Finding Memory Usage in Java

Asked 15 years, 10 months ago Modified 8 years, 8 months ago

Viewed 10k times

6

Following is the scenario i need to solve. I have struck with two solutions.

I need to maintain a cache of data fetched from database to be shown on a Swing GUI. Whenever my JVM memory exceeds 70% of its allocated memory, i need to warn user regarding excessive usage. And once JVM memory usage exceeds 80%, then i have to halt all the database querying and clean up the existing cache fetched as part of the user operations and notifying the user. During cleanup process, i will manually handle deleting some data based up on some rules and instructs JVM for a GC. Whenever GC occurs, if memory cleans up and reaches 60% of the allocated memory, I need to restart all the Database handling and giving back control to the user.

For checking JVM memory statistics i found following two solutions. Could not able to decide which is best way and why.

1. Runtime.freeMemory() - Thread created to run every 10 seconds and check for the free memory and if memory exceeds the limits mentioned, necessary popups will intimate user and will call the methods to halt the operations and freeing up the memory.

2. MemoryPoolMXBean.getUsage() - Java 5 has introduced JMX to get the snapshot of the memory at runtime. In, JMX i cannot use Threshold notification since it will only notify when memory reaches/exceeds the given threshhold. Only way to use is Polling in MemoryMXBean and check the memory statistics over a period.

In case of using polling, it seems for me both the implementations are going to be same.

Please suggest the advantages of the methods and if there are any other alternatives/any corrections to the methods using.

java memory memory-management garbage-collection

Share

Improve this question

Follow

edited Apr 12, 2016 at 8:42

Peter Lawrey

533k ● 82 ● 767 ● 1.1k

asked Feb 17, 2009 at 5:08

Satti

91 ● 1 ● 4

8 Answers

Sorted by: Highest score (default)

Just a side note: `Runtime.freeMemory()` doesn't state the amount of memory that's left of allocating, it's just the

15

amount of memory that's free within the currently allocated memory (which is initially smaller than the maximum memory the VM is configured to use), but grows over time.

When starting a VM, the max memory (`Runtime.maxMemory()`) just defines the upper limit of memory that the VM may allocate (configurable using the -Xmx VM option). The total memory (`Runtime.totalMemory()`) is the initial size of the memory allocated for the VM process (configurable using the -Xms VM option), and will dynamically grow every time you allocate more than the currently free portion of it (`Runtime.freeMemory()`), until it reaches the max memory.

The metric you're interested in is the memory available for further allocation:

```
long usableFreeMemory= Runtime.getRuntime().maxMemory(
    -Runtime.getRuntime().totalMemory()
    +Runtime.getRuntime().freeMemory()
```

or:

```
double usedPercent=(double)(Runtime.getRuntime().total
    -Runtime.getRuntime().freeMemory())/Runtime.getRun
```

Share Improve this answer edited Nov 6, 2014 at 14:54

Follow

answered Feb 17, 2009 at 8:49

Peter Walser
15.7k ● 4 ● 55 ● 80

7

The usual way to handle this sort of thing is to use `WeakReference`s and `SoftReference`s. You need to use both - the weak reference means you are not holding multiple copies of things, and the soft references mean that the GC will hang onto things until it starts running out of memory.

If you need to do additional cleanup, then you can add references to queues, and override the queue notification methods to trigger the cleanup. It's all good fun, but you do need to understand what these classes do.

Share Improve this answer Follow

edited Feb 18, 2009 at 13:03

answered Feb 18, 2009 at 12:54

paulmurray
3,413 ● 1 ● 23 ● 17

6

It is entirely normal for a JVM to go up to 100% memory usage and them back to say 10% after a GC and do this every few second.

You shouldn't need to try managing the memory in this way. You cannot say how much memory is being retained until a full GC has been run.

I suggest you work out what you are really trying to achieve and look at the problem another way.

Share Improve this answer

Follow

answered Feb 17, 2009 at 7:22

Peter Lawrey

533k ● 82 ● 767 ● 1.1k

> IMO, it's not entirely normal. These parameters can be tuned (pool sizes, full GC intervals, etc.) – Yoni Roit Feb 17, 2009 at 7:36

4

The requirements you mention are a clear contradiction with how Garbage Collection works in a JVM.

because of the behaviour of the JVM it will be very hard to warn you users in a correct way. Altogether stopping als database manipulation , cleaning stuff up and starting again really is not the way to go.

Let the JVM do what it is supposed to do, handle all memory related for you. Modern generations of the JVM are very good at it and with some finetuning of the GC parameters you will get a a much cleaner memory handling then forcing things yourself

Articles like http://www.kodewerk.com/advice_on_jvm_heap_tuning_dont_touch_that_dial.htm mention the pros and cons and offer a nice explanation of what the VM does for you

Share Improve this answer

Follow

answered Feb 17, 2009 at 13:46

Peter

5,778 ● 22 ● 23

2

I've only used the first method for similar task and it was OK.

One thing you should note, for both methods, is to implement some kind of debouncing - i.e. once you recognize you've hit 70% of memory, wait for a minute (or any other time you find appropriate) - GC can run at that time and clean up lots of memory.

If you implement a Runtime.freeMemory() graph in your system you'll see how the memory is constantly going up and down, up and down.

Share Improve this answer

Follow

answered Feb 17, 2009 at 5:15

Yoni Roit

28.6k ● 7 ● 38 ● 32

1

VisualVM is a bit nicer than JConsole because it gives you a nice visual Garbage Collector view.

Share Improve this answer

Follow

answered Dec 8, 2009 at 22:12

djangofan

29.6k ● 61 ● 207 ● 301

0

Look into JConsole. It graphs the information you need so it is a matter of adapting this to your needs (given that you run on a Sun Java 6).

This also allows you to detach the surveiling process from what you want to look at.

Share Improve this answer

Follow

answered Feb 17, 2009 at 23:00

Thorbjørn Ravn Andersen

75.3k ● 34 ● 199 ● 352

0

Very late after the original post, I know, but I thought I'd post an example of how I've done it. Hopefully it'll be of some use to someone (I stress, it's a proof of principal example, nothing else... not particularly elegant either :))

Just stick these two functions in a class, and it should work.

EDIT: Oh, and `import java.util.ArrayList;` `import java.util.List;`

```
public static int MEM(){
    return (int)(Runtime.getRuntime().maxMemory()-
Runtime.getRuntime().totalMemory()
+Runtime.getRuntime().freeMemory())/1024/1024;
}

public static void main(String[] args) throws Interrup
{
    List list = new ArrayList();
```

```
    //get available memory before filling list
    int initMem = MEM();
    int lowMemWarning = (int) (initMem * 0.2);
    int highMem = (int) (initMem *0.8);


    int iteration =0;
    while(true)
    {
        //use up some memory
        list.add(Math.random());

        //report
        if(++iteration%10000==0)
        {
            System.out.printf("Available Memory: %dMb
MEM(),list.size());

            //if low on memory, clear list and await g
continuing
            if(MEM()<lowMemWarning)
            {
                System.out.printf("Warning! Low memory
Clearing list and cleaning up.\n",MEM());

                //clear list
                list = new ArrayList();  //obviously,
put your warning logic

                //ensure garbage collection occurs bef
to list, to avoid immediately entering this block agai
                while(MEM()<highMem)
                {
                    System.out.printf("Awaiting gc...(
remaining)\n",MEM());
                    //give it a nudge
                    Runtime.getRuntime().gc();
                    Thread.sleep(250);
                }

                System.out.printf("gc successful! Cont
remaining). List size: %d\n",MEM(),list.size());
                Thread.sleep(3000); //just to view out
            }
```

```
            }
        }
    }
```

EDIT: This approach still relies on sensible setting of memory in the jvm using -Xmx, however.

EDIT2: It seems that the gc request line really does help things along, at least on my jvm. ymmv.

Share Improve this answer Follow

edited Sep 15, 2011 at 15:33

answered Sep 15, 2011 at 15:14

miklatov
63 1 8