JVM Memory Management and ColdFusion Log Analysis

Posted At : September 9, 2010 5:24 PM | Posted By : Steven Erat
Related Categories: Java, Quality Assurance, ColdFusion, Computer Technology

The following is a document I wrote for knowledge sharing with some peers, but I feel that it might have some value to other ColdFusion Devs, Testers, and Admins out there. The purpose was to illustrate how I went about analyzing CF's performance during a prior troubleshooting session. I'm re-purposing the content here after scrubbing some private information. Hopefully it still makes sense, although slightly out of context.

These are some technical notes on what to look for when analyzing ColdFusion server performance history. It includes concepts and techniques to assess what performance related problems might exist, emphasizing memory usage issues first. This is a somewhat simplified explanation about how the JVM manages memory and its relation to CF applications, and about how I went about analyzing them. There are many similar resources on the web, but I found many of them are quite technical, so this article is written with more of a layman's approach to make it more digistible to those not as familiar with troubleshooting ColdFusion or Java apps.

The ColdFusion Application Server runs inside (is "contained" in) a higher level JRun J2EE server. The J2EE server (and therefore the CF server) run on top a JVM (Java Virtual Machine). To analyze the ColdFusion and JVM performance, I take a forensic approach. I start by collecting the ColdFusion and JRun server logs. I also colelct the JVM Garbage Collection (GC) log that has been manually enabled to log information regarding how the JVM is cleaning up the memory that is has used. The JVM is configured with an algorithm that tells it what approach to take when cleaning up and freeing memory. The application's Java objects (like queries, session variables, local variables, cfc instances, etc) are held in the JVM's memory. Objects are said to hold "references" in memory, meaning that something in the application is potentially using that object. When the application no longer has a need for an object, its memory is dereferenced. That dereferenced memory can be released by the JVM and then reused by other objects that require it.

Garbage Collection is the mechanism that searches through all objects in memory and determines if they are referenced or not, and for releasing the memory of dereferenced objects. The JVM has 2 major divisions for where it puts objects in memory, where the divisions are referred to as Generations. All objects first go into the "New Generation" (sometimes called the Young Generation). Then beginning a few seconds after the application starts, the JVM starts performing "Minor GCs" that only look at the objects in the New Generation. If objects in the New Gen are no longer needed (no longer referenced), GC cleans them up and marks the memory as free. For

New Gen objects that are still used, it moves them into another division called the "Tenured Generation" (sometimes called the Old Generation). That's where long lived objects are stored.

Objects that go into the New Gen that never make it into the Tenured Gen are usually variable scopes that "live" only for the length of a ColdFusion request. Examples are the Variables scope, the Request scope, URL and Form scopes, and private Local scope inside CFC methods. When the page request ends, those objects aren't referred to any more so the GC will (try to) free any memory they used to be reused by other objects. However, ColdFusion has longer lived objects as well, and examples are variables in the Session scope or Application scope, but also includes less obvious objects such as Cached Queries (and perhaps the virtualized RAMFS). Those longer lived objects will get moved from the New Gen to the Tenured Gen when GC runs because the application continues to use (hold references to) those objects after the page request ends.

Note this is a simplified description. The JVM Generations are actually broken down into small divisions (Eden, Survivor spaces) within the larger ones, but its not necessary to discuss it at that level here. There is also the "Permanent Generation" which stores very long lived objects like the classes that represent CFM/CFC templates. The PermGen has it's own heap, and is separate from the main JVM heap. See web links below for online references with more details.

There are several important terms to understand regarding memory and the JVM. The JVM can "allocate" memory, meaning it asks for that amount of memory from the operating system (OS). But once it allocates memory it doesn't necessarily "use" all that memory. Usually the JVM tries to allocate a bit more memory than it actually needs to use. So the allocated memory is approximately the same as the overall memory foot print you would see when looking at the jrun.exe process in Task Manager on Windows. The memory allocated by the JVM is referred to as the "heap". The JVM can be configured to have a maximum and minimum size to the heap. Since asking the OS for more memory is an expensive operation, setting the minimum heap size equal to the maximum heap size is thought to produce better performance because as soon as the application server starts, the JVM will allocate (reserve) as much memory as it will ever be allowed to, eliminating having to ask the OS for more memory or to give free memory back to the OS.

The JVM start up parameters are configured via a file at JRun4/bin/jvm.config. Here is a snippet from a jvm.config on a PROD_SERVER server and an explanation of it:

```
java.args=-Xmx12800M -server -Xms12800m -XX:+PrintGCDetails
-XX:+PrintGCTimeStamps -XX:+PrintHeapAtGC -verbose:gc
-Xloggc:PROD_GC.log -XX:PermSize=512m -XX:MaxPermSize=1024m
-XX:NewRatio=3 -XX:+UseParallelGC
```

These are the critical parameters passed to the JVM when it starts, as well as some parameters that help to debug it's memory usage. A few other parameters were omitted from this snippet because they weren't related to the discussion.

- **-Xmx12800M** Â« This is the Max Heap size setting. Here it is set to 12800MB or 12.5GB
- **-Xms12800m** Â«This is the Minimum Heap size setting. It is set to be equal to the Max Heap for better performance
- **-XX:+PrintGCDetails** Â«This is used for debugging. It means write the details of Garbage Collection to a log file
- **-XX:+PrintGCTimeStamps** Â« This is used for debugging. It means write timestamps of when GC activity occurred into the GC log file.
- **-XX:+PrintHeapAtGC** Â« This is used for debugging. This means to include heap information when logging GC activity.
- **-verbose:gc** Â« This is used for debugging. It tells the JVM to log as much information as possible about GC activity.
- **-Xloggc:PROD_GC** Â« This is used for debugging. Found in JRun4/bin/ directory, the log name here would be **PROD_GC.log**
- **-XX:PermSize=512m** Â«This is an optional parameter for the starting size of the "Permanent Generation" heap. PermGen not discussed here.
- **-XX:MaxPermSize=1024m** Â«This max size of the Perm Gen.
- **-XX:NewRatio=3** Â«This means to make the New Generation size equal to Â° the total heap size (literally a 1:3 ratio of New to Tenured)
- **-XX:+UseParallelGC** Â«This is the GC algorithm to use. There are several to choose from based on need. Not discussed here.

The exact time Garbage Collection is run can't be controlled. An application could ask the JVM to run GC, but the JVM will not necessarily run GC when requested but will schedule it at the next opportunity. Normally, applications don't ask the JVM to perform GC. Instead, the JVM is configured to use one of several possible algorithms about how best to perform Garbage Collection. The algorithm is chosen is based on needs of the application and can affect performance in different ways. The goal is to choose an algorithm that works well for your hardware resources and application needs. Ideally for web applications it is desirable to choose an algorithm that produces

minimal interruption to the user experience. The one here, -XX:+UseParallelGC, is the default configuration when ColdFusion is installed. This is a deep topic I won't go into further here.

Since the JVM minimum heap is set to the max heap, to understand the JVM memory behavior you have to also look at Memory Used, Memory Free, and Memory Allocated. In this case the Min Heap size reserved is 12.5GB for the JVM, so the JVM will actually allocate the whole 12.5GB right away. But in practice the CF servers only use a fraction of the total amount reserved.

If you theoretically gave a JVM as much memory as it could ever need, the normal JVM behavior would be for the Memory Used to grow to a high water mark, and only on occasion exceed it. That high water mark is sometimes referred to as an application's memory footprint. In fact, because GC runs in a periodic fashion and releases some fraction of memory used, a graph of JVM Memory Used would look a lot like a flat line with saw teeth. If, however, an application seems to gradually increase the amount of memory it uses, producing a graphed line of an upwardly slanted saw tooth, then it would be said that the application has a memory leak where something in the app constantly requires more and more memory. A memory leak can happen in an application when it creates objects that live a long time, and keeps creating them rather than destroying previously created objects and reusing that memory. In ColdFusion terms, this could happen due a problem in the programming logic of an app where objects like queries or even small variables are stored in the long lived Application scope. In practice, the Application scope lives indefinitely, and objects are left there unless the program specifically removes them. So it an app unwittingly kept putting more and more objects in the Application scope, the JVM memory used would be seen to grow and grow indefinitely.

In more practical terms, JVMs are configured with a Maximum Heap size to limit how much memory the JVM could ever have. Ideally the application memory footprint would be determined through testing and the Max Heap size would set somewhat larger than that sawtoothed footprint, giving the app enough for what it needs, but not much more. Sometimes an app is coded in a way that causes it to unexpectedly require significantly more memory than the typical footprint for that app. This could be a memory leak behavior with potential for an infinite upward trend, or it could be that the unexpected requirement is actually 2x or 3x as much as normal without increasing further (for you marine biologists out there, that would called a Spring Tide, the highest of the high tides).

When an application causes the JVM to attempt to use more memory than it is permitted to allocate to the heap, a Java exception occurs, java.lang.OutOfMemoryError (OOM). When an OOM error is logged, it may appear with

several different sub-contexts that have different meaning. Usually it would be logged as "*java.lang.OutOfMemoryError: Java heap space*". This context typically means exactly what it looks like. The JVM tried to use more memory than it had. However, the OOM errors can be logged in other forms such as "*java.lang.OutOfMemoryError: GC overhead limit exceeded*". In this case, there is an exact definition stating that the JVM was spending 98% of its time trying to perform GC but could only free up less than 2% of the memory used.

When an OOM exception occurs, whatever request or activity the JVM was executing would be terminated abruptly without completing. Moreover, when several OOMs occur together chronologically it is a symptom that the application is using the maximum amount of memory allowed and that the JVM throughput performance (how many requests it can process per unit time) is decreased. The JVM throughput decreases because it is spending more time than normal trying to free up memory from the heap by looking for unused, dereferenced objects. When a JVM GC cycle runs, it can be a Minor GC where just the New Gen is cleaned up, but less frequently a "*Full GC*" is run which will try to free up memory in both the New and Tenured Generations. When a Full GC runs, all other activity in the JVM halts and no requests are processed until the Full GC is complete. Full GCs are sometimes referred to as Stop The World GCs.

Normally, Full GCs are short and infrequent, happening a couple times per hour and usually lasting 50 to 100 milliseconds at most. But when the JVM is having OOM issues, Full GCs can happen several times per minute, lasting 5 to 20 seconds or more each time. In that worst case scenario, the JVM throughput would fall dramatically and the application would appear to be unresponsive or hung since the JVM is effectively stalled while it keeps trying to free up memory. Usually at that point someone in IT is notified because some monitor alarms have gone off, and the application server will be restarted. Sometimes the JVM can recover if given some more time, but to the end user of an application it will appear dead until then.

When I begin reviewing the logs, I collect them from JRun4/logs, from JRun4/servers/PROD_SERVER/cfusion.ear/cfusion.war/WEB-INF/cfusion/logs, and the PROD_GC.log from JRun4/bin and transfer them to my local computer. It's important to note that the GC log, named PROD_GC.log here, **will be overwritten completely every time the CF server is restarted**. So while the JRun and CF logs will contain a full history, the GC log is only as good as the last restart.

Here are some steps I usually take to make sense of the log information:

- **All logs**: Examine them from a place where they can be quickly and efficiently searched. I copy them to my Mac laptop where I could use Unix command line tools to extract useful information into a condensed format. For example, from the large and verbose GC log, I was able to isolate the timestamp and duration of the Full GCs by running this command:

```
ogrep "Full GC" PROD_GC.log | cut -d"[" -f1,6 | grep secs
| cut -d" " -f1,5,6 | cut -d"]" -f1
```

This produced this kind of clean output from a very verbose and noisy log file:

3.285: real=0.06 secs
3603.606: real=1.17 secs
7205.023: real=1.16 secs
10806.527: real=1.77 secs

GC log: Determine the log start time from the file creation date, then convert the timestamps from seconds since server start to actual date/times. This way the GC log can be compared chronologically to events in the CF or JRun logs. Key features to look for are Full GCs that occur close together chronologically, and Full GCs that last more than a second or two.

I'll add as a foot note to this that I tried the **IBM PMAT** tool to analyze the GC log, recommended by **Brandon Harper**, but found several problem. First, on one server during the memory related problem time, the timestamps were out of order, rendering the tool useless. I also found that the logs do contain "Full GC" statements yet the PMAT tool kept reporting 0 Full GC occurrences.

- **JRun logs**: Manually scan a couple recent logs to see the range of entries that occur to get a feel for what type of issues are frequent. Then do some searches to look for things known to be bad like Memory errors. For example the following will extract all memory errors from all the logs and write them to a new file to easily review.

```
ogrep Memory PROD_SERVER-o*.log > memory_errors.txt
```

This might produce output like this (Note that Memory errors often don't get timestamps written next to them so you have to manually review the log having them):

```
PROD_SERVER-out41.log:java.lang.OutOfMemoryError: GC
overhead limit exceeded
PROD_SERVER-out41.log:java.lang.OutOfMemoryError: Java
```

```
heap space
PROD_SERVER-out45.log:Exception in thread
java.lang.OutOfMemoryError: GC overhead limit exceeded
PROD_SERVER-out45.log:Exception in thread
java.lang.OutOfMemoryError: GC overhead limit exceeded
PROD_SERVER-out45.log:java.lang.OutOfMemoryError: Java
heap space
```

JRun logs: If the Metrics logging feature has been enabled, then the JRun logs will contain frequent entries showing the number of requests running, number of sessions, and used/free JVM memory. Metrics may appear in their own log or may be mixed in with the -out.log. One way to quickly examine the Metrics logging for, say, the number of sessions active on a given date would be:

```
ogrep "09/03" PROD_SERVER-out*.log | grep "Sess:" | cut d"
" f2,8,9,10,11 > Sessions_0903.txt
```

Results:

```
05:44:46 Sess: 2727 Total/Free=12903744/10739858
05:45:06 Sess: 2733 Total/Free=12903744/10639087
05:45:26 Sess: 2731 Total/Free=12196800/8246066
05:45:49 Sess: 2726 Total/Free=12014976/4159270
05:46:12 Sess: 2725 Total/Free=12014976/2085405
05:46:59 Sess: 2718 Total/Free=12014976/1330244
```

CF logs: Get a list of when the CF server restarted from the server.log. The following will output the date and time of every CF server restart:

```
ogrep started server.log | cut -d"," -f3,4
```

Results:

```
"09/01/10","00:00:46
"09/05/10","16:08:55"
"09/06/10","08:50:12"
```

- **CF logs**: Scan the recent application.log, exception.log, and server.log to see if anything of interest appears, then perform searches to isolate any specific types of

entries.

- **All logs**: Once you've identified log entries and events of interest, start cross referencing entries from all the different logs where relevant. *The objective is to build a story of what happened in the application by following the chronology of events in the logs as they unfolded.*

Cross reference memory errors to server restarts. Cross reference Full GCs to groupings of timeout errors in CF logs. Follow memory used over time and look for patterns of unusual or suddent memory growth. If there are unusual events that occur with the same periodicity, say every day at X o'clock, then look for scheduled tasks that occur at that time, or correlate to social patterns like sudden web load right around 9:00 AM when users of the web application arrive at work.

It should be noted that at times when the JVM is reporting frequent memory errors that other unusual events may be seen in the logs that don't occur at other times. I think of these as secondary effects caused by insufficient memory, and that if memory were no problem the secondary issues wouldn't happen. An example is that on one server I found that the timestamps of log entries (JRun logs, CF logs, and GC log) were written out of order. Unusual timeout errors might occur, or other unusual errors might occur during the OOM issues. So be wary of spending time chasing down those secondary issues, and try to seek out the upstream, primary cause instead.

To provide an example of my initial findings on the servers, the first hot issue I was able to identify was OOM errors and very high memory usage that occurred every day at exactly the same time to the minute. Within 2 minutes memory would grow from 1.5GB used to 10GB used. The pattern was observed on the main, clustered production servers, although it might be on one prod server today and a different prod server tomorrow as the instances are load balanced. Because the problem occurred when regular web app users were off the system early in the morning I looked for scheduled tasks that might be involved. One such scheduled task was identified to run at that exact time each day. The code was reviewed, and from a QA point of view it was assessed that there was potential for high memory usage from a single request based on monolithic logic that analyzed tens of thousands of documents with mult-level nested looping constructs. The key problem is that Java objects created during page execution can't be Garbage Collected until the page request is completed. All objects created by the request exist for the life of the request. So as long as its running it could be requiring more and more memory as it processes those tens of thousands of documents.

By analogy, the ColdFusion web application server is normally intended for high traffic

with relatively fast running requests, but it was being used as though it were more of a Mainframe computer running a single, large C program. While using CF that way is possible, doing so will produce very different application behaviors and have different performance requirements than it would normally while serving lots and lots of short web page requests.

At this point since the evidence against this scheduled task was strong, the onus is on Development to confirm or refute that the job consumes memory as thought, and then to revise the job for better performance. One technique was suggested, to break down the job into a series of small jobs that run in succession. Say run the first few thousand documents to be analyzed, then the next few thousand, until done. That would allow each mini-job to complete, and allow objects created during the request to be garbage collected and the memory reused for the next job. If each mini-job runs shorter and processes fewer documents, then the total memory requirement would likely be much less. This would be a scalable tactic since you could chain together many such jobs in sequence during times when business grows and larger numbers of documents need to be analyzed.

For a more academic understanding of JVM memory management see these articles:

Tuning Garbage Collection Outline (An overview)

http://www.petefreitag.com/articles/gctuning/

Tuning Java Garbage Collection (Very thorough)

http://www.oracle.com/technetwork/java/gc-tuning-5-138395.html

Troubleshooting a Leaky Heap in your JVM (ColdFusion specific)

http://www.coldfusionmuse.com/index.cfm/2008/2/12/leaky.heap.jvm

JVM Options

http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html