Java Class Metadata: A User Guide

Andrew Dinn: Red Hat

Part 3: Relating Metadata Stats to Structure

Introduction

In the previous article we saw some basic output obtained from the jcmd VM.class_stats diagnostic command and looked at a few ways of understanding the output in terms of how they relate to the internal structures used to model the class and method base.

Index	Super	**InstBytes**	KlassBytes	annotations	CpAll	MethodCount
	Bytecodes	MethodAll	ROAll	RWAll	Total	ClassName
1	-1	5163352	480	0	0	0
	0	0	24	584	608	[C
2	-1	2517064	480	0	0	0
	0	0	24	584	608	[Ljava.lang.Object;
3	46	2306634	560	0	1384	7
	149	1824	1096	2984	4080	java.util.HashMap$Node

This article will describe in more detail what those internal data structures look like. That will serve both to explain how the aggregate numbers add up and also make it possible to understand how the more detailed stats that are also available are to be interpreted. At the same time the explanation should make it clear precisely how every element of this internal model is populated: directly in response to entries in the class bytecode file; and, ultimately, as a consequence of choices made when designing and implementing the class. The final article will provide a categorical list of those detailed stats and provide advice as to how to relate them to the original class design and how to decide between different choices which have different metadata costs.

The description below, which explains how each subtotal is calculated from summing the sizes of the different C++ objects tallied in each group, will provide enough detail to make it clear what each of these component stats tracks when they are introduced in the final article, which also provides advice on how to minimalise the related cost.

KlassBytes

This column lists the total number of bytes used to store details of each class as a class *per se* i.e. excluding storage of any numeric literals or symbolic constants the class pulls in and excluding storage of the details of its methods, annotations and fields.

There is one main data structure per class, an instance of a C++ class called Klass although, actually what is allocated is an instance of a specific subclass of Klass -- either InstanceKlass, ObjArrayKlass or one of several other special case classes (e.g. C++ class TypeArrayKlass is used as the Klass which holds details of the primitive array types, byte[], int[], etc.).

```
Klass <--+--- InstanceKlass
          \-- ArrayKlass <--+--- ObjArrayKlass
                             \-- TypeArrayKlass
```

This statistic doesn't vary much for most classes because they simply store the same basic data in the fields common to Klass. An InstanceKlass is slightly larger than an ObjArrayKlass, etc but not by a great deal. Each of these instances occupies just under 500 bytes.

Any difference in the reported size is accounted for by two factors:

- the count incorporates a few linked C++ instances which record details of extra class structure needed to model more complex classes
- for an InstanceKlass extra space may be allocated in a variable length tail which immediately follows the Klass instance

The extra tail is used several auxiliary structures which are described in detail below

Most of the class data is stored in generic fields of Klass, which serve to locate the class in loader and type hierarchy.

```
Klass (184 bytes)
+-----------------------------------+
| name: Symbol*                     |
| loader: ClassloaderData*          |
| loader_next: Klass*               |
| super: Klass*                     |
| secondary_super_cache: Klass*     |
| secondary_supers: Array<Klass*>   |
| primary_supers: Klass*[8]         |
| super_check_offset: int           |
| first_sub: Klass*                 |
| sibling_next: Klass*              |
| modifiers/access: long            |
| gc_lock : . . .                   |
| java_mirror : oop                 |
| . . .                             |
+-----------------------------------+
```

Note that the quoted size of the C++ type is for the latest 64-bit x86 release of jdk8 (jdk8u-b144).

`loader_next` links each Klass in a chain from the ClassloaderData object that corresponds to its defining Java ClassLoader. loader provides a back link to that loader. n.b. the

ClassloaderData object doesn't only identify the Java loader, it also points to the memory manager which tracks region(s) from which this loader's Klass and related instances are allocated.

`name` identifies the Symbol Dictionary entry which idenitfies this klass. It is used as a hash key during class loading to reserve a place holder in the ClassloaderData, handling races to create the Klass.

`primary_supers` caches details of up to 8 classes in the super hierarchy starting from Object at offset 0 and working down to this class, assuming its depth is less than 8. If the class depth is 8 or more then extra supers are included in a vector of Klass* pointers referenced from secondary_supers. The latter array is also used to list all interfaces implemented (directly or transitively) by the Klass or, in the case of an array class, all compatible array super types preceded by the two interfaces implemented by all arrays, Cloneable and Serializable.

So, for example, for class String primary_supers contains 2 entries for Object and String and secondary_supers contains 3 entries for Serializable, Comparable and CharSequence. By contrast, for class String[] primary_supers contains 3 entries for Object, Object[] and String[] and secondary_supers contains 5 entries for Cloneable, Serializable, Serializable[], Comparable[] and CharSequence[].

This precomputed info allows almost all isSuper() and isSubclass() tests to be implemented by a small loop which increments a pointer initialised to point at slot 0 of primary_values. With less than 8 supers, super_check_offset indexes the entry containing this Klass i.e.the pointer high value at which the loop terminates.

With more than 8 supers, super_check_offset indexes field secondary_super_cache i.e. it identifies an address lower than primary_values slot 0. In this case the super check does not immediately iterate over the 8 values in the embedded primary_supers field. First it compares against the Klass* found in this slot (if any). If that comparison fails it then searches the full list in primary_supers, then any overflow entries in secondary_supers. If the latter search succeeds then the matched super is written back into secondary_super_cache on the assumption that any subsequent check will be for the same super type.

Clearly, this cached info can also be used to accelerate instanceof checks which compare not just against super classes but also against interfaces.

`first_sub` provide a quick lookup for the first rank of subclasses if it exists. Any further subclasses are chained using field sibling_next. There are a small number of other non-reference fields for value like modifiers, locks etc.

Note that the KlassBytes stat does not include the memory used for Symbol referenced from field name. Symbol costs are only available as an aggregate cost using the NMT stats diagnostic.

Every Klass instance also includes one further, special field, java_mirror, with C++ type oop (i.e. object-oriented pointer). This is a pointer to a Java object allocated on the Java heap which also happens to be the instance of java.lang.Class corresponding to this C++ Klass. Instances of java.lang.Class are not like other Java objects, not just because they are a reflective proxy for a Java class. They also have a variable size and that is because the tail of the class has variant contents. It is worth explaining this detail before describing the remaining Klass metadata.

Normally a Java object only includes storage for the instance fields declared by its Java class or superclasses. That is still true for instances of java.lang.Class -- they provide storage for all local or inherited instance fields. However, they also contain *extra* storage, enough to store values for all *static* fields of the Java class they proxy.

So, for example, let's assume we have a Java class called Foo. The mirror for the Klass modelling Foo will be the Java object identified by the Java literal Foo.class. If, say, Foo has 2 static fields of type int and Object then the mirror instance will include storage for all the fields of java.lang.Class plus extra space to store these two static values. The diagram below shows the instance of java.lang.Class on the Java heap, with its header pointing at the corresponding C++ Klass instance in the Metadata heap and the main body of the instance storing the values for instance fields name, serialVersionUID etc. The oop field java_mirror in the Klass points back at the instance of java.lang.Class.At the end of the Java instance is an extra data area storing the static field data.

```
Java Heap Space                         Metadata Space
                         +----------------------------------+
                         |                                  |
java.lang.Class          v        Klass                     |
+----------------------------+    +-----------------------+ |
| headerBits : long          |    | name : Symbol = "Foo" | |
| klass : Klass*       o--------> | . . .                 | |
+----------------------------+    | java_mirror : oop   o---+
| name : String = "Foo"      |\   +-----------------------+
| serialVersionUID :  . . .  | |
| reflectionData :    . . .  | | instance fields of Class
| . . .                      |/
|----------------------------|
| fooInt : int               |\ static fields of Foo.class
| fooObject : Object         |/
+----------------------------+
```

This extra overhead shows up in the line in the class_stats table that details storage for java.lang.Class. In this one case the value of InstBytes is much greater than InstCount * InstSize. Several of the other stats for java.lang.Class appear somewhat inflated because they are actually recording info about static fields and methods of the C++ Klass instances that these Java objects mirror.

Returning to the KlassBytes stats, another part of the variation in this statistic -- usually not the bulk of it -- is accounted for by variant fields belonging to each subclass of Klass. ArrayKlass and its two main subclasses, ObjArrayKlass and TypeArrayKlass, only add a small number of extra fields.

```
     ArrayKlass (224 bytes)
     +-----------------------------+
     | (Klass fields)              |
     | . . .                       |
     +-----------------------------+
     | dimension : int             |
     | higher_dimension: Klass*    |
     | lower_dimension: Klass*     |
     | component_mirror : oop      |
     | . . .                       |
     +-----------------------------+
```

`dimension` identifies levels of indirection from the base class e.g. if this is the ArrayKlass for Java class Foo[] it would be 1.

`lower_dimension` and `higher_dimension` identify the enclosing and component Metadata Klass instances (e.g. continuing the example this would be the ArrayKlass for Foo[][] (or null if it has not yet been created) and the InstanceKlass for Foo.

`component_mirror` points to the java_mirror of the Klass in lower_dimension.

ObjArrayKlass adds two more fields to support traversal down through arrays of lower dimension.

```
     ObjArrayKlass (240 bytes)
     +-----------------------------+
     | (ArrayKlass fields)         |
     | . . .                       |
     +-----------------------------+
     | element_klass: Klass*       |
     | bottom_klass: Klass*        |
     +-----------------------------+
```

TypeArrayKlass is used as the klass for 1-d arrays of primitives: byte[], char[] etc. It adds only one extra field to those inherited from ArrayKlass which identifies the maximum allowed length for the specific type of primitive array.

```
     TypeArrayKlass (232 bytes)
     +-----------------------------+
     | (ArrayKlass fields)         |
     | . . .                       |
     +-----------------------------+
     | max_length: int             |
     +-----------------------------+
```

InstanceKlass is used to model normal, instantiable Java classes. It adds many extra fields to those in its parent, Klass

```
InstanceKlass (424 bytes + 3*map/table)
+-------------------------------------+
| (Klass fields)                      |
| . . .                               |
+-------------------------------------+
| array_klass: Klass*                 |
| array_name : Symbol*                |
| local_intfs: Array<Klass*>*         |
| transitive_intfs : Array<Klass*>*   |
| local_methods: Array<Method*>*      |
| default_methods: Array<Method*>*    |
| class_method_order: Array<int>*     |
| vtable_method_order: Array<int>*    |
| generic_signature_idx: u2           |
| source_file_idx : u2                |
| java_field_count: u2                |
| java_static_oop_field_count : u2    |
| nonstatic_field_size: int           |
| static_field_size: int              |
| field_info: Array<u2>*              |
| class_annotations: Annotations*     |
| constants:     ConstantPool*        |
| flags/maj_min_version : long        |
| . . .                               |
| oop_map_size: int                   |
| vtable_length: int                  |
| itable_length: int                  |
| . . .                               |
+-------------------------------------+
```

`array_klass` and `array_name` provide quick access to the 1-d array for this InstanceKlass (if it exists).

`local_intfs` and `transitive_intfs` hold vectors which, respectively, cache details of locally implemented interfaces and all implemented interfaces. Storage for these vectors is included in KlassBytes.

`local_methods` records all methods declared by this InstanceKlass. The storage for this vector is included in KlassBytes but the Method objects it references are accounted for in the Method stats for this class.

`default_methods` records details of all default methods which this class inherits from implemented interfaces. The storage for this vector is included in KlassBytes but the Method objects it references are accounted for in the Method stats for the interface which owns the default methods.

Two int arrays `class_method_order` and `vtable_method_order` are used to record method order for this class's local methods. The first records source file order. The second records the order in which they are added to the 'vtable'.

n.b. a vtable for some given Klass is a table containing code pointers for every local or inherited method which might be executed using an invokevirtual instruction. Note that this means private methods are not included in the vtable. The pointer installed in a vtable may address JIT-compiled code or simply a stub routine which enters the interpreter. For some given InstanceKlass, the initial segment of the vtable will point to code for an inherited method or for a local overriding version of the method. Subsequent entries point to code for local public or protected methods. This organization ensures that code which executes a virtual method call can load a vtable from the method target's Klass, index at a predefined offset into the vtable and branch to the code address in order to execute code appropriate for the target.

n.b.b. A similar structure, called an itable, is used to resolve interface invocations. However, an InstanceKlass needs an itable for every interface it implements, whether directly or transitively. itables are listed preceded by a pointer to their owning interface's Klass. Interface invocation requires traversing the list to find the itable labelled with the target Klass before indexing into the itable to find the target code address for the call.

`generic_signature_idx` and `generic_signature_idx` are constant pool (CP) indices which are unsigned 2-byte (u2) numbers. These can be used to lookup Symbols which identify the class's source file name and generic type signature for the class (both of which may be absent). These 4 bytes are packed alongside two more u2 values, `java_field_count` and `java_static_oop_field_count`, which count how many fields the Java class declares and how many of those fields are static oop fields.

`static_field_size` and `non_static_field_size`, both packed into a long word, record how many bytes are needed to store, respectively, static field data and instance data. This is not directly proportional to the field count because takes into account: packing of Java 32-bit and 64-bit data; and whether or not Java object pointers can be stored in a 32-bit slot.

`field_info` stores details of all fields declared local to the class, encoded into a vector of u2. Each field is described by 6 primary 2-byte values: access flags, name CP index, sig CP index, initial value CP index, low_offset, high_offset. Following these entries the table then includes an extra, 7th 2-byte value for each field which contains the field's generic sig CP index (yes, this was shoe-horned in between JDK5/6). Storage for this vector is included in the KlassBytes total.

`annotations` points to a structure holding details of class-level annotations or null if none are present. The referenced Annotations structure is accounted for in the annotations statistics.

`constant_pool` points to a structure holding details of the constant values and Symbols referenced from this class. The referenced ConstantPool structure is accounted for in the CpAll statistics.

There are various other fields recording other important data including: class access permissions; the class major/minor version; whether this is a class or an interface; whether it is an inner class or not; details of any inner classes it may have; and dynamic class state used by JNI and the JIT.

n.b. for an agent transformed class the class will also retain a pointer to an untransformed byte[] representing the original class byte code.

Finally, an InstanceKlass contains three counts, `oop_map_size`, `vtable_length` and `itable_length`, that record the size of the 3 elements that may be embedded in its variable size tail: the oop map, vtable and itable.

Clearly, the various vectors (Array<> instances) which are attached to each InstanceKlass add extra storage costs that grow with the number of local fields and methods and with the complexity of the class and interface hierrachy in which the class sits. However, the largest variation in the KlassBytes statistic for any given class is usually down to the storage needed to embed OopMap, vtable and itable segments in the Klass's variable length tail.

OopMaps are needed to tell the garbage collector how to find object references embedded in Java instances. An OopMap is made up of one or more OopMapBlocks each of which contains a (int) start offset and (uint) count. So, each block describes a sequence of contiguous fields of object type. Field oop_map_size records how many blocks are needed to describe the layout of all object fields. If they are all allocated in one block with no intervening non-Object values then the count is 1 and the OopMap requires only two words plus the count. If object fields are interspersed with non-object fields the size of the map can be proportional to the number of fields.

As explained earlier, vtables and itables are blocks of memory which contain a pointer to the owning Klass or to a specific Interface implemented by the owner followed by a list of code addresses. Each code address identifies a method entry point (or stub) used to execute a virtual/interface method call. Private methods are omitted from a class’s vtable because they can only ever be invoked directly. Final methods which do not override a parent method can also be omitted for the same reason. So, the size of a vtable is proportional to the number of non-private, non-final methods declared in or inherited by a class.

itables contain pointers to the code which implements methods of a specific interface. An itable has to have an entry for each method in the class. That may be a default method or, in the case of an abstract class, a pseudo-method which fails if it is ever called.

A class may implement multiple interfaces so it may include multiple itables. The itable segment is prefixed with a directory consisting of pairs of Klass pointers and offsets, one for each implemented interface. The code implementing an interface invocation walks the directory to find the correct table and then indexes it it to find the required implementation method.

So, the itable segment size is proportional to the number of methods in all N implemented interfaces plus N * 2 index entry words.

annotations

Annotations are embedded in a class file in two separate regions and this split is also reflected in the in-memory representation. Symbol text for annotations names, type names and field names and for Class and String literals supplied in annotation field values is stored in the text area of the classfile and indexed from the file’s constant pool. Numeric values for annotation fields are also located in the constant pool. By contrast. the target and structure of each annotation is described in attribute records attached to the various standard classfile entries that define the class, per se, class members and properties of those members.

The association of each attribute record with a corresponding classfile entry determines the annotation target (e.g. class, field, method, type parameter, method parameter, local var etc). The records contents consist of a mix of tags describing the annotation format and constant pool indices identifying the relevant annotation names, type and field names and values. Annotations with multiple values embed multiple constant pool indices to identify tose values. Annotation fields whose value is another annotation or list of annotations recursively embed similar annotation definitions.

A similar split is used when modelling annotations as C++ metadata. Symbols are stored in the System Dictionary, Strings are allocated on the Java heap and both are referenced from the constant pool along with numeric values embedded ito constant pool slots. Annotation metadata itself is encoded as byte vectors (C++ type AnnotationArray = Array<u1>), which include type tags and constant pool offsets, identifying the annotation name, layout and values, or as sequences of such byte vectors (C++ type Array<AnnotationArray*>).

Class- and field-level annotations are embedded within an enclosing object (C++ type Annotations) linked from the owning InstanceKlass.

```
Annotations (32 bytes)
+--------------------------------------------------+
| class_annotations: AnnotationArray*              |
| field_annotations: Array<AnnotationArray*>       |
| class_type_annotations: AnnotationArray*         |
| field_type_annotations: Array<AnnotationArray*> |
+--------------------------------------------------+
```

Method-level annotations for method, parameter, type and definitions of annotation default values are referenced (as Array<u1>*) from the associated ConstMethod object (see below).

The annotations column in the jcmd display counts storage used for all these different types of structural metadata but does not include the cost of associated constant pool entries (see below) needed to store references to Symbols and Strings or numeric constant values. As mentioned previously, Symbol costs are not accounted for in the class stats diagnostic.

CpAll

This column combines storage for several data structures used to model the constant pool data loaded from a class's bytecode. Mostly this is just the pool itself, a block of memory with slots holding numeric values, references to Objects, Strings and Symbols or indices which indirect to related pool entries, plus the tags array which identifies what type of data is resides in each pool slot. However, there a few more auxiliary data structures required.

The most important auxiliary structure is the resolved_references array (a Java Object[] allocated on the Java heap). Object references are not actually stored directly in the pool slot. They are installed in the resolved_references array with the corresponding slot storing an index into that array. This allows values to be created and installed on demand and makes it simple for the garbage collector to find and mark the referenced objects.

Note that per-class constant pool costs do not account for storage required to allocate Symbols in the System dictionary or Strings on the Java heap. They merely account for the storage needed to store references to those objects.

The main unit of allocation used to store constant pool metadata is a variable length instance of (C++) class ConstantPool.

```
ConstantPool (80 bytes + N * 8 bytes)
+--------------------------------------+
| tags: Array<u1>                      |
| pool_holder: InstanceKlass*          |
| resolved_references: oop             |
| reference_map: Array<u2>             |
| flags: int                           |
| length: int                          |
| cache: ConstantPoolCache*            |
| . . .                                |
+--------------------------------------+
| pool[0]                              |
| . . .                                |
| pool[length-1]                       |
+--------------------------------------+
```

`tags` is allocated as a byte vector separate from the pool to minimise wasted space. It is included in the CPAll statistic.

`length` identifies the number of pool entries and the length of the tags array. It is packed into a long field with a few flags recording the pool state.

`pool_holder` is a backpointer to the InstanceKlass this ConstantPool belongs to.

`resolved_references` is the Java heap Object[] storing objects referenced from the class pool. The pool slot is updated with an integer index into the Object[] when the array entry is populated with an Object.

`reference_map` allows the index for an entry in resolved_references to be mapped back to the index of the pool/tag slot which references it.

Lastly, the constant pool field `cache` references another structure, an instance of C++ class ConstantPoolCache. This class /does not/ do what it says on the tin; it /is/ a cache but not for constant pool entries. The ConstantPoolCache for a given class is used by the (runtime JITted) Java interpreter to provide a quick path to the ConstantPool and also to translate bytecode operands that represent constant pool entries for classes, methods, field names and resolved Object references to the corresponding resolved class, method, field or Object.

```
ConstantPoolCache (16 bytes + N * 32 bytes)
+--------------------------------------+
| length: int                          |
| constant_pool: ConstantPool*         |
+--------------------------------------+
| cache_entry[0]                       |
| . . .                                |
| cache_entry[length-1]                |
+--------------------------------------+
```

The basic structure holds a `length` count for the number of cache entries and a back pointer to the owning constant pool. The latter can be used to access Objects located in the resolved_references array or numeric constants located in CP slots. A variable length tail stores an array of entries detailing resolved fields or methods of the class.

```
ConstantPoolCacheEntry (32 bytes)
+--------------------------------------+
| indices : long                       |
| f1: Metadata*                        |
| f2: long                             |
| flags: long                          |
+--------------------------------------+
```

indices packs the CP slot index for the member in question with info determining whether the member has been resolved (resolution may necessitate loading related classes, updating a vtable etc).

f1 may point to a Method or field-owner Klass

f2 may hold a vtable index or resolved_references index or a direct pointer to a final method

flags identifies the type of entry (field or method) and the various member attributes (static, final, volatile etc) and also stores a field index or method parameter count

The interpreter installs a pointer to the current method owner's ConstantPoolCache in a dedicated register when it begins a method call and restores the previous method owner's ConstantPoolCache when it executes a return. The first time a constant pool index is encountered in method bytecode which references an Object or identifies a method/field the interpreter must make an expensive call out to the JVM to resolve the Object reference or method/field. The callout will create and install any required Object in resolved_references. If a field or member is being referenced then it will load whatever classes are required to allow the access or invoke to proceed and install the relevant details of the method/field into the ConstantPoolCache. At subsequent encounters the interpreter can simply use the value stored in the ConstantPoolCache. Resolution may be attempted in parallel so a locking strategy must be used to manage updates to the ConstantPoolCache.

MethodAll

This column identifies all the storage used to track information about all the methods of a given class i.e. it is aggregated data. So, when looking at this figure it is critical to divide the amount of storage used by the value in column MethodCount in order to obtain the per method overheads. Of course, that division operation that does not diminish the impact of the aggregate figures on footprint.

It is still important to consider carefully why any class might need a large method count and whether it might be better to refactor a class to declare less methods. Also, the average figure may be relatively low but specific methods might still be responsible for a large amount of the cost. Contrary to the obvious expectation the bulk of the method storage is rarely due to method bytecode -- bytecode provides a very compact representation for code instructions.

Most of the details needed to keep track of a method are split into three separate linked C++ instances. The primary class, Method, holds runtime execution info for the method

```
Method (88 bytes)
+--------------------------------------+
| code: nmethod*                       |
| i2i_entry: address                   |
| from_interpreted_entry: address      |
| from_compiled_entry: address         |
| adapter: AdapterHandlerEntry         |
| vtable_index: int                    |
| flags: int                           |
| constMethod: ConstMethod*            |
| method_data: MethodData*             |
| method_counters: MethodCounters*     |
+--------------------------------------+
```

`code` points to the latest JIT-compiled code for the method. The memory it points to is accounted for under the NMT Code statistics.

The 4 fields from `i2i_entry` to `adapter` identify 4 different code entry points for the method, called for different transitions from/to interpreted/compiled code or method handles.

The method's vtable index is packed into a long word with flags identifying various method properties.

The last 3 fields, `constMethod` (sic), `method_data` and `method_counters` are pointers to auxiliary structures whose storage is summed into the MethodAll stat.

Method objects are normally fixed size. A Method instance that represents a native method will also include extra storage holding a few words of data describing how to relocate register and stack arguments from Java to C++ calling convention.

A MethodCounters instance is a fixed size object (32 bytes) which stores a few simple numeric stats appropriate to all methods, such as invocation count, throw_out count, call/loop backedge execution rate, compile level etc. More complex stats specific to the method definition are stored in the MethodData instance (see below).

A ConstMethod stores runtime-invariant method info that may be needed at runtime.

```
ConstMethod (48 bytes + embedded tables)
+--------------------------------------+
| constants: ConstantPool*             |
| constMethod_size: int                |
| flags : u2                           |
| result_type: u1                      |
| code_size: u2                        |
| name_index: u2                       |
| sig_index: u2                        |
| method_idnum: u2                     |
| max_stack: u2                        |
| max_locals: u2                       |
| size_of_parameters: u2               |
| orig_method_idnum: u2                |
| . . .                                |
+--------------------------------------+
| . . .                                |
+--------------------------------------+
```

- a back pointer to the owner class's constant pool
- a method stackmap (encoded in an Array<u1>*) which allows the layout of method stack frames to be identified at every point where the method might pause to allow a GC
- the CP index of the method's name and signature Symbols
- various simple numeric fields: access flags, bytecode size, parameter count, max stack depth, local count, result basic type (either object or one of the primitive types), etc.
- the total allocation size for the ConstMethod including the variant tail

A variable length tail allocated beyond the end of a ConstMethod is used to embed the method's bytecode plus a variety of tables of information which may or may not be present depending upon how the method is defined and how it is compiled. Optional data includes

- local variable table, present if local variables were compiled into the bytecode (this table merges in details of local variable generic types when local variable type tables are present in the bytecode)
- compressed line number table, present if line numbers were compiled into the bytecode
- exception table, present if the bytecode includes try catch regions (this is visible in javap -c output)
- checked exceptions, present if the method declares checked exceptions
- cp index of method generic signature, if the method is generic
- pointers to annotations data (Array<u1>*) for method, parameter, type annotations and annotation default definitions (only if present in the original method)

Each embedded table is succeeded by a length field. Table length field addresses and table start addresses are computed in reverse by subtracting table lengths from the end address of the allocated block.

A MethodData instance holds profile statistics gathered during execution of the method in both the interpreter and in C1-profiled compiled code. Unlike MethodCounters these stats are specific to particular bytecode operations that occur in the method.

```
MethodData (264 bytes + embedded profile data)
+-------------------------------------+
| method:Method*                      |
| size: int                           |
| extra_data_lock: Mutex              |
| nof_decompiles: u4                  |
| nof_overflow_recompiles: u4         |
| nof_overflow_traps: u4              |
| trap_hist: u1[20]                   |
| invocation_count: InvocationCounter |
| backedge_count: InvocationCounter   |
| . . .                               |
+-------------------------------------+
| . . .                               |
+-------------------------------------+
```

The MethodData instance includes a variety of fixed fields which precede a variable length tail that includes profile data that is specific to the structure of the associated method code. Fields include.

- a back pointer to the Method
- the allocation size of the MethodData

- numeric stats counting traps, decompiles and recompiles
- a trap history buffer
- counters to track invocations of the method and loop iterations in the method code

The tail embeds a variety of different types of stats records. The majority of them are associated with entry points, branch points (if/else, loop back edge, switches), static, direct, virtual and interface calls and method returns. Other stats record execution of traps planted by the JIT which handle things like null dereferences, omitted (cold) branches, and speculative assumptions about arguments and value types etc.

A MethodData instance is created on demand during interpretation when invocations of a method exceed an associated threshold or when a trap requires stats to be recorded. MethodData instances may be updated when a method is compiled and/or recompiled. Some of these stats are omitted at certain compilation levels and others are only needed when the compiler strategy involves certain types of speculative optimization. So, the size of a MethodData instance is partially determined by code structure and partly by decisions made based on runtime feedback.

Profile and deopt stats guide both initial and repeat compilation. Profile stats slowly guide the compiler towards selecting the hottest paths through the code. Deopt stats guide compiler choices. For example if the compiler always observes a specific value type it could plant a type check guard with a trap on fail then generate code only for the expected type or it can generate code for all possible types; likewise, the compiler could explicitly test for null and branch to handle it or instead just dereference and catch (rare) null dereferences in the signal handler.