# Troubleshooting a Leaky Heap in Your JVM

Posted At : February 12, 2008 1:19 PM | Posted By : Mark Kruger
Related Categories: Coldfusion 8, Coldfusion Troubleshooting

The phrase "memory leak" can cause shivers to run down the spine of the most seasoned developer. Having some process on your server that is gloaming onto memory and failing to release it is a guaranteed all nighter lurking somewhere in your future. Recently we were debugging a new, soon to be released application. We discovered what looked like a memory leak. The JVM memory used would climb steadily toward the maximum heap size. When the runtime garbage collection kicked in it would reduce the memory only by about a third of the increase. So, for example, memory use would climb from 300 megs to 600 megs and then GC would reduce usage back to 500 megs and so on. This situation would inevitably lock up the server with out of memory errors. What follows is a recap of our troubleshooting journey.

## Stop Gap Measure: System GC

While CF Guru Mike Klostermeyer examined the code that was most commonly in play, I was tasked with examining the Java Settings. The first thing I did was fiddle with manually firing garbage collection. One set of code I found looked like this:

```
<cfset runtime.gc()>

<cfdump var="#runtime.freememory()#"><BR>
    <!--- run GC --->
    <cfset runtime.gc()>
<cfdump var="#runtime.freememory()#"><BR>
```

This code ran the GC with exactly the same behavior as before - giving back *some* memory, but with no baseline. However, the "free memory" method turned out to be useful in another way. I created a CF page with the following code:

```
<Cfset runtime = CreateObject("java","java.lang.Runtime").getRuntime()>

<cfset fm = runtime.freememory()/>

<Cfset fm = int((fm/1024)/1024)/>

<cfset usedmem = 1270-fm/>
<cfoutput><br>
<br>
Free: #fm# megs<br>
Used: #usedmem# megs<br>
</cfoutput>
```

This was quick view of where memory usage was at on the server. Meanwhile back to my problem.

I decided to try a "system" garbage collection. As I understand it, the runtime GC is a suggestion, but the system GC is a "stop the world" command. I whipped up the following code:

```
<cfset obj = createObject("java","java.lang.System")/>

    <cfset obj.gc()/>

    <cfset obj.runFinalization()/>
```

This worked really well. It took about 1 to 1.5 seconds to run and Memory usage on the heap dropped down to a floor of about 25% each time I ran this code. I combined it with some of my "free memory" code and scheduled a task that ran the system GC whenever memory reached 1 gigabyte of usage. This solved our stability problems but it was not optimal. We knew we still had a leak somewhere. When discussing the situation with some folks on one of my email lists it was suggested that the speed of the system GC was a good clue. The system GC seemed to be able to *get back* quite a bit of memory very fast. It must be because there are many de-referenced objects on the "old generation" part of the heap.

Now in case you missed it in school or on Entertainment Tonight, Java divides the heap into "new" generation space and "old" (tenured) generation space. Without boring you with copious details, objects are always created on the "new" generation space on the theory that most objects live a very short time. For example, when you create a variable in the local scope it survives until the request ends and then it can be safely deleted from the heap. So the vast majority of variables and objects in any code base survive only a short time and live their entire life in the hurky jurky world of the "new" generation. Objects that are intended to live beyond a single request (like application and session variables) get a buyout and are moved to the "old" generation space.

A lot of the oddly named Java switches that we play with in the Jvm.config file have to do with allocating memory or collecting memory on the "old" or the "new" heap space. For example, -XX:+UseParNewGC specifies to the JVM which GC to use for cleaning up the new space. Anyway, the theory in our case was that the runaway memory allocation was occurring in "tenured" (old) memory. Mike and I began to work with the scopes that we considered candidates for "old" memory - application, server and session scoped objects and variables. After a day Mike finally found this snippet of code.

```
<cfif isDefined("#tmpString#")>
     <Cfset snapData =
        session.data["#arguments.params.uniqueID#"]>
<cfelse>
<Cfif isDefined("session.data")>
<Cfset structClear(session.data)>
</CFIF>
<cfset snapData =
             application.ChartDataObj.getSnapshotData(symbol=arguments.params.symbol)>
<cfset session.data["#arguments.params.uniqueID#"] = snapData>
</cfif>
```

The purpose of this code is to retrieve a Real time stock quote in order to append the value to one of 12 or 13 studies and charts. Because we didn't want to get the quote 12 times in a row we are storing the quote in the session and then we accessing it from the subsequent (Nearly simultaneous) requests. The "application.ChartDataObj" is a collection of methods with no properties attached. So the code above either pulls the data directly from the session, or creates it directly and references it in the session. In either case the goal of this code block is to create the "snapData" variable (an array) for use later on in the function. The variable "snapData" is correctly vared at the top of the function.

When mike removed all of this code and replaced it with just:

```
<cfset snapData =
        application.ChartDataObj.getSnapshotData(symbol=arguments.params.symbol)>
```

Our memory problems disappeared. Yes, memory still climbed steadily, but the regular GC operations of the JVM took care of recovering the memory as expected. Now we had a problem. We needed some way of caching this data. We still needed to overcome the necessity to hit our quote server 12 times in a row for the same information. What could we do? The answer was to deep copy the return var from the function using duplicate. Yes, I know it sounds simple but after 2 days of troubleshooting this is what actually solved our problem:

```
<cfif isDefined("#tmpString#")>
     <Cfset snapData =
        session.data["#arguments.params.uniqueID#"]>
<cfelse>
<Cfif isDefined("session.data")>
<Cfset structClear(session.data)>
</CFIF>
<cfset snapData =

    duplicate(application.ChartDataObj.getSnapshotData(symbol=arguments.params.symbol))>
<cfset session.data["#arguments.params.uniqueID#"] = snapData>
</cfif>
```

This code caused the data returned by the application object to be copied *by value* into the session instead of *by reference*. When we implemented this code, our memory climb shallowed out noticeably and the regular runtime GC was able to bring it back down to the floor of 25% at regular intervals of an hour or so.

### Rule of Thumb

What can be gleaned from this exercise? Well at least one "rule of thumb" for us is to carefully consider how we handle objects that are cached in persistent scopes. To boil it down to a single rule it would be "Avoid referencing returned objects from one persistent scope to another and copy by value instead".

### Final JVM Arguments

In case you wanted a rundown of our final JVM arguments arrived at through trial and error - we found the following to work well in our environment (Your environment may be quite different):

```
java.args=-server -Xmx1280m -Xms1280m -Dsun.io.useCanonCaches=false -XX:PermSize=64m
-XX:+UseConcMarkSweepGC -XX:NewSize=48m -XX:SurvivorRatio=4 -XX:+UseParNewGC
-XX:MaxPermSize=192m
```

We are also indebted (as always) to the many fine gurus who help out cheerily on the email lists to which we subscribe. The following blog posts on Java and Coldfusion deserve honorable mention:

- Pete Freitag - GC Tuning
- Robi Sen - JVM Options
- Daryl Banttari - CMS Collector

We are also indebted to the new server monitor in Coldfusion 8 and to the old tried and true SeeFusion application that provides excellent introspection into the inner works of the JVM.