Java Class Metadata: A User Guide

Andrew Dinn: Red Hat

Part 4: A Class Metadata Stats Reference

Obtaining Other Available Class Stats

The sample listing provided earlier only includes the default set of Class Metadata stats. A more detailed breakdown of the metadata memory costs is possible. You can omit some of the default stats and/or choose to include others by passing an explicit, comma-separated list of names for the desired stats as argument to GC.class_stats. For example, the following command would list the minimal, useful set of stats:

```
  $ jcmd <pid> GC.class_stats
KlassBytes,MethodCount,CpAll,MethodAll,RoAll,RwAll,ClassLoader
```

Stats are always listed in a pre-defined order and the listing always includes an Index, Super index and Classname Many other stats can be obtained simply by adding the relevant name to the comma-separated list. Each of available stats is described in turn below.

Interpretation of each Available Stat

This section describes the full set of available stats, starting the doc string provided in the source code, expanded on slightly in cases where the doc string is limited or unclear. Further comments highlight stats that are likely to make a significant contribution to total metadata costs and, where possible, explains how certain implementation decisions might exacerbate the costs and how one might mitigate them.

Metadata Stats Reference

InstSize

Size of each object instance of the Java class

InstCount

Number of object instances of the Java class

InstBytes

The number of bytes used to store instances of this class.

This is usually (InstSize * InstNum). The only exception is class java.lang.Class, whose InstBytes also includes the slots used to store static fields. InstBytes is not counted in ROAll, RWAll or Total

Mirror

Size of the Klass's java_mirror object i.e. the associated instance of java.lang.Class.

This grows in proportion to the number of static fields and methods defined in the class.

ClassLoader

This is not actually a statistic but it is available to qualify classes further than by name alone.

This field identifies the classloader to which the reported class belongs. If stats are entered into a spreadsheet then this field can be used to sort and group related classes for a subcomponent loaded by a specific classloader. This is very useful when looking at classes deployed, say, as a Wildfly deployment, all of which will be managed by their own dedicated JBoss Modules loader.

KlassBytes

Size of the InstanceKlass or ArrayKlass for this class.

Note that this incorporates the space allocated for the vtable, itables and class oop map. This means that this stat will increase with number of local and inherited methods, with the number of local and inherited interfaces implemented by the class and with the number of methods defined in each of those interfaces.

Where KlassBytes is very large it will most likely be because of vtable or itable costs.

K_secondary_supers

Number of bytes used by the Klass::secondary_supers() array.

This cost is included in the KlassBytes total.

This array points to all parent types not included on the transitive supers list. For example, class String has one super, Object. It implements three local interfaces, Serializable, Comparable and CharSequence. So, the secondary supers array occupies 32 bytes (including the length and 3 Klass* pointers). Whereas, class String[] has two supers, Object[] and Object. It implements 5 local interfaces, Cloneable, Serializable, Serializable[], Comparable[] and CharSequence[] (the first two are implemented by every array).

Depending on the class structure this array may be null, shared with the local or transitive interfaces array, shared with a parent or take a default value common to many classes, in which case the cost may be zero.

VTab

Size of the embedded vtable in InstanceKlass

This cost is included in the KlassBytes total.

This grows with to number of non-private local and inherited methods. VTable entries accumulate down the class extends hierarchy i.e. if a method is recorded in a given class's VTable then it will also occur in every VTable of a subclass which extends that class. Local final methods and private methods are not included in the VTable as they can only be called by a direct invoke (INVOKESPECIAL).

This is very likely to be one of the main factors leading to an inflated KlassBytes cost. In particular, generated code can give rise to large vtables because generators sometimes produce classes with large numbers of methods. If possible generate private or local final methods which will not add to vtable costs (better still use inline bytecode if possible: bytecode is compact and will avoid method metadata costs).

ITab

Size of the embedded itable in InstanceKlass

This cost is included in the KlassBytes total.

This grows with the number of local and inherited interfaces implemented by the class and with the number of methods defined in each of those interfaces. ITable entries accumulate down the interface extends hierarchy i.e. if a method is recorded in a given interface’s ITable then it will also occur in every ITable of an subinterface which extends that interface.

This is very likely to be one of the main factors leading to an inflated KlassBytes cost. In particular, generated code can give rise to large itables because generators sometimes produce implementations for interfaces with large numbers of methods. If possible try to avoid using interfaces to isolate generated classes from generated implementations (override an abstract super, or better still dispense with overriding and simply generate the super with inline bytecode if possible: bytecode is compact and, anyway, the bytecode costs should be much the same for both alternatives but you will omit method metadata costs for the subclass).

OopMap

Size of the embedded nonstatic_oop_map in InstanceKlass

This cost is included in the KlassBytes total.

At its smallest this will comprise a single pair of ints identifying the start offset and count of all the contiguous object type fields in the class. If object type fields are split into N contiguous blocks by intervening non-object fields then this will occupy N * 2 ints.

This will rarely add much to the KlassBytes overhead. However,grouping related object fileds together as much as possible will save a small amount of space.

IK_methods

Number of bytes used by the InstanceKlass::methods() array

This cost is included in the KlassBytes total.

This is the vector (Array<Method*>) that links the class to its associated Method metadata objects. It's size grows with the count of the class's local and inherited methods.

IK_method_ordering

Number of bytes used by the InstanceKlass::method_ordering() array

This cost is included in the KlassBytes total.

This is the vector (Array<int>) that records the classfile (ordinal) position of the methods listed in the IK_methods vector. It's size is the same as IK_methods.

IK_default_methods

Number of bytes used by the InstanceKlass::default_methods() array

This cost is included in the KlassBytes total.

This is the vector (Array<Method*>) that lists methods provided by the Klass's interfaces as default implementations. It is only allocated when older compiled classes omit methods subsequently added to extend an interface (e.g. Collection methods added in JDK8).

IK_default_vtable_indices

Number of bytes used by the InstanceKlass::default_vtable_indices() array

This cost is included in the KlassBytes total.

This is a vector (Array<int>) that records the vtable index of methods appearing in the IK_default_methods vector. It is only allocated when that vector is needed an dhas the same size.

IK_local_interfaces

Number of bytes used by the InstanceKlass::local_interfaces() array

This cost is included in the KlassBytes total.

This is a vector (Array<Klass*>) identifying all locally implemented interfaces.

IK_transitive_interfaces

Number of bytes used by the InstanceKlass::transitive_interfaces() array

This cost is included in the KlassBytes total.

This is a vector (Array<Klass*>) identifying all locally implemented and inherited interfaces.

IK_fields

Number of bytes used by the InstanceKlass::fields() array

This cost is included in the KlassBytes total.

This is a vector of 2 byte values (Array<u2>) encoding details of all local and inherited fields. There are 7 values for each field: access flags, name CP index, sig CP index, initial value CP index, low_offset, high_offset, generic type sig CP index.

IK_inner_classes

Number of bytes used by the InstanceKlass::inner_classes() array

This cost is included in the KlassBytes total.

This is a vector of 2 byte values (Array<u2>) encoding details of all this class's inner classes. There are 4 values for each inner class: inner_class_CP index, outer_class_CP_index, // inner_name_CP_index, inner_class_access_flags. If this class is defined inside an enclosing method then the vector includes a few extra values idenitfying the enclosing method and its owner class. This vector is only allocated when inner classes or an enclosing method exist.

IK_signers

Number of bytes used by the InstanceKlass::signers() array

This cost is included in the KlassBytes total.

A signed class includes an array of signer data. This is actually an Object[] allocated on the Java heap and stored in a field of the Java mirror.

class_annotations

Size of class annotations

This is the size of all annotations attached to the class.

This cost is included in the annotations total.

class_type_annotations

Size of class type annotations

This is the size of all type annotations attached to the class.

This cost is included in the annotations total.

fields_annotations

Size of field annotations

This is the aggregate size of all annotations attached to fields of the class.

This cost is included in the annotations total.

fields_type_annotations

Size of field type annotations

This is the aggregate size of all type annotations attached to fields of the class.

This cost is included in the annotations total.

methods_annotations

Size of method annotations

This is the aggregate size of all annotations attached to methods of the class.

This cost is included in the annotations total.

methods_parameter_annotations

Size of method parameter annotations

This is the aggregate size of all annotations attached to parameters of methods of the class.

This cost is included in the annotations total.

methods_type_annotations

Size of methods type annotations

This is the aggregate size of all type annotations attached to parameters of methods of the class.

This cost is included in the annotations total.

methods_default_annotations

Size of methods default annotations

This is the aggregate size of all annotation default definitions attached to methods of the class.

This cost is included in the annotations total.

annotations

Size of all annotations

This includes costs for all class, field and method annotation structure, including annotation default value definitions provided in classfile AnnotationDefault attributes.

Cp

Size of InstanceKlass::constants()

This is the size of the constant pool which stores numeric values, references to symbolic values or indexes from one pool entry to other related entries.

CpTags

Size of InstanceKlass::constants()->tags()

This is the size of the tags array used to identify what is stored in each constant pool entry.

CpCache

Size of InstanceKlass::constants()->cache()

This is the size of the pool owner class's ConstantPoolCache which is used by the interpreter to perform quick lookups of any resolved classes, methods, strings etc mentioned in method bytecode. A single instance is created the first time that any method of the pool owner is interpreted and shared in all interpreter frames for those methods.

CpOperands

Size of InstanceKlass::constants()->operands()

This is an array of shorts used to hold operands for variable-size invokedynamic calls. It is normally empty and the size is rarely significant.

CpRefMap

Size of InstanceKlass::constants()->reference_map()

This is the array of integers used to map resolved_reference indices back to their original constant pool indices. It's size is determined by the number of class pool entries that resolve to classes, methods, strings etc.

CpAll

Sum of Cp + CpTags + CpCache + CpOperands + CpRefMap

MethodCount

Number of methods in this class

MethodBytes

Size of the Method objects

The per method average cost hardly varies. It has a minimum of 96 bytes and is rarely over 105 bytes.

ConstMethod

Size of the ConstMethod objects.

Note that this includes the fixed cost for the ConstMethod object itself plus the variable cost for data embedded in the tail region. The per method average cost varies quite substantially, depending mostly on the size of the method bytecode, line number table, local variable table. The size of the exception table, checked exceptions table, generic signature index and annotations array pointers usually don't add much to the total.

For generated code like parsers and loggers which have many methods the costs for line number and local variable tables is likely to be substantial. Omitting them from the classfile would probably be a very good idea. When all generated methods consume and/or produce exceptions according to a common exception contract, simplification of the exception signature by catching or throwing a common supertype might also save a significant amount of space.

MethodData

Size of the MethodData object

This is never likely to be a significant cost. A MethodData object is only created when the interpreter decides that a method has been executed enough times to make it worth profiling the method. So, for most methods the cost is zero.

It is possible to trim the size of the profile data by switching off type profiling using the command line option -XX:+TypeProfileLevel=0 to omit tracking of types for parameters/return types at entry, exit and method invocations. Type checking may be enabled solely for dynamic methods only using -XX:+TypeProfileLevel=111

StackMap

Size of the stackmap_data

This may be zero in some cases, either because methods are too simple to require stack maps or because bytecode is in pre-jdk7 format without ay stack map data included. Otherwise, it encodes the layout of the bytecode stack at specific offsets into the bytecode.

Stackmaps record what objects are in scope. They are only needed at points where a method may trigger a call to the garbage collector or generate/pass through an exception. So, as well as new allocations, that also includes loop back edges, method returns and potentially excepting bytecodes like get/putfield where an object is first dereferenced (excluding this which is always non-null).

A map records the bytecode location, stack height whether each stack slot stores a live object reference. Maps are compressed into a byte vector (Array<u1>) so each map is usually small. So, stackmap_data will only ever be large when a class has many methods most of which are large and complex.

Bytecodes

Of the MethodBytes column, how much are the space taken up by bytecodes

This breaks out a specific part of the ConstMethod total. The per method costs vary quite a lot, in a few cases 0, but mostly between 10 and a few hundred bytes. This is generally only a small part of the total cost for methods.

MethodAll

Sum of MethodBytes + Constmethod + Stackmap + MethodData

ROAll

Size of all class meta data that could (potentially) be placed in read-only memory. (This could change with CDS design)

RWAll

Size of all class meta data that must be placed in read/write memory. (This could change with CDS design)

Total

ROAll + RWAll. Note that this does NOT include InstBytes.