

William Louth's Weblog [Wordpress Archive]

The Art of API Design and Performance Engineering

Better Java Thread Stack Trace Dumps

by williamlouth on February 8, 2009

When performing JVM runtime diagnostics the standard (*and need I say pretty crude*) approach adopted by most working for companies with very little in the way of mature IT management processes and tooling is the thread (*or exception*) stack trace dump.

There are numerous problems with this approach including the lack of object & argument state information (*already solved quite comprehensively*) as well as timing & resource usage which I hope to address here.

Lets start with the sample code used to generate the stack trace print outs presented in this entry.

A simple deep recursive call chain execution ending with a wait followed by a stack trace print.

```
public class Dumping {

    private static final int RUN_COUNT = 100000;
    private static final long WAITING = 1;
    private static final int ENTRYPOINT = 200;
    private static PrintWriter WRITER;
    {...}

    public static void main(String[] args)
        throws Exception {...}

    public static void test() {
        for (int i = RUN_COUNT; i > 0; i--) {
            try {
                call(ENTRYPOINT);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }

    private static void $0(int next) throws Exception {
        switch (next) {
            case 0: waiting(); break;
            default: throw new IllegalArgumentException();
        }
    }

    private static void waiting() throws Exception {
        Thread t = Thread.currentThread();
        synchronized (t) { t.wait(WAITING); }
        new Exception().printStackTrace(WRITER);
    }

    private static void $200(int next) throws Exception {
        call(--next);
    }

    public static void call(int next) throws Exception {
        switch (next) {
            case 200: $200(next); break;
            case 199: $199(next); break;
            case 198: $198(next); break;
        }
    }
}
```

Here is a small extract of a single stack trace printed by the above code.

```

1 java.lang.Exception
2   at stacks.Dumping.waiting(Dumping.java:65)
3   at stacks.Dumping.$0(Dumping.java:57)
4   at stacks.Dumping.call(Dumping.java:682)
5   at stacks.Dumping.$1(Dumping.java:1480)
6   at stacks.Dumping.call(Dumping.java:678)
7   at stacks.Dumping.$2(Dumping.java:1476)
8   at stacks.Dumping.call(Dumping.java:675)
9   at stacks.Dumping.$3(Dumping.java:1472)
10  at stacks.Dumping.call(Dumping.java:672)
11  at stacks.Dumping.$4(Dumping.java:1468)
12  at stacks.Dumping.call(Dumping.java:669)
13  at stacks.Dumping.$5(Dumping.java:1464)
14  at stacks.Dumping.call(Dumping.java:666)
15  at stacks.Dumping.$6(Dumping.java:1460)
16  at stacks.Dumping.call(Dumping.java:663)
17  at stacks.Dumping.$7(Dumping.java:1456)
18  at stacks.Dumping.call(Dumping.java:660)
19  at stacks.Dumping.$8(Dumping.java:1452)
20  at stacks.Dumping.call(Dumping.java:657)
21  at stacks.Dumping.$9(Dumping.java:1448)
22  at stacks.Dumping.call(Dumping.java:654)
23  at stacks.Dumping.$10(Dumping.java:1444)
24  at stacks.Dumping.call(Dumping.java:651)
25  at stacks.Dumping.$11(Dumping.java:1440)
26  at stacks.Dumping.call(Dumping.java:647)
27  at stacks.Dumping.$12(Dumping.java:1436)
28  at stacks.Dumping.call(Dumping.java:644)
29  at stacks.Dumping.$13(Dumping.java:1432)

```

The problem with the output is that we have no inclination to how long each *frame* has been on the threads execution call stack and the resources it has consumed since added.

We cannot even determine whether two identical stack traces, printed by the same thread, represent the exact same execution call (*frame*) stack. *Is this a new request at the same execution point or has the code stalled (waited) since the last print?*

In addition the stack trace print contains frames that represent *noise* in our runtime analysis if we were focused on tracking down performance bottlenecks. This can also be the case when analyzing applications deployed to standard middleware containers.

Now lets rewrite the `waiting` method to use our recently announced `stack probes` provider extension.

```

private static void waiting() throws Exception {
    Thread t = Thread.currentThread();
    synchronized (t) { t.wait(WAITING); }
    if (Probes.isStackEnabled())
        Probes.get().getStack().print(WRITER);
}

```

Here is an extract of the stack trace printed.

```

1 |<stack>
2   <probe name="stacks.Dumping.waiting" id="203002">
3     <meter name="clock.time" value="1237"/>
4   </probe>
5   <probe name="stacks.Dumping.$0" id="203001">
6     <meter name="clock.time" value="1238"/>
7   </probe>
8   <probe name="stacks.Dumping.$1" id="203000">
9     <meter name="clock.time" value="1238"/>
10  </probe>
11  <probe name="stacks.Dumping.$2" id="202999">
12    <meter name="clock.time" value="1239"/>
13  </probe>
14  <probe name="stacks.Dumping.$3" id="202998">
15    <meter name="clock.time" value="1239"/>
16  </probe>
17  <probe name="stacks.Dumping.$4" id="202997">
18    <meter name="clock.time" value="1240"/>
19  </probe>
20  <probe name="stacks.Dumping.$5" id="202996">
21    <meter name="clock.time" value="1240"/>
22  </probe>
23  <probe name="stacks.Dumping.$6" id="202995">
24    <meter name="clock.time" value="1241"/>
25  </probe>
26  <probe name="stacks.Dumping.$7" id="202994">
27    <meter name="clock.time" value="1241"/>
28  </probe>

```

A few observations. It is in XML format. Each fired and metered `Probe` (*frame*) has a unique id *within the context of each thread*. The wall clock time (in microseconds) is listed as a `Meter` for each frame on the stack.

Note: With the id we can compare across stack trace prints and performance call stack sampling with a much higher degree of accuracy (no second guessing whether the method indeed remained active during interval).

Did you notice that the intermediary `stacks.Dumping.call` method is absent in the above output? The reason is the `call(int)` method had been executed a sufficiently number of times (default=1000) prior to this particular stack print for the resource metering runtime to have already determined that it is not a hotspot (*in terms of inherent cost*) and does not need to be metered any further. *It was fired but not metered.*

Note: The runtime allows for different meters to be used in determining whether a particular named probe (method) is a hotspot or not.

Here is another stack trace printed later in the same run.

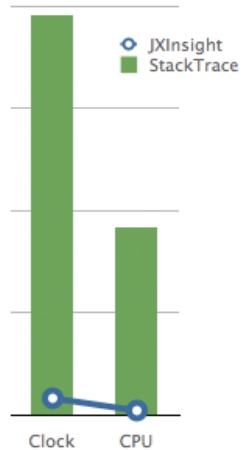
```
1 <stack>
2   <probe name="stacks.Dumping.waiting" id="203003">
3     <meter name="clock.time" value="1279"/>
4   </probe>
5   <probe name="stacks.Dumping.test" id="2">
6     <meter name="clock.time" value="1570311"/>
7   </probe>
8   <probe name="stacks.Dumping.main" id="1">
9     <meter name="clock.time" value="1570418"/>
10  </probe>
11 </stack>
12
```

Now all the `stacks.Dumping.$*` methods have completely disappeared. We are left with probes fired and metered for methods marked as hotspots or still under analysis by the probes runtime metering strategies.

You have probably already guessed that if you enable additional resource meters within the metering runtime these will also be included in the probe stack prints.

```
1 <stack>
2   <probe name="stacks.Dumping.waiting" id="203002">
3     <meter name="clock.time" value="1236"/>
4     <meter name="waiting.time" value="1094"/>
5   </probe>
6   <probe name="stacks.Dumping.$0" id="203001">
7     <meter name="clock.time" value="1237"/>
8     <meter name="waiting.time" value="1094"/>
9   </probe>
10  <probe name="stacks.Dumping.$1" id="203000">
11    <meter name="clock.time" value="1237"/>
12    <meter name="waiting.time" value="1094"/>
13  </probe>
14  <probe name="stacks.Dumping.$2" id="202999">
15    <meter name="clock.time" value="1238"/>
16    <meter name="waiting.time" value="1094"/>
17  </probe>
18  <probe name="stacks.Dumping.$3" id="202998">
19    <meter name="clock.time" value="1239"/>
20    <meter name="waiting.time" value="1094"/>
21  </probe>
22  <probe name="stacks.Dumping.$4" id="202997">
23    <meter name="clock.time" value="1239"/>
24    <meter name="waiting.time" value="1094"/>
25  </probe>
26  <probe name="stacks.Dumping.$5" id="202996">
27    <meter name="clock.time" value="1240"/>
28    <meter name="waiting.time" value="1094"/>
29  </probe>
```

Conventional call stack dumps quickly fill up a log file with a large amount of irrelevant and duplicated data but they are also extremely costly both in terms of wall clock time and CPU usage.



I am not advocating the complete elimination of thread stack trace dumps from logs *though I am sure many of us would wish it to be so especially when staring at the same exception printed 20*

times or more as it passes back up through the request processing pipeline.

Having the complete call frame stack can be helpful in a few cases but for the most part it is woefully inefficient (*both at runtime and off-line analysis*), wasteful, and severely lacking in important runtime diagnostic information – object state and resource usage.

There are solutions available today that make problem determination pretty efficient even for the most complex of systems. You just need to step outside of *the runtime* and to stop staring into the Sun (*it can blind you*).

Note: With the solution above probe stack prints can also cross multiple language boundaries – Java-to-Ruby.

FROM: JXINSIGHT, PROFILING