, like dictionary of loaded classes.

Tools for Metaspace

- jmap -permstat option renamed jmap -clstats
- Prints class loader statistics of Java heap. For each class loader, its name, liveness, address, parent class loader, and the number and size of classes it has loaded are printed. In addition, the number and size of interned Strings are printed.
- jstat -gc option shows Metaspace instead of PermGen.
- jcmd <pid> GC.class_stats.
- Gives detailed histogram of class metadata sizes.
- Start java with -XX:+UnlockDiagnosticVMOptions

Improved GC Performance

If you understand well about the Metaspace concept, it is easily to see improvement in Garbage Collection

- During full collection, metadata to metadata pointers are not scanned.
- A lot of complex code (particularly for CMS) for metadata scanning was removed.
- Metaspace contains few pointers into the Java heap.
- Pointer to java/lang/Class instance in class metadata
- Pointer to component java/lang/Class in array class metadata
- No compaction costs for metadata.
- Reduces root scanning (no scanning of VM dictionary of loaded classes and other internal hashtables).
- Improvements in full collection times.
- Working on class unloading in G1 after concurrent marking cycle

Summary

- Hotspot metadata is now allocated in Metaspace.
- Chunks in mmap spaces based on liveness of class loader.
- Compressed class pointer space is still fixed size but large.
- Tuning flags available but not required.
- Change enables other optimizations and features in the future
- Application class data sharing.
- Young collection optimizations, G1 class unloading.

- Metadata size reductions and internal JVM footprint projects