Java Class Metadata: A User Guide

Andrew Dinn
Red Hat OpenJDK Team

February 2018

Motivation

~~Native Memory Stats~~

Class Metadata Memory Stats

What Does the Metadata Model Look Like?

Optimizing Class Metadata

Questions

Motivation

What is Java Class Metadata?

- It is *not* bytecode
  - ok, it is the JVM’s *internal model* of everything in the bytecode . . . and more
- Metadata unpacks the bytecode and mixes in . . .
  - Resolution state
    - linkage to other classes/interfaces, fields, methods, constants
  - Interpretation state
    - quick access cache for resolved references
    - basic profile counters
  - Compilation state
    - code entry addresses, linking stubs (i2c, c2i)
    - sophisticated profile counters

Why Java Class Metadata?

- A managed runtime must model the code base at runtime
  - what gets loaded & linked is not defined in advance
  - dynamic linkage must also respect visibility and access
- Interpreter and JIT
  - must know about the class organization
    - supers, implemented interfaces, fields, methods, etc
  - referenced during interpretation, compilation and JIT code execution
- Reflection requires reification of the class model
  - also, method handle APIs reference and check the class/method base
- JVMTI agents
  - can query and update the class base at runtime

Why Not Just Bytecode?

- direct vs indirect access
  - access to info in bytecode requires a complex byte array traversal
  - Metadata employs an (optimized) web of linked records
- implicit vs explicit data
  - bytecode format leaves many values and relationships implicit
  - Metadata computes and caches info needed by runtime
- update in place
  - bytecode comes by the slab
  - with metadata runtime-derived info can be located where it is needed
- bytecode verbosity
  - per class copies of symbols and objects: Ljava/lang/Object, add, ()V, ""
  - Metadata ensures one unique symbol, or oop is shared by all

Why Should I Care*?

- It’s nice to know how this all works
  - yes, here at Red Hat we like to encourage new Hotspot devs
- Metadata *can* make a noticeable contribution to resident image size
  - especially fat apps running in sandboxes with small amounts of data
  - so beware drawing naive conclusions when benchmarking!
- Design decisions you make can incur or avoid Metadata costs
  - which is what I really want to show you in this talk

*apologies to Pete Townshend

~~Native Memory Stats~~

Native (aka Class And Other Metadata) Memory

- Native Memory system covers all JVM-internal data structures
  - JIT compiler data
  - Code cache
  - GC management data
  - Class model
  - Symbol Table
  - Threads
- As opposed to Heap Memory used for Java objects

Native Memory (2)

- The JVM has its own memory management subsystem
  - Employs Metaspaces – independently mapped Vmem regions
    - Alloc hierarchically: Chunk, Block
      - Free list management ensures very high occupancy
    - Also provides Arenas (wipe-and-restart alloc pools)
      - Used by JIT compiler phases
    - Metaspace virtual to physical map extended as needed
  - Metadata/Metaspace(C++) instances inherit alloc behaviour
    - `class MetaspaceObj`
    - `class Metadata: public MetaspaceObj`

Native Memory Stats

Start your program with NMT switched on:
(n.b. HelloWait prints Hello and waits for some input)

```
$ java -XX:+NativeMemoryTracking=summary HelloWait
Hello
enter: <cr>
```

Native Memory Stats (2)

Use jcmd to request NMT stats

```
$ jcmd -l
5553 HelloWait
$jcmd 5553 VM.native_memory summary
5553

Native Memory Tracking:

. . .
```

Native Memory Stats (3)

Use jcmd to request NMT stats

```
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)
  . . .
```

Class Metadata Memory Stats

Class Metadata Memory Stats

Start your program with -XX:+UnlockDiagnosticVMOptions

```
$ PREPEND_JAVA_OPTS=-XX:+UnlockDiagnosticVMOptions \
    <wildfly_dl_dir>/bin/standalone.sh
. . .
```

Class Metadata Memory Stats (2)

Use jcmd to request class metadata stats

```
$ jcmd -l
  13442 /home/adinn/jboss/wf/wildfly-10.1.0.Final
$ jcmd 13442 GC.class_stats
  13442:
Index Super InstBytes KlassBytes annotations    CpAll ...
    1    -1   5163352        480           0        0 ...
```

Per Class Metadata Memory Stats

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

Summary Class Metadata Memory Stats

Index	Super	InstBytes	KlassBytes	annotations	CpAll	MethodCount
	Bytecodes	MethodAll	ROAll	RWAll	Total	ClassName
10228	. . .					
	. . .					
		22063568	6336992	55256	18873504	93169
	3664807	23219712	16172528	35387024	51559552	
			12.3%	0.1%	36.6%	-
	7.1%	45.0%	31.4%	68.6%	100.0%	

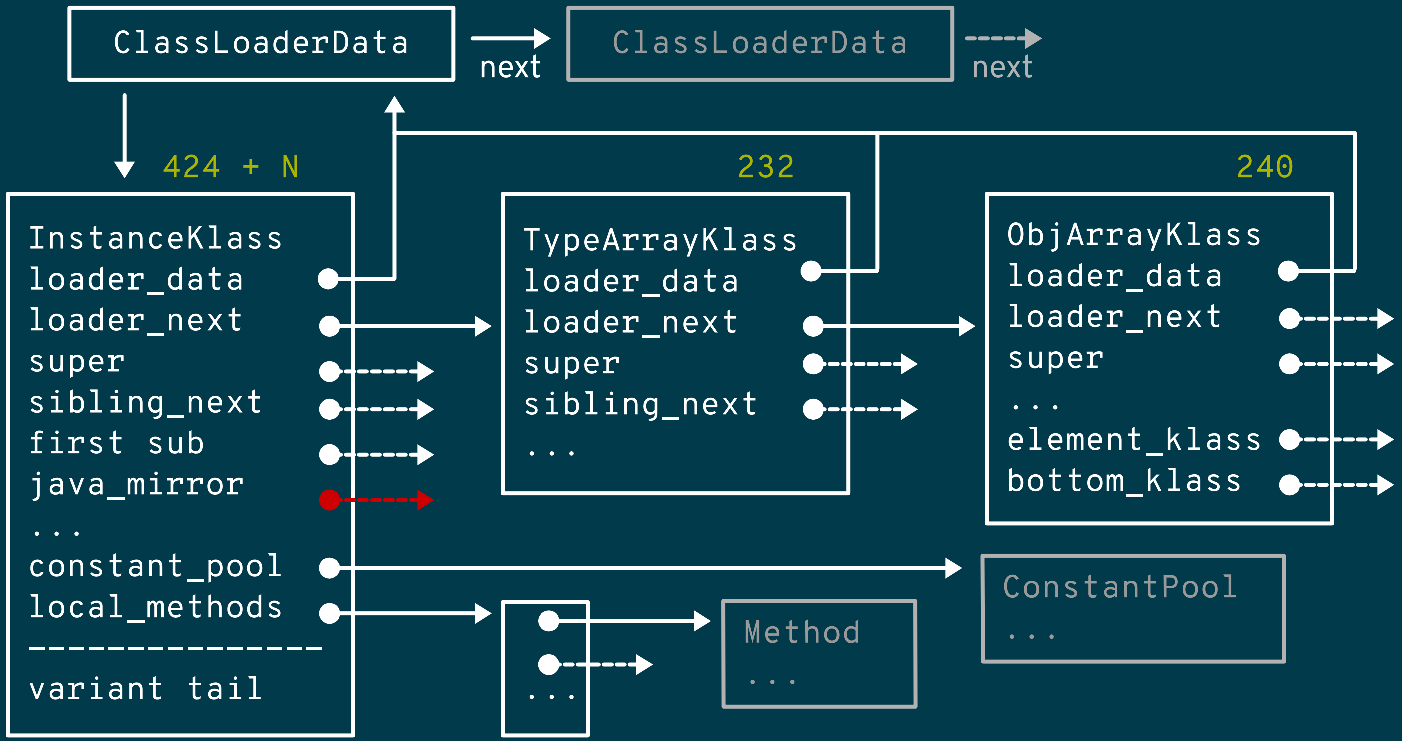
Summary Stats Breakdown

- Methods 45%:
  - comprising 93,000 methods/23,750,000 bytes
  - i.e. roughly 9 methods per class and 250 bytes per method
- CpAll 31%:
  - comprising 10,200 constant pools/18,840,000 bytes
  - i.e. roughly 1850 bytes per pool or 200 x 8-byte pool entries + 1-byte tags
- KlassBytes 12%:
  - comprising 10,200 classes/6,340,000 bytes
  - i.e. roughly 620 bytes per class
- ByteCodes 7%:
  - 8,990,000 bytes, giving an average of 40 bytes per method

What Does the Metadata Model Look Like?

What Do Those Stats Really Mean?

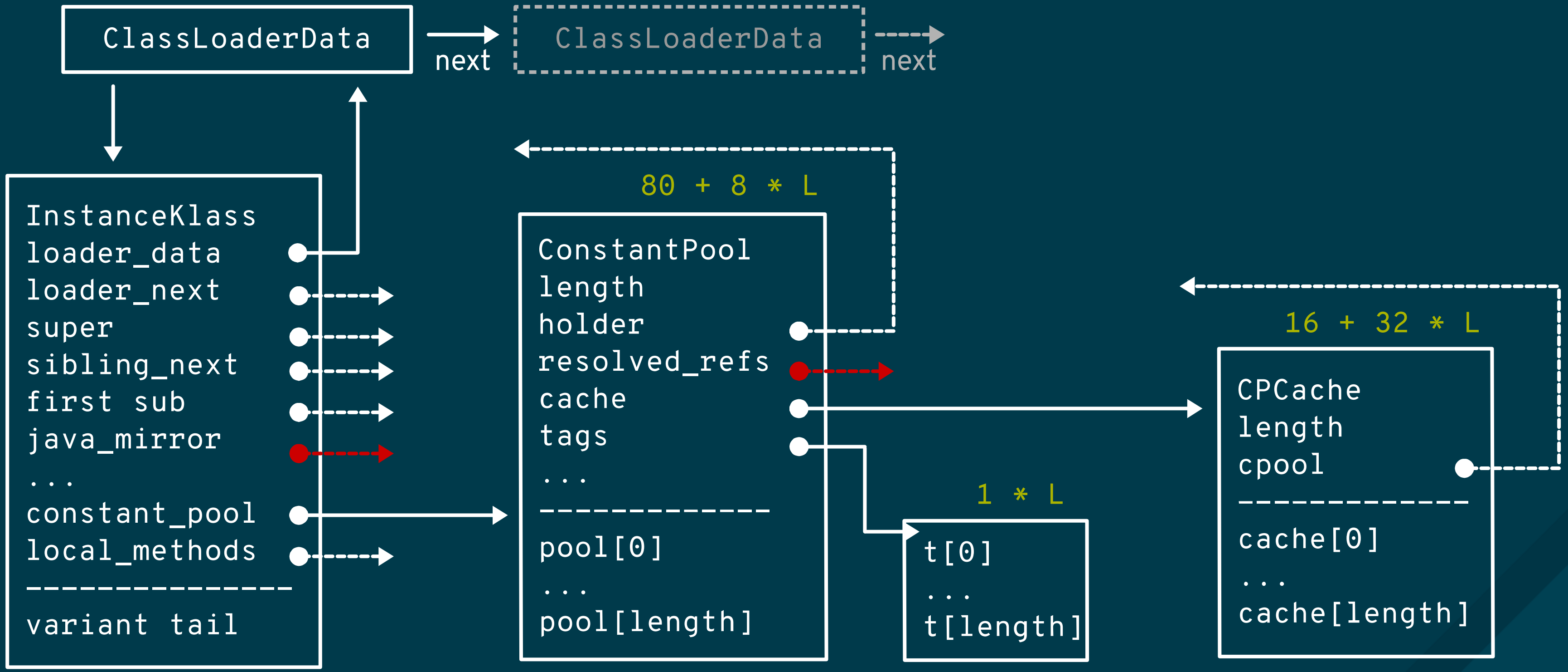
Metaspace Constant Pool Objects



Metaspace InstanceKlass Variant Tail

- OopMap
  - Array of offsets to all object fields in any instance
  - Not present for interfaces
- VTable
  - block of pointers to code entry addresses needed for virtual methods
  - No entry needed if method is non-public or is locally introduced and final
- ITables
  - Chain of blocks of pointers to code entry addresses needed for interface methods
  - A block per implemented interface, preceded by Klass* and size

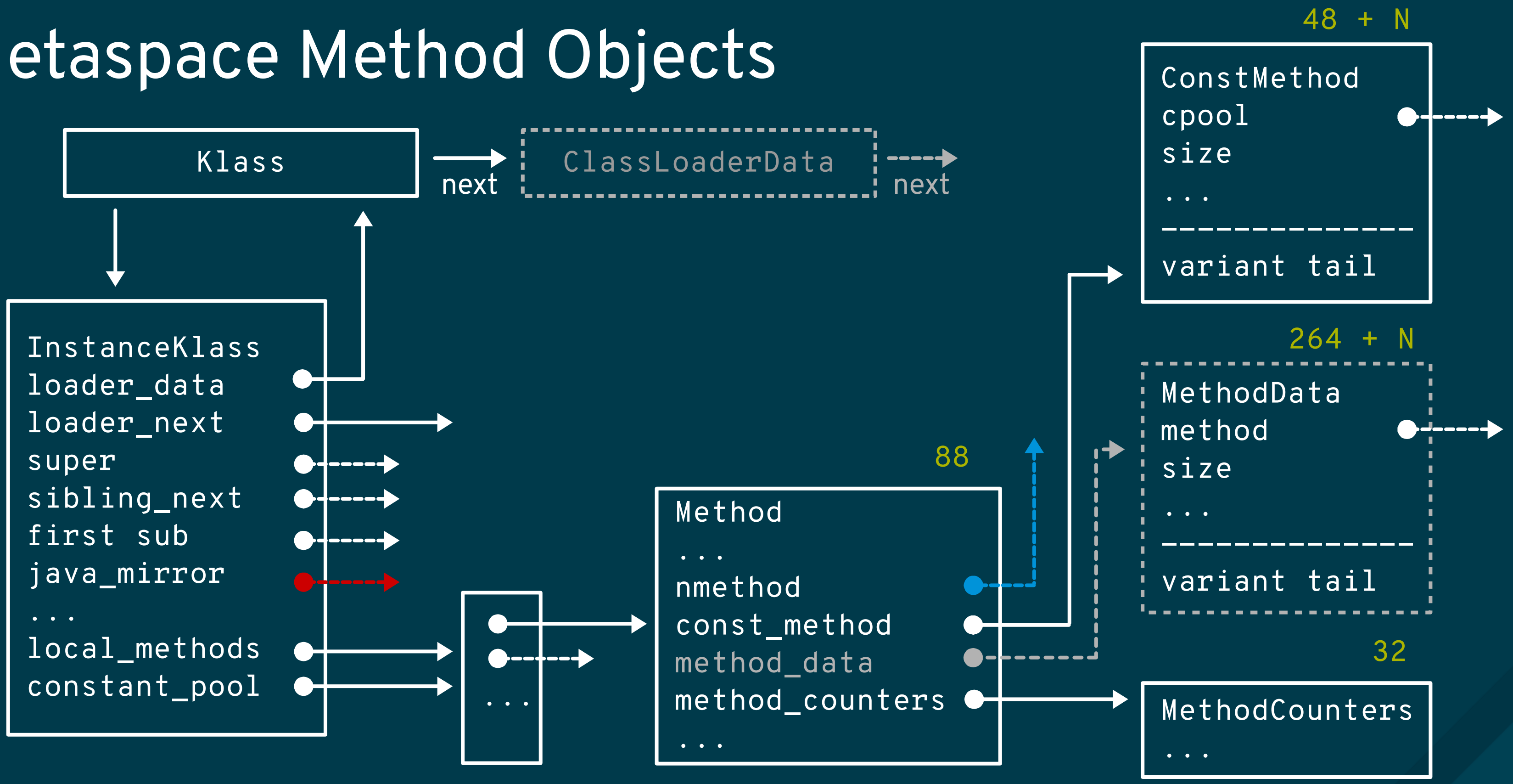
Metaspace Klass & Constant Pool Objects



Metaspace Constant Pool & Cache Variant Tail

- ConstantPool
  - 8 bytes per entry, for numeric constants, pointers to Symbols, Class, Method etc
  - 1 byte tag per 8 byte entry
  - refs array (Object[]) on heap holds object references
- ConstantPoolCache
  - not what it says on the box
  - resolution state for intepreter
  - for references to (this or other) classes, methods and fields

Metaspace Method Objects



Metaspace Method Objects Variant Tail

- ConstMethod
  - local var table
  - line number table
  - exception table
  - pointer to method annotations (ubyte[]) array
  - method bytecode
  - . . .
- MethodData
  - variety of different counters used by C1 and C2 compiler
  - contents and size entirely depends on method complexity (#calls, #branches, etc)
  - only allocated when method is JIT-compiled

Optimizing Class Metadata

What Do Those Stats Really Mean?

EAP Stats sorted by KlassBytes

Index	Super	InstBytes	**KlassBytes**	annotations	CpAll	MethodCount
	Bytecodes	MethodAll	ROAll	RWAll	Total	ClassName
1422	8851	80	11320	0	52376	950
	20406	194104	120952	152320	273272	EjbLogger_$logger
1708	8851	64	10576	0	46752	857
	16746	185832	107120	150024	257144	ControllerLogger_$logger
492	8851	64	9336	0	38512	702
	11650	135792	81776	113368	195144	InfinispanLog_$logger

Generated Logger Design

- Interface class Logger implemented by generated class Logger_$logger

  -
    ```
    public interface EjbLogger extends BasicLogger {
      . . .
      @LogMessage(level = ERROR)
      @Message(id = 4, value = "failed to get tx manager status; ignoring")
      void getTxManagerStatusFailed(@Cause Throwable cause);
      . . .
    ```
  - EJBLogger_$logger extends BasicDefaultLogger 2 x itable, 2 x vtable
- Generated implementation calls calls public method to retrieve value string

  -
    ```
    void getTxManagerStatusFailed(@Cause Throwable cause) {
      error(getTxManagerStatusFailed$str(), cause)
     }
    ```
  - Doubles method count (n.b. meant to allow message translation by overriding)

Improved Logger Design

- Make Logger a class not an interface (with empty log methods)
  - Generate actual class as a replacement for Logger
  - Avoids large itable for Logger class
- Make logging methods final
  - Avoids need for vtable
- Inject format strings as literals in generated code
  - Alternatively, use message table loaded as a resource
  - Halves method count of generated class

Thank You

Questions

Slides and Detailed Blog Articles available at

http://github.com/adinn/fosdem2018