HPROF Agent

Contents

Overview

This document describes the JVM TI Agent HPROF delivered in the Java Development Kit (JDK). It is intended as demonstration code for JVM TI, and as a functional replacement for the older HPROF JVMPI Agent delivered in past releases.

Previous 1.4 and earlier releases of the JDK contained an HPROF agent built on the experimental JVMPI. The newer JVM TI replaces both JVMDI and JVMPI.

Note: Differences between this HPROF implementation and the older JVMPI based HPROF are marked in *RED ITALICS* throughout this document.

Start-up

HPROF is a simple profiler agent shipped with the JDK. It is a dynamically-linked library that interacts with the JVM TI and writes out profiling information either to a file or to a socket in ascii or binary format. This information can be further processed by a profiler front-end tool.

It is capable of presenting CPU usage, heap allocation statistics and monitor contention profiles. In addition it can also report complete heap dumps and states of all the monitors and threads in the Java virtual machine.

HPROF can be invoked by:

```
java -agentlib:hprof ToBeProfiledClass
```

Depending on the type of profiling requested, HPROF instructs the virtual machine to send it the relevant JVM TI events and processes the event data into profiling information. For example, the following command obtains the heap allocation profile:

```
java -agentlib:hprof=heap=sites ToBeProfiledClass
```

Following is the complete list of options that can passed to hprof :

```
java -agentlib:hprof=help

     HPROF: Heap and CPU Profiling Agent (JVMTI Demonstration Code)

hprof usage: java -agentlib:hprof=[help]|[<option>=<value>, ...]

Option Name and Value  Description                    Default
---------------------  -----------                    -------
heap=dump|sites|all    heap profiling                 all
cpu=samples|times|old  CPU usage                      off
monitor=y|n            monitor contention             n
format=a|b             text(txt) or binary output     a
file=<file>            write data to file             java.hprof[{.txt}]
net=<host>:<port>      send data over a socket        off
depth=<size>           stack trace depth              4
interval=<ms>          sample interval in ms          10
cutoff=<value>         output cutoff point            0.0001
lineno=y|n             line number in traces?         y
thread=y|n             thread in traces?              n
doe=y|n                dump on exit?                  y
msa=y|n                Solaris micro state accounting n
force=y|n              force output to <file>         y
verbose=y|n            print messages about dumps     y

Obsolete Options
----------------
gc_okay=y|n

Examples
--------
  - Get sample cpu information every 20 millisec, with a stack depth of 3:
      java -agentlib:hprof=cpu=samples,interval=20,depth=3 classname
  - Get heap usage information based on the allocation sites:
      java -agentlib:hprof=heap=sites classname

Notes
-----
  - The option format=b cannot be used with monitor=y.
  - The option format=b cannot be used with cpu=old|times.
  - Use of the -Xrunhprof interface can still be used, e.g.
       java -Xrunhprof:[help]|[<option>=<value>, ...]
    will behave exactly the same as:
       java -agentlib:hprof=[help]|[<option>=<value>, ...]

Warnings
--------
  - This is demonstration code for the JVMTI interface and use of BCI,
    it is not an official product or formal part of the JDK.
  - The -Xrunhprof interface will be removed in a future release.
  - The option format=b is considered experimental, this format may change
    in a future release.
```

By default, heap profiling information (sites and dump) is written out to java.hprof.txt (ascii). The monitor=y|n option has proven to be problematic and may be replaced with something more useful.

The output in most cases will contain ID's for traces, threads, objects, etc. Each type of ID will typically start with a different number than the other ID's, e.g. traces might start with 300000.

Note: *The gc_okay option is no longer supported.*

Heap Allocation Profiles (heap=sites)

Following is the heap allocation profile generated by running the Java compiler (`javac`) on a set of input files. Only parts of the profiler output are shown here.

```
Command used: javac -J-agentlib:hprof=heap=sites Hello.java

SITES BEGIN (ordered by live bytes) Fri Feb  6 13:13:42 2004
          percent          live          alloc'ed  stack class
 rank   self  accum     bytes objs     bytes  objs trace name
    1 44.13% 44.13%   1117360 13967  1117360 13967 301926 java.util.zip.ZipEntry
    2  8.83% 52.95%    223472 13967   223472 13967 301927 com.sun.tools.javac.util.List
    3  5.18% 58.13%    131088     1   131088     1 300996 byte[]
    4  5.18% 63.31%    131088     1   131088     1 300995 com.sun.tools.javac.util.Name[]
```

A crucial piece of information in heap profile is the amount of allocation that occurs in various parts of the program. The `SITES` record above tells us that 44.13% of the total space was allocated for java.util.zip.ZipEntry objects. Note that the amount of live data is only a fraction of the total allocation that has occurred at a given site; the rest has been garbage collected.

A good way to relate allocation sites to the source code is to record the dynamic stack traces that led to the heap allocation. Following is another part of the profiler output that illustrates the stack traces referred to by the four allocation sites in output shown above.

```
TRACE 301926:
        java.util.zip.ZipEntry.<init>(ZipEntry.java:101)
        java.util.zip.ZipFile+3.nextElement(ZipFile.java:417)
        com.sun.tools.javac.jvm.ClassReader.openArchive(ClassReader.java:1374)
        com.sun.tools.javac.jvm.ClassReader.list(ClassReader.java:1631)

TRACE 301927:
        com.sun.tools.javac.util.List.<init>(List.java:42)
        com.sun.tools.javac.util.List.<init>(List.java:50)
        com.sun.tools.javac.util.ListBuffer.append(ListBuffer.java:94)
        com.sun.tools.javac.jvm.ClassReader.openArchive(ClassReader.java:1374)

TRACE 300996:
        com.sun.tools.javac.util.Name$Table.<init>(Name.java:379)
        com.sun.tools.javac.util.Name$Table.<init>(Name.java:481)
        com.sun.tools.javac.util.Name$Table.make(Name.java:332)
        com.sun.tools.javac.util.Name$Table.instance(Name.java:349)

TRACE 300995:
        com.sun.tools.javac.util.Name$Table.<init>(Name.java:378)
        com.sun.tools.javac.util.Name$Table.<init>(Name.java:481)
        com.sun.tools.javac.util.Name$Table.make(Name.java:332)
        com.sun.tools.javac.util.Name$Table.instance(Name.java:349)
```

Each frame in the stack trace contains class name, method name, source file name, and the line number. The user can set the maximum number of frames collected by the HPROF agent. The default limit is 4. Stack traces reveal not only which methods performed heap allocation, but also which methods were ultimately responsible for making calls that resulted in memory allocation.

Heap Dump (heap=dump)

A complete dump of the current live objects in the heap can be obtained with:

```
Command used: javac -J-agentlib:hprof=heap=dump Hello.java
```

This is a very large output file, but can be viewed and searched in any editor.

CPU Usage Sampling Profiles (cpu=samples)

HPROF can collect CPU usage information by sampling threads. Following is part of the output collected from a run of the `javac` compiler.

```
Command used: javac -J-agentlib:hprof=cpu=samples Hello.java

CPU SAMPLES BEGIN (total = 462) Fri Feb  6 13:33:07 2004
rank   self  accum   count trace method
   1 49.57% 49.57%     229 300187 java.util.zip.ZipFile.getNextEntry
   2  6.93% 56.49%      32 300190 java.util.zip.ZipEntry.initFields
   3  4.76% 61.26%      22 300122 java.lang.ClassLoader.defineClass2
   4  2.81% 64.07%      13 300188 java.util.zip.ZipFile.freeEntry
   5  1.95% 66.02%       9 300129 java.util.Vector.addElement
   6  1.73% 67.75%       8 300124 java.util.zip.ZipFile.getEntry
   7  1.52% 69.26%       7 300125 java.lang.ClassLoader.findBootstrapClass
   8  0.87% 70.13%       4 300172 com.sun.tools.javac.main.JavaCompiler.<init>
   9  0.65% 70.78%       3 300030 java.util.zip.ZipFile.open
  10  0.65% 71.43%       3 300175 com.sun.tools.javac.main.JavaCompiler.<init>

...
CPU SAMPLES END
```

The HPROF agent periodically samples the stack of all running threads to record the most frequently active stack traces. The `count` field above indicates how many times a particular stack trace was found to be active. These stack traces correspond to the CPU usage hot spots in the application.

CPU Usage Times Profile (cpu=times)

HPROF can collect CPU usage information by injecting code into every method entry and exit, keeping track of exact method call counts and the time spent in each method. This uses Byte Code Injection (BCI) and runs considerably slower than cpu=samples. Following is part of the output collected from a run of the `javac` compiler.

```
Command used: javac -J-agentlib:hprof=cpu=times Hello.java

CPU TIME (ms) BEGIN (total = 2082665289) Fri Feb  6 13:43:42 2004
rank   self  accum   count trace method
   1  3.70%  3.70%       1 311243 com.sun.tools.javac.Main.compile
   2  3.64%  7.34%       1 311242 com.sun.tools.javac.main.Main.compile
   3  3.64% 10.97%       1 311241 com.sun.tools.javac.main.Main.compile
   4  3.11% 14.08%       1 311173 com.sun.tools.javac.main.JavaCompiler.compile
   5  2.54% 16.62%       8 306183 com.sun.tools.javac.jvm.ClassReader.listAll
   6  2.53% 19.15%      36 306182 com.sun.tools.javac.jvm.ClassReader.list
   7  2.03% 21.18%       1 307195 com.sun.tools.javac.comp.Enter.main
   8  2.03% 23.21%       1 307194 com.sun.tools.javac.comp.Enter.complete
   9  1.68% 24.90%       1 306392 com.sun.tools.javac.comp.Enter.classEnter
  10  1.68% 26.58%       1 306388 com.sun.tools.javac.comp.Enter.classEnter
```

```
...
CPU TIME (ms) END
```

Here the count represents the true count of the times this method was entered, and the percentages represent a measure of thread CPU time spent in those methods.

Binary Dump Format (format=b)

The basic fields in the binary output are u1 (1 byte), u2 (2 byte), u4 (4 byte), and u8 (8 byte). An ID in this implementation is a u4, however the size of an ID is really determined by the "size of identifiers" field in the header.

WARNING: This format is still considered highly experimental, however, all attempts were made to match the format of past HPROF implementations.

The binary output begins with the information:

[u1]*	An initial NULL terminated series of bytes representing the format name and version, in this implementation and historically, the string "JAVA PROFILE 1.0.1" (18 u1 bytes) followed by a NULL byte. If the TAG "HEAP DUMP SEGMENT" is used this string will be "JAVA PROFILE 1.0.2".
u4	size of identifiers. Identifiers are used to represent UTF8 strings, objects, stack traces, etc. They can have the same size as host pointers or sizeof(void*), but are not required to be.
u4	high word of number of milliseconds since 0:00 GMT, 1/1/70
u4	low word of number of milliseconds since 0:00 GMT, 1/1/70

Followed by a sequence of records that look like:

u1	TAG: denoting the type of the record
u4	TIME: number of microseconds since the time stamp in the header
u4	LENGTH: number of bytes that follow this u4 field and belong to this record
[u1]*	BODY: as many bytes as specified in the above u4 field

The following TAGs are supported:

STRING IN UTF8	0x01	ID: ID for this string [u1]*: UTF8 characters for string (NOT NULL terminated)
LOAD CLASS	0x02	u4: class serial number (always > 0)

		ID		class object ID
		u4		stack trace serial number
		ID		class name string ID
UNLOAD CLASS	0x03	u4		class serial number
STACK FRAME	0x04	ID		stack frame ID
		ID		method name string ID
		ID		method signature string ID
		ID		source file name string ID
		u4		class serial number
		u4	> 0	line number
			0	no line information available
			–1	unknown location
			–2	compiled method (*Not implemented*)
			–3	native method (*Not implemented*)
STACK TRACE	0x05	u4		stack trace serial number
		u4		thread serial number
		u4		number of frames
		[ID]*		series of stack frame ID's
ALLOC SITES	0x06	u2		Bit mask flags:
			0x1	incremental vs. complete
			0x2	sorted by allocation vs. line
			0x4	whether to force GC (*Not Implemented*)
		u4		cutoff ratio (floating point)
		u4		total live bytes
		u4		total live instances
		u8		total bytes allocated
		u8		total instances allocated
		u4		number of sites that follow:

Record	Tag	Sub-record	Sub-tag	Type	Description
				u1	array indicator: 0 means not an array, non-zero means an array of this type (See [Basic Type](#))
				u4	class serial number
				u4	stack trace serial number
				u4	number of live bytes
				u4	number of live instances
				u4	number of bytes allocated
				u4	number of instances allocated
HEAP SUMMARY	0x07			u4	total live bytes
				u4	total live instances
				u8	total bytes allocated
				u8	total instances allocated
START THREAD	0x0A			u4	thread serial number
				ID	thread object ID
				u4	stack trace serial number
				ID	thread name string ID
				ID	thread group name ID
				ID	thread parent group name ID
END THREAD	0x0B			u4	thread serial number
HEAP DUMP *or* HEAP DUMP SEGMENT	0x0C *or* 0x1C	Contains any number of sub-tags, each begins a u1 field (no order implied here):			
		ROOT UNKNOWN	0xFF	ID	object ID
		ROOT JNI GLOBAL	0x01	ID	object ID
				ID	JNI global ref ID
		ROOT JNI LOCAL	0x02	ID	object ID
				u4	thread serial number
				u4	frame number in stack trace (-1 for empty)

ROOT JAVA FRAME	0x03	ID: object ID u4: thread serial number u4: frame number in stack trace (–1 for empty)
ROOT NATIVE STACK	0x04	ID: object ID u4: thread serial number
ROOT STICKY CLASS	0x05	ID: object ID
ROOT THREAD BLOCK	0x06	ID: object ID u4: thread serial number
ROOT MONITOR USED	0x07	ID: object ID
ROOT THREAD OBJECT	0x08	ID: thread object ID u4: thread serial number u4: stack trace serial number

CLASS DUMP	0x20	ID: class object ID
		u4: stack trace serial number
		ID: super class object ID
		ID: class loader object ID
		ID: signers object ID
		ID: protection domain object ID
		ID: reserved
		ID: reserved
		u4: instance size (in bytes)
		u2: size of constant pool and number of records that follow:
		  u2: constant pool index
		  u1: type of entry: (See Basic Type)
		  value: value of entry (u1, u2, u4, or u8

			based on type of entry)
		u2	Number of static fields:
			ID: static field name string ID
			u1: type of field: (See [Basic Type])
			value: value of entry (u1, u2, u4, or u8 based on type of field)
		u2	Number of instance fields (not including super class's)
			ID: field name string ID
			u1: type of field: (See [Basic Type])

INSTANCE DUMP	0x21	ID	object ID
		u4	stack trace serial number
		ID	class object ID
		u4	number of bytes that follow
		[value]*	instance field values (this class, followed by super class, etc)
OBJECT ARRAY DUMP	0x22	ID	array object ID
		u4	stack trace serial number
		u4	number of elements
		ID	array class object ID
		[ID]*	elements
PRIMITIVE ARRAY DUMP	0x23	ID	array object ID
		u4	stack trace serial number
		u4	number of elements
		u1	element type (See [Basic Type])
		[u1]*	elements (packed array)

<table>
<tr><td>HEAP DUMP END</td><td>0x2C</td><td>Terminates a series of HEAP DUMP SEGMENTS. Concatenation of HEAP DUMP SEGMENTS equals a HEAP DUMP.</td></tr>
<tr><td>CPU SAMPLES</td><td>0x0D</td><td><table><tr><td>u4</td><td>total number of samples</td></tr><tr><td>u4</td><td>number of traces that follow:<table><tr><td>u4</td><td>number of samples</td></tr><tr><td>u4</td><td>stack trace serial number</td></tr></table></td></tr></table></td></tr>
<tr><td>CONTROL SETTINGS</td><td>0x0E</td><td><table><tr><td>u4</td><td>Bit mask flags:<table><tr><td>0x1</td><td>alloc traces on/off</td></tr><tr><td>0x2</td><td>cpu sampling on/off</td></tr></table></td></tr><tr><td>u2</td><td>stack trace depth</td></tr></table></td></tr>
</table>

Basic Type

2	object
4	boolean
5	char
6	float
7	double
8	byte
9	short
10	int
11	long

Handling of Arrays

There will be a "LOAD CLASS" tag for type type of each array in the dump. In the LOAD CLASS record, the class name string ID will refer to a string with a human-readable name of the array type that is formatted as the type name would be in Java source code. Thus, the LOAD CLASS record for the type char[] will be "char[]", for short[][][] will be "short[][][]" and for MyType[] will be "MyType[]".

Socket Connection and Communication

WARNING: This command format is still considered highly experimental, however, all attempts were made to match the format of past HPROF implementations.

Commands can be sent to HPROF via the socket connection, the accepted COMMAND TAGS are:

FORCE GC (*Not implemented*)	0x01
DUMP HEAP	0x02
ALLOC SITES	0x03
HEAP SUMMARY	0x04
EXIT THE VM	0x05
DUMP TRACES	0x06
CPU SAMPLES	0x07
CONTROL	0x08
EOF (used to terminate socket connection)	0xFF

The commands take the form:

<table>
<tr><td>u1</td><td>COMMAND TAG</td></tr>
<tr><td>u4</td><td>serial number</td></tr>
<tr><td>u4</td><td>number of bytes that follow</td></tr>
<tr><td>[u1]*</td><td>
<table>
<tr><td>ALLOC SITES</td><td>
<table>
<tr><td>u2</td><td>Flags:</td></tr>
<tr><td>u4</td><td>cutoff ratio (floating point between 0.0 and 1.0)</td></tr>
</table>
</td></tr>
<tr><td>CPU SAMPLES</td><td>
<table>
<tr><td>u2</td><td>ignored</td></tr>
<tr><td>u4</td><td>cutoff ratio (floating point between 0.0 and 1.0)</td></tr>
</table>
</td></tr>
<tr><td>CONTROL</td><td>
<table>
<tr><td>u2</td><td>Sub option:
<table>
<tr><td>0x1</td><td>Turn alloc traces on</td></tr>
<tr><td>0x2</td><td>Turn alloc traces off</td></tr>
<tr><td>0x3</td><td>Turn CPU sampling on:
<table><tr><td>ID</td><td>thread object ID (0 for all threads)</td></tr></table>
</td></tr>
<tr><td>0x4</td><td>Turn CPU sampling off:
<table><tr><td>ID</td><td>thread object ID (0 for all threads)</td></tr></table>
</td></tr>
<tr><td></td><td></td></tr>
</table>
</td></tr>
</table>
</td></tr>
</table>
</td></tr>
</table>

0x5	Clear CPU sampling	
0x6	Set max stack depth:	
	u2	New max stack depth

Source Code

The source to HPROF is available in the JDK download in the demo/jvmti/hprof directory.

***As used on this web site, the terms "Java Virtual Machine" or "JVM" mean a virtual machine for the Java platform.**

Last modified: 2005