Why does my Oracle JVM create all these objects for a simple 'Hello World' program?

Asked 2 years, 7 months ago Active 2 years, 6 months ago Viewed 912 times

9

6

I was playing around with `jmap` and found that simple "Hello World" Java program creates thousands of objects. Here is truncated list of objects **Oracle JVM update 131** creates on startup:

```
 num     #instances         #bytes  class name
----------------------------------------------
   1:           402        4903520  [I
   2:          1621         158344  [C
   3:           455          52056  java.lang.Class
   4:           194          49728  [B
   5:          1263          30312  java.lang.String
   6:           515          26088  [Ljava.lang.Object;
   7:           115           8280  java.lang.reflect.Field
   8:           258           4128  java.lang.Integer
   9:            94           3760  java.lang.ref.SoftReference
  10:           116           3712  java.util.Hashtable$Entry
  11:           126           3024  java.lang.StringBuilder
  12:             8           3008  java.lang.Thread
  13:            74           2576  [Ljava.lang.String;
  14:            61           1952  java.io.File
  15:            38           1824  sun.util.locale.LocaleObjectCache$CacheEntry
  16:            12           1760  [Ljava.util.Hashtable$Entry;
  17:            53           1696  java.util.concurrent.ConcurrentHashMap$Node
  18:            23           1472  java.net.URL
  19:            14           1120  [S
  20:             2           1064  [Ljava.lang.invoke.MethodHandle;
  21:             1           1040  [Ljava.lang.Integer;
  22:            26           1040  java.io.ObjectStreamField
  23:            12           1024  [Ljava.util.HashMap$Node;
  24:            30            960  java.util.HashMap$Node
  25:            20            800  sun.util.locale.BaseLocale$Key
```

I know that the JVM loads classes from JAR files and expect to see `java.lang.Class` , `java.lang.String` and `[Ljava.lang.Object` . 258 `java.lang.Integer` objects are clear tp me too: this is the `Integer` cache.

But `java.lang.reflect.Field` ? `Hashtable` ? Many `StringBuilder` s? `java.util.concurrent.ConcurrentHashMap` ? Where does this come from?

The program is pretty simple:

```
public class Test {
    public static void main(String[] args) throws IOException {
        System.out.println("Hello world");
        System.in.read();
```

```
    }
}
```

JVM details:

```
java version "1.8.0_131"
Java(TM) SE Runtime Environment (build 1.8.0_131-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.131-b11, mixed mode)
```

Ubuntu 16.04.

java jvm

edited Jun 2 '17 at 16:29

asked Jun 2 '17 at 16:12

Dmitriy Dumanskiy
5,585 4 22 45

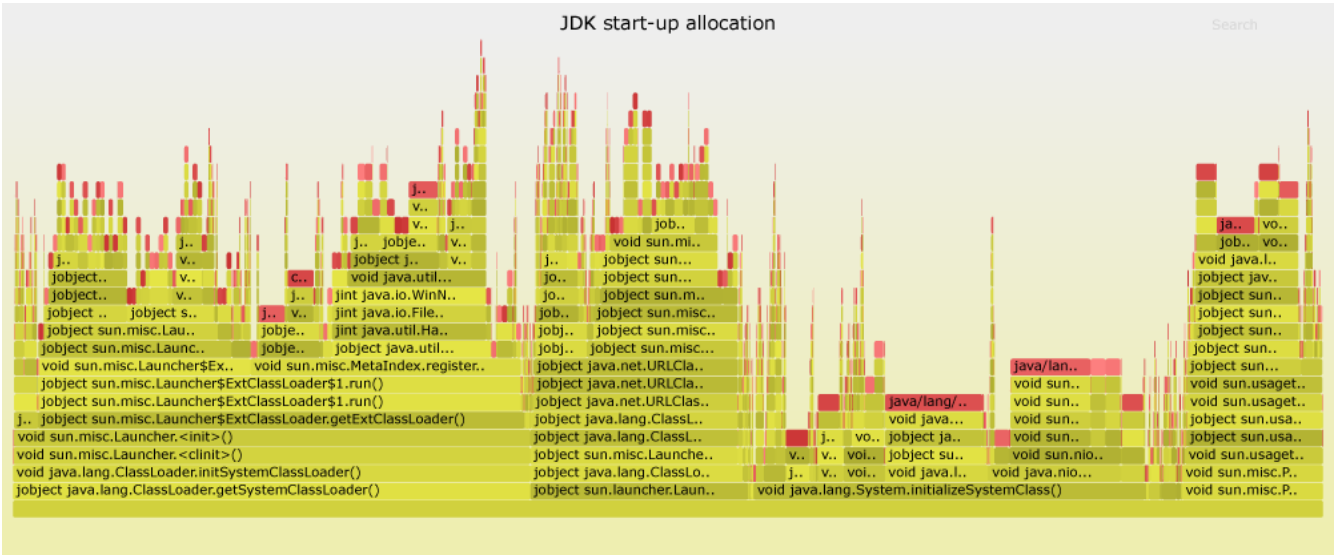
A java.lang.Class has java.lang.Fields, so why wouldn't there be Field instances? Hashtable is used a lot internally in places you wouldn't inutitively expect it. Since the program is so simple, the garbage collector probably hasn't even run once, so you see *every* single object created during VM startup. – Durandal Jun 2 '17 at 17:37

3 Answers

22

You can find the answer yourself by running the application with `-XX:+TraceBytecodes` flag. This flag is available in debug builds of HotSpot JVM.

Here is the detailed Flame Graph (clickable SVG) showing the stack traces where the allocated objects come from.



In my case the main sources of start-up allocations were

- URLClassLoader and Extension ClassLoader

- Locale cache
- UsageTrackerClient
- MetaIndex registry
- System Properties
- Charset initialization

P.S. The script used to generate the Flame Graph from TraceBytecodes output.

edited Jun 3 '17 at 22:06

answered Jun 3 '17 at 4:04

apangin
61.6k 9 130 164

2 Wow this is a cool histogram. A bit less informative (but still gives some glue about cause by observing order) is to use `-verbose:class` . – eckes Jun 3 '17 at 8:24

Thanks again! This answer is much better. – Dmitriy Dumanskiy Jun 3 '17 at 8:43

1 BTW: I think one reason why the Field constructor does not show up in the ByteCode traces is because they are generated in native code by Class#getDeclaredFields0(). – eckes Jun 14 '17 at 15:47

1 @eckes This is true. Objects allocated internally by JVM can be traced using JVM TI VMObjectAlloc event. I've omitted this for simplicity, since `getDeclaredFields0` already gives an idea where those objects come from. – apangin Jun 14 '17 at 15:51

I've summarized the ways to trace object allocation in this answer. – apangin Jun 14 '17 at 15:53

3

There are a lot of maintenance data structures. E.g. every initialized JVM has these system properties, which is a subtype of `Hashtable` , hence, explains the `Hashtable.Entry` instances.

Also, core classes like `java.lang.Character` know the Unicode properties of all characters, also, you see `Locale` -specific classes in you stats, as these have to be properly initialized at startup. What makes these examples so interesting, is, that they are loading these information from files or embedded resources, so their initialization involves I/O and caching mechanisms, whose artifacts you see in your output.

Also, other objects created during the startup process might not have been garbage collected yet. There are a lot of operations, like processing the class path and the jar files specified by it or parsing the command line options, which are more complex than the “Hello World” program that will be executed at the end. Mind that you can create a heap dump instead of just a histogram, so you can see who is holding a reference to the existing objects.

edited Jun 6 '17 at 9:04

answered Jun 2 '17 at 18:03

Holger
194k 25 285 534

Thanks. Isn't Hashtable outdated dozens of years ago :)? I wonder why it is still used? – Dmitriy Dumanskiy Jun 2 '17 at 18:09

2 That's compatibility. `System.getProperties()` has been declared with a return type of `java.util.Properties`, which is a subclass of `java.util.Hashtable`. Neither, the return type nor the class hierarchy, can be changed without breaking backwards compatibility. It's not the only place with such a legacy. – Holger Jun 2 '17 at 18:24

Your point about `jmap` is strange. Only client side of `jmap` is implemented in Java - it runs in a different process and does not affect classes loaded in the target JVM. On the target side it uses AttachListener / DiagnosticCommand APIs of HotSpot JVM. The code which collects and prints the class histogram is written in C++ and does not create a single Java object. – apangin Jun 3 '17 at 0:49

1 @eckes Dynamic Attach Mechanism is a built-in JVM feature, it has nothing to do with RMI. – apangin Jun 3 '17 at 4:36

1 @eckes: I can avoid the attach by using a local JMX bean, but that may still load JMX related classes which weren't loaded in an ordinary startup. By the way, you're mixing up `jmap` and `jstack`. – Holger Jun 6 '17 at 8:47

Checking if tools load additional classes

2

I tried the following program:

```
package test;
public class MainSleep {
    public static void main(String[] args) throws InterruptedException {
        synchronized (MainSleep.class) {
            MainSleep.class.wait(5*1000);
        }
    }
}
```

When I run it with:

```
"c:\Program Files\Java\jdk1.8.0_131\bin\java" \
  -verbose:class -cp target\classes test.MainSleep
```

I get verbose class loading messages, then a 5 sec pause and then the shutdown does load even more classes:

```
...
[Loaded sun.misc.PerfCounter from c:\Program Files\Java\jdk1.8.0_131\jre\lib\rt.jar]
[Loaded sun.misc.Perf$GetPerfAction from c:\Program
Files\Java\jdk1.8.0_131\jre\lib\rt.jar]
[Loaded sun.misc.Perf from c:\Program Files\Java\jdk1.8.0_131\jre\lib\rt.jar]
[Loaded sun.misc.PerfCounter$CoreCounters from c:\Program
Files\Java\jdk1.8.0_131\jre\lib\rt.jar]
[Loaded sun.nio.ch.DirectBuffer from c:\Program Files\Java\jdk1.8.0_131\jre\lib\rt.jar]
[Loaded java.nio.MappedByteBuffer from c:\Program
Files\Java\jdk1.8.0_131\jre\lib\rt.jar]
[Loaded java.nio.DirectByteBuffer from c:\Program
Files\Java\jdk1.8.0_131\jre\lib\rt.jar]
[Loaded java.nio.LongBuffer from c:\Program Files\Java\jdk1.8.0_131\jre\lib\rt.jar]
[Loaded java.nio.DirectLongBufferU from c:\Program
Files\Java\jdk1.8.0_131\jre\lib\rt.jar]
[Loaded java.security.PermissionCollection from c:\Program
```

```
Files\Java\jdk1.8.0_131\jre\lib\rt.jar]
[Loaded java.security.Permissions from c:\Program
Files\Java\jdk1.8.0_131\jre\lib\rt.jar]
[Loaded java.net.URLConnection from c:\Program Files\Java\jdk1.8.0_131\jre\lib\rt.jar]
[Loaded sun.net.www.URLConnection from c:\Program
Files\Java\jdk1.8.0_131\jre\lib\rt.jar]
[Loaded sun.net.www.protocol.file.FileURLConnection from c:\Program
Files\Java\jdk1.8.0_131\jre\lib\rt.jar]
[Loaded sun.net.www.MessageHeader from c:\Program
Files\Java\jdk1.8.0_131\jre\lib\rt.jar]
[Loaded java.io.FilePermission from c:\Program Files\Java\jdk1.8.0_131\jre\lib\rt.jar]
[Loaded java.io.FilePermission$1 from c:\Program Files\Java\jdk1.8.0_131\jre\lib\rt.jar]
[Loaded java.io.FilePermissionCollection from c:\Program
Files\Java\jdk1.8.0_131\jre\lib\rt.jar]
[Loaded java.security.AllPermission from c:\Program
Files\Java\jdk1.8.0_131\jre\lib\rt.jar]
[Loaded java.security.UnresolvedPermission from c:\Program
Files\Java\jdk1.8.0_131\jre\lib\rt.jar]
[Loaded java.security.BasicPermissionCollection from c:\Program
Files\Java\jdk1.8.0_131\jre\lib\rt.jar]
[Loaded test.MainSleep from file:/D:/ws/BIS65/test-java8/target/classes/]
[Loaded sun.launcher.LauncherHelper$FXHelper from c:\Program
Files\Java\jdk1.8.0_131\jre\lib\rt.jar]
[Loaded java.lang.Class$MethodArray from c:\Program
Files\Java\jdk1.8.0_131\jre\lib\rt.jar]
[Loaded java.lang.Void from c:\Program Files\Java\jdk1.8.0_131\jre\lib\rt.jar]
...
[Loaded java.lang.Shutdown from c:\Program Files\Java\jdk1.8.0_131\jre\lib\rt.jar]
[Loaded java.lang.Shutdown$Lock from c:\Program Files\Java\jdk1.8.0_131\jre\lib\rt.jar]
```

So this would be the baseline. When I now use `jstack` or `jmap` on that file and check for verbose class loading messages I can see if it introduces new classes (not sure about instances of course).

With `jstack -l` and `jstack` , the following additional one class is loaded:

```
[Loaded java.lang.Class$MethodArray from
[Loaded java.lang.Void from
[Loaded java.util.concurrent.locks.AbstractOwnableSynchronizer ...
[Loaded java.lang.Shutdown from
[Loaded java.lang.Shutdown$Lock from
```

With `jstack -F` or `jstack -m` no(!) additional class is loaded:

```
[Loaded java.lang.Class$MethodArray from
[Loaded java.lang.Void from
[Loaded java.lang.Shutdown from
[Loaded java.lang.Shutdown$Lock from
```

None of `jmap -clstat` , `-finalizerinfo` , `-heap` , `-histo` or `-histo:live` loaded additional classes:

```
[Loaded java.lang.Class$MethodArray from
[Loaded java.lang.Void from
[Loaded java.lang.Shutdown from
[Loaded java.lang.Shutdown$Lock from
```

The same is true for `jmap -dump:format=b,file=ignore.hprof` with and without the -F option as well as with and without the `live` flag.

Just for completeness, if I use *jvisualvm* or *jconsole* it will always trigger a lot of JMX class loads for thread, heap and application snapshots. Most likely because it always opens the dashboard for the process.

Exploring Heap Content

So now that we have established this I took a look at the `jmap -dump:format=b` (non-live, non-forced) heap dump with MAT, looking for the Fields you have been interested in:

The MAT **unreachable objects histogram** (which shows instances found in the heap but not connected to any GC root, which is basically all not-yet collected garbage) has 3038 objects, and the top 10:

Class Name	Objects	Shallow Heap
char[]	1.026	113.848
java.lang.String	599	14.376
int[]	423	7.664
java.lang.Object[]	220	14.192
java.lang.StringBuilder	137	3.288
java.lang.reflect.Field	115	8.280
java.lang.ProcessEnvironment$CheckedEntry	66	1.056
java.io.File	59	1.888
java.lang.Class	32	0
java.lang.StringBuffer	30	720

There is currently no single live `Field` instance visible with MAT and only very limited of `Class` instances. This looks much like a .hprof or MAT problem: the Class instances seems to not show any of their fields in the heap dump. I think they should be softly(!) referenced by `Class#reflectionData : SoftReference<ReflectionData<T>>` , but I think this should be visible in the heap dump and not losing 115 fields. (There is no `Class$ReflectionData` in the live heap and 14 `Class$ReflectionData` in the unreachable histo. That can fit well with 115 Fields.

(I guess I will check back with Serviceability-dev@openjdk on that. This does not fit in a comment, so this is an incomplete answer but I intend to enhance it).

edited Jun 14 '17 at 15:47

answered Jun 6 '17 at 18:35

eckes

8,527 1 45 62

That's indeed very interesting. Perhaps, it is connected to the fact that it only reports 32 `java.lang.Class` instances instead of >400… – Holger Jun 8 '17 at 9:25

Started Thread here: mail.openjdk.java.net/pipermail/serviceability-dev/2017-June/… – eckes Jun 14 '17 at 15:43