CodeHeap State Analytics (OpenJDK)

Preface

This documentation is intended for skilled JVM developers or support engineers. It is by no means a full, and generally understandable, description of all features and properties mentioned herein.

Basics

The Java Virtual Machine (JVM for short) generates a lot of native code for various stubs and adapters and, particularly, for the many Java methods which become Just-In-Time (JIT) compiled over time.

The class CodeHeap is the base class providing storage for the generated native code and functions necessary for space management. Each chunk of storage provided to a requestor forms a (new) CodeHeap block. Space is allocated from free or unused space in segment_size units (defined by the CodeCacheSegmentSize parameter). It is helpful, but not mandatory, to have the CodeCacheSegmentSize default value equal to the instruction cache line size. Currently, the default value for segment_size is 64 bytes (128 bytes on PPC, 256 bytes on s390x).

The class maintains a list of all existing blocks as well as a list of all free blocks. Adjacent free blocks are coalesced immediately.

General description

The CodeHeap state analytics are divided into two parts. The first part examines the entire CodeHeap and aggregates all information that is believed useful/important. The second part, which consists of several, independent steps, prints the previously collected information with emphasis on various aspects. Data collection and printing is done on an "on request" basis. While no request is being processed, there is no impact on performance. The CodeHeap state analytics do have some memory footprint. The "aggregate" step allocates some data structures to hold the aggregated information for later output. These data structures live until they are explicitly discarded (function "discard") or until the VM terminates. There is one exception: the function "all" does not leave any data structures allocated.

Requests for real-time, on-the-fly analysis can be issued via:

```
jcmd <pid> Compiler.CodeHeap_Analytics [<function>] [<granularity>]
```

If you are (only) interested in how the CodeHeap looks like after running a sample workload, you can use the command line option:

```
-Xlog:codecache=Trace
```

To see the CodeHeap state when a "CodeCache full" condition exists, start the VM with the command line option:

```
-Xlog:codecache=Debug
```

It will produce output only for the first time the condition is recognized. Both command line option variants produce output identical to the jcmd function:

```
jcmd <pid> Compiler.CodeHeap_Analytics all 4096
```

Currently, the parameters cannot be controlled via command line arguments.

Aggregation is necessary as the CodeHeap may become rather huge. Aggregation condenses the contents of a certain amount of CodeHeap space (a multiple of CodeHeap segment size, 4096 bytes by default) into an analysis granule. Such a granule contains enough detail to get initial insight while keeping the internal structure sizes in check. Printing everything with the finest granularity - which is the CodeHeap segment size - isn't a good idea to start with. First of all, the printed information might be overwhelming and the size of the output file might prove difficult to handle. Furthermore, the internal aggregation structures become almost as large as the CodeHeap itself.

The CodeHeap is a living thing and the JIT compiler and the method sweeper permanently modify its contents. Therefore, the aggregate must be collected under the CodeCache_lock. Creating a snapshot first and then analyzing/printing the snapshot in multiple ways assures consistent

views. Immediately after the aggregation is complete, the lock is released. That keeps the impact on "normal operation" (JIT compiler and sweeper activity) to a minimum. Under "normal operation" conditions, the lock should not be held for longer than one second. Even so, a slight impact on system behaviour could be observed. In particular, new JIT compilations are blocked while the snapshot is taken.

Function syntax

```
jcmd <pid> Compiler.CodeHeap_Analytics [<function>] [<granularity>]
```

<function> is one of

- all - all of the below, in that order. This is the default. Prints a lot!!
- aggregate - Examine the CodeHeap and remember the data in local structures. Must be run before any other subfunction can be requested.
- UsedSpace - print used (occupied) space information. Currently, this is a list of the 50 largest used blocks.
- FreeSpace - print free space information.
- MethodCount - print method count information.
- MethodSpace - print method space information.
- MethodAge - print age information, compilationID based, nMethods only.
- MethodNames - print name information, includes signature where available.
- discard - free all resources allocated by this function. Helps keep memory footprint low.

<granularity>

specifies the amount of storage space one aggregation element (one granule) represents. It is examined for the “aggregate” subfunction only. Since the “all” subfunction implies “aggregate”, it also accepts this parameter. The granularity cannot be smaller than the segment size of the CodeCache. The segment size can be set at JVM startup with the -XX:CodeCacheSegmentSize command line parameter. It has a platform-dependent default value which varies between 64 bytes and 256 bytes. The number of granules is artificially limited to (512*1024) with the only purpose of limiting the storage consumption of the internal aggregation structures. On the other hand, the granularity should be no greater than 1/256 of the used CodeCache space to provide meaningful output. There are other restrictions to the allowed values as well. It must be an integer multiple of the segment size and, due to implementation limitations, it can't be greater than 65535*(segment size). That poses an upper limit to granularity of at least 4096K-64bytes. The specified value is checked against all these limitations and silently adapted. The default value is 4096 bytes. Example: For a 1GB CodeHeap, the granule size must be at least 2048 bytes to not exceed the number of granules limit. On the other hand, a granule size of 4096kB would translate into just 256 granules for 1GB of heap space.

Output description

All output, except for the “MethodNames” and "FreeSpace" subfunctions, uses one character or a “bundle” of characters (let’s call both a print granule) to represent one aggregation granule. The granules are printed left to right, line by line, in address-ascending order. Each line is prefixed with the start address of that line and its offset (from CodeHeap begin).

The form of the output changes substantially if the granule size becomes equal to the segment size. In this special case (let's call it segment granules case), the space a granule represents can be either fully occupied or completely free. Consequentially, the module count is either one or zero. With that in mind, we can use the print granules to indicate the type of the module occupying the space. A legend is printed at the beginning of each flavor section which translates the one-character module-type encoding into a human-readable form.

Print granule flavors

Print granules come in different flavors, depending on the type of information each character of the print granule transports.

The “count” flavor

The print granule uses hexadecimal digits [1..f] to display a counted value. That value could be the number of modules (Blobs, Stubs, nMethods, …) contained in the associated aggregation granule. For easier visual distinction, empty [‘ ‘] granules and granules with a count higher than 15 [‘*’] are indicated with characters that are not hexadecimal digits. In the segment granules case, the output is identical to the "space" flavor output. With segment granules, there are no multi-character print granules for "count".

The “space” flavor

The print granule uses decimal digits [0..9] to indicate the occupation rate (or fill state) of the associated aggregation granule. For example, ‘2’ indicates an occupation range from 20% to 29.999%. For easier visual distinction, completely empty [‘ ‘] and completely filled [‘*’] granules are

indicated with characters that are not decimal digits. In the segment granules case, the output is identical to the "count" flavor output. With segment granules, there are no multi-character print granules for "space".

The “age” flavor

Because there is no compilation timestamp readily available, we have to resort to the compilation id to get an idea of the (relative) age of a module. The print granule uses decimal digits [0..8] to indicate the relative age of the youngest module in the granule. The age range that each digit stands for is calculated dynamically, based on the highest assigned compilation id. The ranges are kind of logarithmic, where ‘0’ indicates the youngest 1/256, ‘8’ indicates the oldest half.

Single character print granules

Here, one single character summarizes the contents of the associated aggregation granule, according to the granule flavor.

Multi character print granules

Here, a bundle of characters details the contents of the associated aggregation granule, according to the granule flavor. A bundle consists of two (tier1, tier2) or three (tier1, tier2, stubs+blobs) characters. The characters are separated by colons, and the bundles are separated by spaces.

UsedSpace output

This output is meant to identify insanely large blocks. Such blocks could possibly be generated by "runaway" JIT compilations, where excessive inlining blows up the code. The output is a sorted list (by size) of the top 50 blocks, showing (where available)

- - the start address of the block
- - the offset (relative to CodeCache begin)
- - the block length
- - the block type (what kind of code does it contain?)
- - information about the generating compiler (for nmethods)
- - the block name (for nmethods: method name and signature)

FreeSpace output

This output is meant to show how fragmented the code heap currently is. Future functions could rely on this analysis to take measures to increase the largest free block size.

The output consists of three parts. The first part is a list of all free blocks, that shows:

- - the start address
- - the length
- - the distance (gap) to the next free block
- - the number of (occupied) blocks between this and the next free block

The second part provides a TopTen list of the largest free blocks, with the same details as the first part.

The third part again prints a TopTen list of the largest free blocks. These free blocks do not actually exist. They are constructed from the current free block, its successor free block and all the occupied blocks in between these two. Only constructed blocks that do not contain any unmovable code are considered. In other words, they must only contain nMethods. The printed details are the same as with the first part.

MethodNames output

For each occupied block, its address, offset from the CodeHeap begin, and size (if available) is printed, followed by information about the block contents (see description of UsedSpace output). In cases where no method name is available (all but nmethods, stubs and blobs, for example), a more generic code block type is displayed. In the segment granules case, the block type is printed as well.