Java 内存泄漏分析及思路总结

封面图来自 圆明园-福海

概述

本文主要分析一次java内存泄漏的事故, 内容主要分两块

- java层面的分析
- 进程层面的分析

最后还会有一个堆外内存泄漏如何分析解决的思路小结

java层面的分析

使用 `-Xmx512M` 启动java进程, 但通过 linux top 命令发现, java 进程的 RES (resident memory usage 常驻内存, 进程当前使用的内存) 一直在增长 并且远大于设定的最大堆, 猜测可能有堆外内存溢出

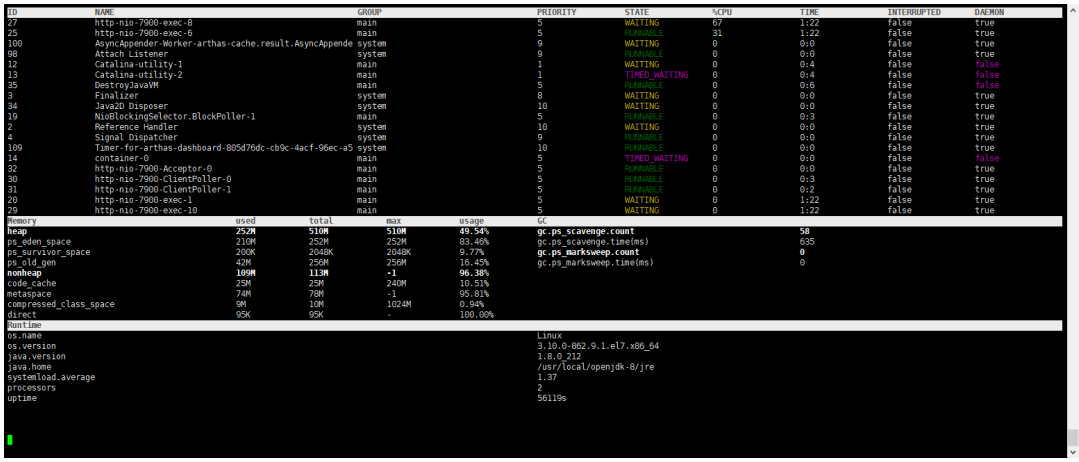
```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 2640 root      20   0 3068932 867324  16476 S   0.3 22.3   9:19.60 java -Xmx512m -Xms512m -XX:+UseParallelOldGC
```

再经过一段时间的压力测试后, 肯定不正常了, RES 为1.3g 远超限定的 512M

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 2640 root      20   0 3727400   1.3g  16488 S  81.7 33.9  19:19.44 java -Xmx512m -Xms512m -XX:+UseParallelOldGC
```

使用 arthas 验证下我们的想法

```
docker exec -it async-project /bin/bash -c "java -jar arthas-boot.jar"
# 或者 or
docker exec -it async-project /bin/bash -c "wget https://alibaba.github.io/arthas/arthas-boot.jar && java -jar arthas-boot.jar"
```

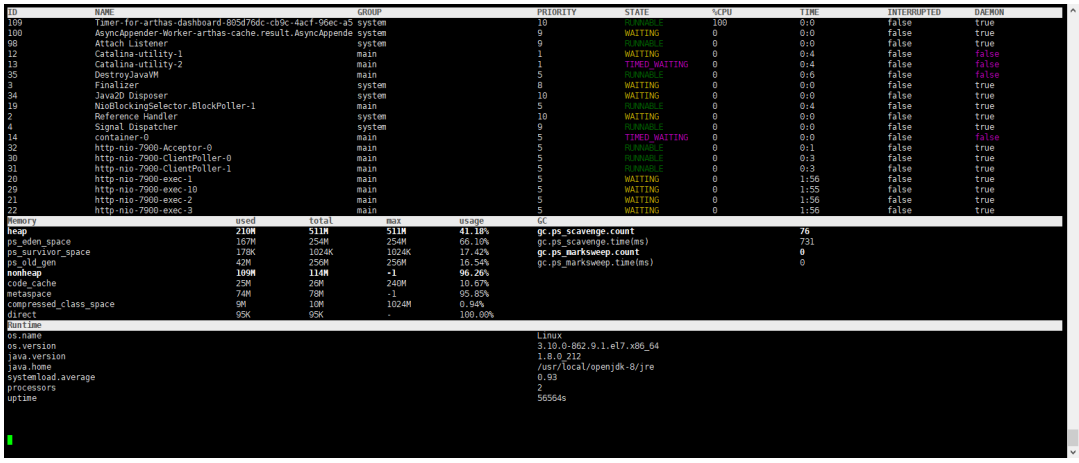


ID	NAME	GROUP	PRIORITY	STATE	%CPU	TIME	INTERRUPTED	DAEMON
27	http-nio-7900-exec-8	main	5	WAITING	67	1:22	false	true
25	http-nio-7900-exec-6	main	5	RUNNABLE	31	1:22	false	true
100	AsyncAppender-Worker-arthas-cache.result.AsyncAppende	system	9	WAITING	0	0:0	false	true
98	Attach Listener	system	9	RUNNABLE	0	0:0	false	true
12	Catalina-utility-1	main	1	WAITING	0	0:4	false	false
13	Catalina-utility-2	main	1	TIMED_WAITING	0	0:4	false	false
35	DestroyJavaVM	main	5	RUNNABLE	0	0:6	false	false
3	Finalizer	system	8	WAITING	0	0:0	false	true
34	Java2D Disposer	system	10	WAITING	0	0:0	false	true
19	NioBlockingSelector.BlockPoller-1	main	5	RUNNABLE	0	0:3	false	true
2	Reference Handler	system	10	WAITING	0	0:0	false	true
4	Signal Dispatcher	system	9	RUNNABLE	0	0:0	false	true
109	Timer-for-arthas-dashboard-805d76dc-cb9c-4acf-96ec-a5	system	10	RUNNABLE	0	0:0	false	true
14	container-0	main	5	TIMED_WAITING	0	0:0	false	false
32	http-nio-7900-Acceptor-0	main	5	RUNNABLE	0	0:0	false	true
30	http-nio-7900-ClientPoller-0	main	5	RUNNABLE	0	0:3	false	true
31	http-nio-7900-ClientPoller-1	main	5	RUNNABLE	0	0:2	false	true
20	http-nio-7900-exec-1	main	5	WAITING	0	1:22	false	true
29	http-nio-7900-exec-10	main	5	WAITING	0	1:22	false	true

Memory	used	total	max	usage	GC	
heap	252M	510M	510M	49.54%	gc.ps_scavenge.count	58
ps_eden_space	210M	252M	252M	83.46%	gc.ps_scavenge.time(ms)	635
ps_survivor_space	200K	2048K	2048K	9.77%	gc.ps_marksweep.count	0
ps_old_gen	42M	256M	256M	16.45%	gc.ps_marksweep.time(ms)	0
nonheap	109M	113M	-1	96.38%		
code_cache	25M	25M	240M	10.51%		
metaspace	74M	78M	-1	95.81%		
compressed_class_space	9M	10M	1024M	0.94%		
direct	95K	95K	-	100.00%		

Runtime	
os.name	Linux
os.version	3.10.0-862.9.1.el7.x86_64
java.version	1.8.0_212
java.home	/usr/local/openjdk-8/jre
systemload.average	1.37
processors	2
uptime	56119s

在运行一段时间后没有明显变化



ID	NAME	GROUP	PRIORITY	STATE	%CPU	TIME	INTERRUPTED	DAEMON
109	Timer-for-arthas-dashboard-805d76dc-cb9c-4acf-96ec-a5	system	10	RUNNABLE	100	0:0	false	true
100	AsyncAppender-Worker-arthas-cache.result.AsyncAppende	system	9	WAITING	0	0:0	false	true
98	Attach Listener	system	9	RUNNABLE	0	0:0	false	true
12	Catalina-utility-1	main	1	WAITING	0	0:4	false	false
13	Catalina-utility-2	main	1	TIMED_WAITING	0	0:4	false	false
35	DestroyJavaVM	main	5	RUNNABLE	0	0:6	false	false
3	Finalizer	system	8	WAITING	0	0:0	false	true
34	Java2D Disposer	system	10	WAITING	0	0:0	false	true
19	NioBlockingSelector.BlockPoller-1	main	5	RUNNABLE	0	0:4	false	true
2	Reference Handler	system	10	WAITING	0	0:0	false	true
4	Signal Dispatcher	system	9	RUNNABLE	0	0:0	false	true
14	container-0	main	5	TIMED_WAITING	0	0:0	false	false
32	http-nio-7900-Acceptor-0	main	5	RUNNABLE	0	0:1	false	true
30	http-nio-7900-ClientPoller-0	main	5	RUNNABLE	0	0:3	false	true
31	http-nio-7900-ClientPoller-1	main	5	RUNNABLE	0	0:3	false	true
20	http-nio-7900-exec-1	main	5	WAITING	0	1:56	false	true
29	http-nio-7900-exec-10	main	5	WAITING	0	1:55	false	true
21	http-nio-7900-exec-2	main	5	WAITING	0	1:56	false	true
22	http-nio-7900-exec-3	main	5	WAITING	0	1:56	false	true

Memory	used	total	max	usage	GC	
heap	210M	511M	511M	41.18%	gc.ps_scavenge.count	76
ps_eden_space	167M	254M	254M	66.10%	gc.ps_scavenge.time(ms)	731
ps_survivor_space	178K	1024K	1024K	17.42%	gc.ps_marksweep.count	0
ps_old_gen	42M	256M	256M	16.54%	gc.ps_marksweep.time(ms)	0
nonheap	109M	114M	-1	96.26%		
code_cache	25M	26M	240M	10.67%		
metaspace	74M	78M	-1	95.85%		
compressed_class_space	9M	10M	1024M	0.94%		
direct	95K	95K	-	100.00%		

Runtime	
os.name	Linux
os.version	3.10.0-862.9.1.el7.x86_64
java.version	1.8.0_212
java.home	/usr/local/openjdk-8/jre
systemload.average	0.93
processors	2
uptime	56564s

新增jvm参数 -

`XX:NativeMemoryTracking=detail`

再上一步中, 除了通过 arthas , 还可以通过 NMT 工具 (当然推荐使用 arthas 直观方便)

查看 native memory 使用, 在 java 进程启动后, 使用 `jcmd {pid} native_memory` 查看

NativeMemoryTracking可以追踪到堆内内存、code区域、通过unsafe.allocateMemory和 DirectByteBuffer申请的 内存， 但是**追踪不到其他 native code（c代码）申请的堆外内存**。 (参考: spring boot 引起的“堆外内存泄漏”)

下面是压测之后的 native_memory 显示, 没有明显异常, 堆外内存在可

接受范围内

```
root@f6e498ebdfca:/# jcmd 7 VM.native_memory
7:
Native Memory Tracking:
Total: reserved=2015487KB, committed=754723KB
-                 Java Heap (reserved=524288KB,
committed=524288KB)
                            (mmap: reserved=524288KB,
committed=524288KB)

-                     Class (reserved=1122935KB,
committed=84599KB)
                            (classes #14503)
                            (malloc=4727KB #19725)
                            (mmap: reserved=1118208KB,
committed=79872KB)

-                    Thread (reserved=43366KB,
committed=43366KB)
                            (thread #43)
                            (stack: reserved=43148KB,
committed=43148KB)
                            (malloc=137KB #214)
                            (arena=81KB #82)

-                      Code (reserved=255229KB,
committed=32801KB)
                            (malloc=5629KB #8670)
                            (mmap: reserved=249600KB,
committed=27172KB)

-                        GC (reserved=22459KB,
committed=22459KB)
                            (malloc=3471KB #294)
                            (mmap: reserved=18988KB,
committed=18988KB)
```

```
-                  Compiler (reserved=272KB, committed=272KB)
                            (malloc=141KB #643)
                            (arena=131KB #5)

-                  Internal (reserved=23050KB,
committed=23050KB)
                            (malloc=23018KB #18783)
                            (mmap: reserved=32KB,
committed=32KB)

-                    Symbol (reserved=20025KB,
committed=20025KB)
                            (malloc=16820KB #172346)
                            (arena=3205KB #1)

-    Native Memory Tracking (reserved=3687KB, committed=3687KB)
                            (malloc=193KB #2738)
                            (tracking overhead=3494KB)

-               Arena Chunk (reserved=178KB, committed=178KB)
                            (malloc=178KB)
```

怀疑是 直接内存未释放

`-XX:MaxDirectMemorySize=256M` 设置最大直接内存

通过下面代码查看最大的 直接内存

```
System.out.println(sun.misc.VM.maxDirectMemory());
```

但是 执行进行压测后, 还是有明显的直接内存泄漏

另: 其实这一步可以省略, 在 arthas / NMT 中已经分析出来没有泄漏了.

会不会是 jni 导致的内存泄漏

观察 jni global reference 引起的 memory leak

```
jstack {pid} | grep JNI
```

global reference 的数量一直稳定不变, 排除

进程层面的分析

怀疑是 C++算法包导致的内存泄漏

windows 下

修改源码 xxx.cpp , 重新编译

将未使用算法包, 但正常返回的jni接口, 并且**使用 vs 附加到进程 ctrl+alt+P**, 通过诊断工具发现, 在经过 5000 次调用之后, 内存无明显变化

但使用算法包的情况下, 调用 5000 次, 内存不断增长, 导致操作系统卡死

linux 下

修改 xxx.cpp , 重新编译

实际调用 2000 次, 增长内存 10M , 变化不大

使用原来的算法库, 调用 2800次, 增长内存 280M

猜测可能是 C++ 算法包的问题 -> 成功丢出一个 bug =.= 事后发现 C++ 代码中有很多内存没有释放掉才导致了这个问题

后记

其实这个项目的这个 bug 存在蛮久的了, 之前也观察到内存泄漏的情况, 但不知道问题出在哪里. 项目调用的东西很多, 包含堆外直接内存, JNI , JNA 等, 一直没找到解决这类问题的思路.

后面这个问题越来越严重, 查了很多资料 (参考: spring boot 引起的“堆外内存泄漏”), 非常感谢这篇文章中排查问题的过程, 给我提供了一个清晰的思路

下面还是总结下 **思路**, 方便之后排查 **内存泄漏** 的问题

1. 先使用 **Java层面** 的工具 **arthas** 定位哪些地方可能导致内存泄漏
    - 堆内内存
    - code区域
    - 使用 **unsafe.allocateMemory** 和 **DirectByteBuffer **申请

的堆外内存

2. JNI 层面的泄漏
    - 检查 jni global reference 是否未释放
    - 检查 jni LocalReference 生命周期是否过长导致潜在的内存泄漏
3. Native Code 本身的内存泄漏

本篇中的例子就是 C++ 算法包导致的内存泄漏

参考

- 在 JNI 编程中避免内存泄漏
    - https://www.ibm.com/developerworks/cn/java/j-lo-jnileak/
- spring boot 引起的 “堆外内存泄漏”
- https://mp.weixin.qq.com/s/73whP7E3SIB5mn_TLrqT_w