# JVM实践

2024年4月1日 19:34

## JVM 调优的参数可以在哪里设置参数值

- war包部署在tomcat中设置

修改TOMCAT_HOME/bin/catalina.sh文件

- jar包部署在启动参数设置

java -Xms512m -Xmx1024m -jar xxxx.jar

## 用的 JVM 调优的参数都有哪些?

对于JVM调优，主要就是调整年轻代、老年代、元空间的内存空间大小及使用的垃圾回收器类型。

https://www.oracle.com/java/technologies/javase/vmoptions-jsp.html

- 设置堆空间大小
- 虚拟机栈的设置
- 年轻代中Eden区和两个Survivor区的大小比例
- 年轻代晋升老年代阈值
- 设置垃圾回收收集器

## 用的 JVM 调优的参数都有哪些?

- 设置堆空间大小

设置堆的初始大小和最大大小，为了防止垃圾收集器在初始大小、最大大小之间收缩堆而产生额外的时间，通常把最大、初始大小设置为相同的值。

```
-Xms：设置堆的初始化大小
-Xmx：设置堆的最大大小

-Xms:1024
-Xms:1024k
-Xms:1024m
-Xms:1g
```

不指定单位默认为字节

指定单位，按照指定的单位设置

堆空间设置多少合适?

- 最大大小的默认值是物理内存的1/4，初始大小是物理内存的1/64
- 堆太小，可能会频繁的导致年轻代和老年代的垃圾回收，会产生stw，暂停用户线程
- 堆内存大肯定是好的，存在风险，假如发生了fullgc,它会扫描整个堆空间，暂停用户线程的时间长
- 设置参考推荐：尽量大，也要考察一下当前计算机其他程序的内存使用情况

## 用的 JVM 调优的参数都有哪些?

- 虚拟机栈的设置

虚拟机栈的设置：每个线程默认会开启1M的内存，用于存放栈帧、调用参数、局部变量等，但一般256K就够用。通常减少每个线程的堆栈，可以产生更多的线程，但这实际上还受限于操作系统。

```
-Xss  对每个线程stack大小的调整,-Xss128k
```

## 用的 JVM 调优的参数都有哪些?

- 年轻代中Eden区和两个Survivor区的大小比例

设置年轻代中Eden区和两个Survivor区的大小比例。该值如果不设置，则默认比例为8:1:1。通过增大Eden区的大小，来减少YGC发生的次数，但有时我们发现，虽然次数减少了，但Eden区满的时候，由于占用的空间较大，导致释放缓慢，此时STW的时间较长，因此需要按照程序情况去调优。

```
-XXSurvivorRatio=8，表示年轻代中的分配比率：survivor:eden = 2:8
```

- 年轻代晋升老年代阈值

```
-XX:MaxTenuringThreshold=threshold
```

  - 默认为15
  - 取值范围0-15

## 用的 JVM 调优的参数都有哪些?

- 设置垃圾回收收集器

通过增大吞吐量提高系统性能，可以通过设置并行垃圾回收收集器。

```
-XX:+UseParallelGC
-XX:+UseParallelOldGC
```

```
-XX:+UseG1GC
```

## 说一下 JVM 调优的工具?

- 命令工具
  - jps　　进程状态信息
  - jstack　查看java进程内线程的堆栈信息
  - jmap　　查看堆转信息
  - jhat　　堆转储快照分析工具
  - jstat　　JVM统计监测工具
- 可视化工具
  - jconsole　用于对jvm的内存，线程，类 的监控
  - VisualVM　能够监控线程，内存情况

## 说一下 JVM 调优的工具?

- jps

进程状态信息

```
C:\Users\yuhon>jps
27920 Jps
27348 Launcher
28472 Application
6140
```

- jstack

查看java进程内线程的堆栈信息

```
jstack [option] <pid>
```

```
"Reference Handler" #2 daemon prio=10 os_prio=2 tid=0x000001657fbfe000 nid=0x3274 in Object.wait()
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0x0000000716a06c00> (a java.lang.ref.Reference$Lock)
        at java.lang.Object.wait(Object.java:502)
        at java.lang.ref.Reference.tryHandlePending(Reference.java:191)
        - locked <0x0000000716a06c00> (a java.lang.ref.Reference$Lock)
        at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:153)

"main" #1 prio=5 os_prio=0 tid=0x000001657a445800 nid=0x41d4 runnable [0x0000001d359ff000]
   java.lang.Thread.State: RUNNABLE
        at com.heima.jvm.Application.main(Application.java:9)
```

## 说一下 JVM 调优的工具?

- jmap

用于生成堆转内存快照、内存使用情况

```
jmap -heap pid   显示Java堆的信息
jmap -dump:format=b,file=heap.hprof pid
```

- format=b表示以hprof二进制格式转储Java堆的内存
- file=<filename>用于指定快照dump文件的文件名。

知识小贴士

它是一个进程或系统在某一给定的时间的快照。比如在进程崩溃时，甚至是任何时候，我们都可以通过工具将系统或某进程的内存备份出来供调试分析用。

dump文件中包含了程序运行的模块信息、线程信息、堆栈调用信息、异常信息等数据，方便系统技术人员进行错误排查。

## 说一下 JVM 调优的工具?

- jstat

是JVM统计监测工具。可以用来显示垃圾回收信息、类加载信息、新生代统计信息等。

①：总结垃圾回收统计　　jstat -gcutil pid

```
D:\code\jvm-demo\target\classes\com\heima\jvm>jstat -gcutil 28472
  S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT     GCT
  0.00   0.00   8.00   0.00  17.38  19.94       0    0.000     0    0.000    0.000
```

②：垃圾回收统计　　jstat -gc pid

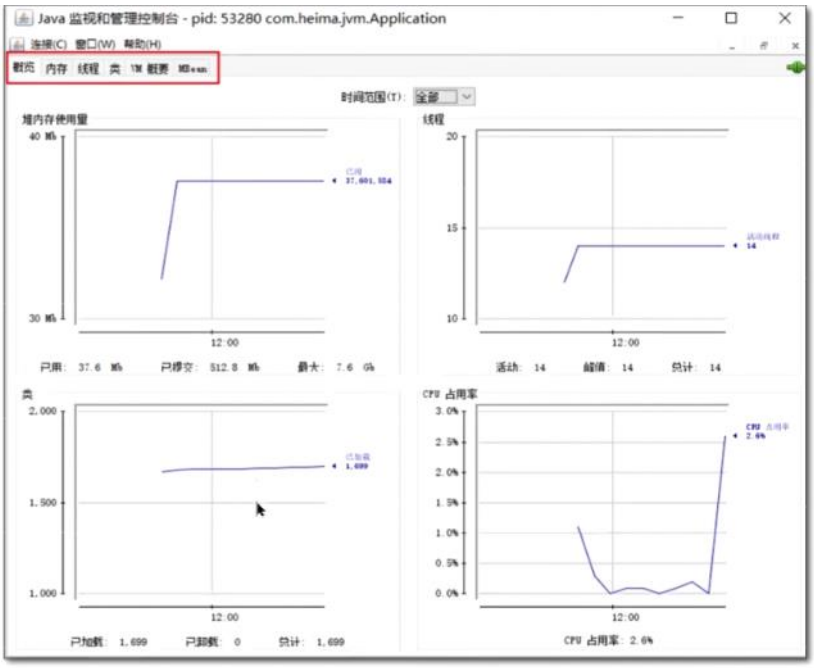
```
D:\code\jvm-demo\target\classes\com\heima\jvm>jstat -gc 28472
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC     MU    CCSC   CCSU   YGC     YGCT    FGC    FGCT     GCT
21504.0 21504.0  0.0    0.0   131072.0 10485.8   348160.0     0.0     4480.0 778.5  384.0   76.6       0    0.000   0      0.000    0.
000
```

## 说一下 JVM 调优的工具?

- jconsole

用于对jvm的内存，线程，类 的监控，是一个基于 jmx 的 GUI 性能监控工具

打开方式：java 安装目录 bin目录下 直接启动 jconsole.exe 就行

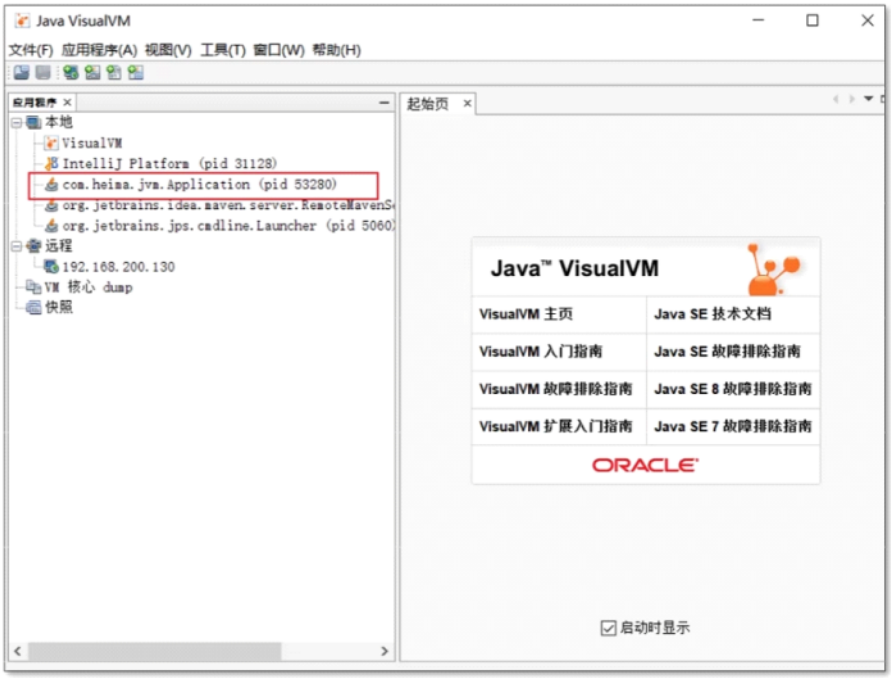



## 说一下 JVM 调优的工具?

- VisualVM

能够监控线程，内存情况，查看方法的CPU时间和内存中的对 象，已被GC的对象，反向查看分配的堆栈

打开方式：java 安装目录 bin目录下 直接启动 jvisualvm.exe就行



## 说一下 JVM 调优的工具?

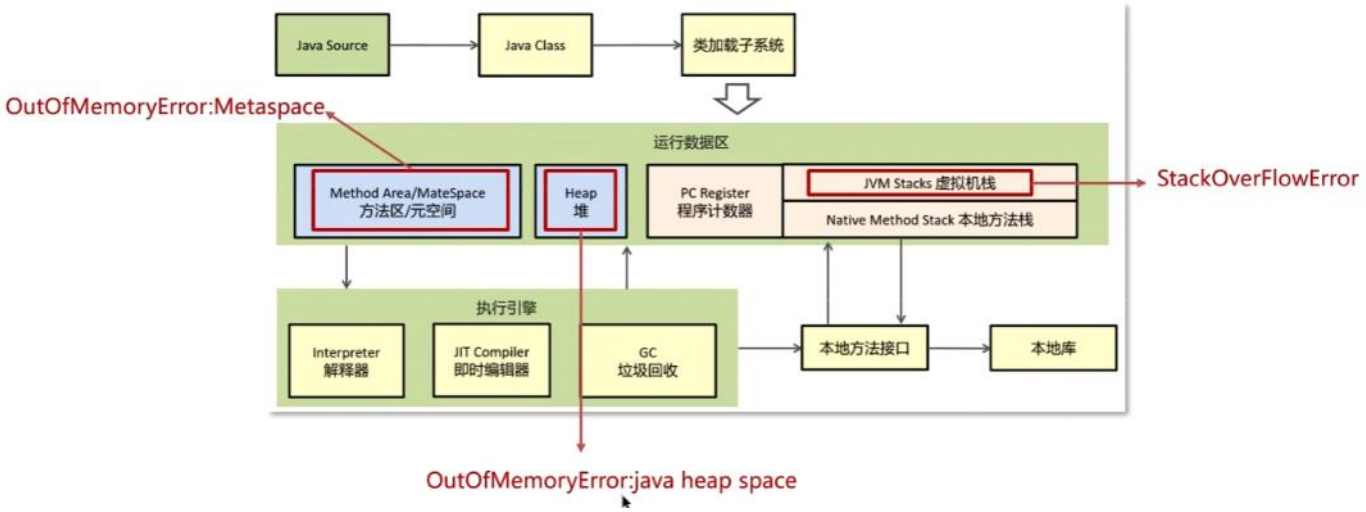
命令工具

- jps　　进程状态信息
- jstack　查看java进程内线程的堆栈信息
- jmap　　查看堆转信息
- jhat　　堆转储快照分析工具
- jstat　　JVM统计监测工具

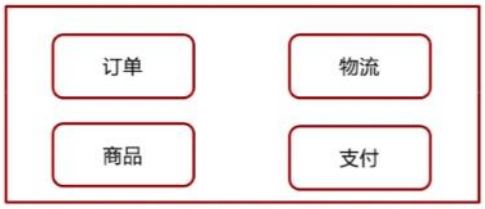
可视化工具

- jconsole　用于对jvm的内存，线程，类 的监控
- VisualVM　能够监控线程，内存情况

## java内存泄露的排查思路?




## java内存泄露的排查思路?




服务器

- 启动闪退
- 运行一段时间宕机

1. 获取堆内存快照dump
2. VisualVM去分析dump文件
3. 通过查看堆信息的情况，定位内存溢出问题

## java内存泄露的排查思路?

1、通过jmap指定打印他的内存快照dump(Dump文件是进程的内存镜像。可以把程序的执行状态通过调试器保存到dump文件中)

- 使用jmap命令获取运行中程序的dump文件

```
jmap -dump:format=b,file=heap.hprof pid
```

- 使用vm参数获取dump文件

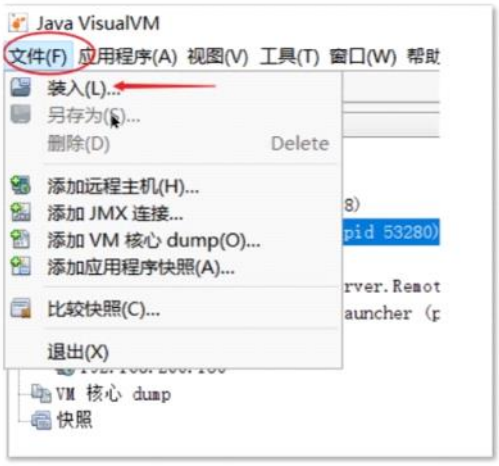
有的情况是内存溢出之后程序则会直接中断，而jmap只能打印在运行中的程序，所以建议通过参数的方式的生成dump文件

```
-XX:+HeapDumpOnOutOfMemoryError
-XX:HeapDumpPath=/home/app/dumps/
```

## java内存泄露的排查思路?

2、通过工具， VisualVM去分析dump文件，VisualVM可以加载离线的dump文件

文件-->装入--->选择dump文件即可查看堆快照信息




## java内存泄露的排查思路?

3、通过查看堆信息的情况，可以大概定位内存溢出是哪行代码出了问题


4、找到对应的代码，通过阅读上下文的情况，进行修复即可

## CPU飙高排查方案与思路?

1.使用top命令查看占用cpu的情况

```
top
```

```
  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
40940 root      20   0 2243512  28436  11604 S 88.3  2.8   0:32.06 java
 1442 root      20   0  159268   6100   4304 S  0.7  0.6   0:08.58 sshd
 1895 root      20   0  159164   6032   4304 S  0.7  0.6   0:07.77 sshd
24446 root      20   0  159164   6020   4312 R  0.3  0.6   0:02.42 sshd
```

2.通过top命令查看后，可以查看是哪一个进程占用cpu较高，上图所示的进程为：40940

## CPU飙高排查方案与思路?

3.查看进程中的线程信息

```
ps H -eo pid,tid,%cpu | grep 40940
```

```
[root@myhbase ~]# ps H -eo pid,tid,%cpu | grep 40940
40940 40940  0.0
40940 40941  0.0
40940 40942  0.0
40940 40943  0.0
40940 40944  0.0
40940 40945  0.0
40940 40946  0.0
40940 40947  0.0
40940 40948  0.0
40940 40949  0.0
40940 40950 87.9
40940 40955  0.0
40940 40958  0.0
```

通过以上分析，在进程40940中的线程40950占用cpu较高

## CPU飙高排查方案与思路?

4.可以根据线程 id 找到有问题的线程，进一步定位到问题代码的源码行号

```
jstack 40940  此处是进程id
```

```
"thread1" #8 prio=5 os_prio=0 tid=0x00007fcf580f5000 nid=0x9ff6 runnable [0x00007fcf355b7000]
   java.lang.Thread.State: RUNNABLE
        at Application.lambda$main$0(Application.java:9)
        at Application$$Lambda$1/531885035.run(Unknown Source)
        at java.lang.Thread.run(Thread.java:748)

"Service Thread" #7 daemon prio=9 os_prio=0 tid=0x00007fcf580b3800 nid=0x9ff4 runnable [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"C1 CompilerThread1" #6 daemon prio=9 os_prio=0 tid=0x00007fcf580b0800 nid=0x9ff3 waiting on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"C2 CompilerThread0" #5 daemon prio=9 os_prio=0 tid=0x00007fcf580ae800 nid=0x9ff2 waiting on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE
```

十六进制

```
40940 40949  0.0
40940 40950 87.9
40940 40955  0.0
40940 40958  0.0
```

十进制

十进制转换为十六进制

```
printf "%x\n" 40955
```

```
[root@myhbase ~]# printf "%x\n" 40955
9ffb
```

## CPU飙高排查方案与思路?

1.使用top命令查看占用cpu的情况

2.通过top命令查看后，可以查看是哪一个进程占用cpu较高

3.使用ps命令查看进程中的线程信息

4.使用jstack命令查看进程中哪些线程出现了问题，最终定位问题