杨七

昵称： 杨七
园龄： 3年5个月
粉丝： 11
关注： 4
+加关注

JVM OOM分析与调优

OutOfMemoryError

除了程序计数器外，其余的几个运行数据区都有可能发生OutOfMemoryError（OOM）的可能。

因此在遇到OOM的问题时应能根据异常的信息快速定位到时哪个内存区域的内存溢出，知道什么样的代码会导致OOM，以及该如何处理。

1、Java堆溢出

Heap堆是OOM故障最主要的发源地，它存储着几乎所有的实例对象。在线上生产环境中，JVM的Xms和Xmx应设置成一样的大小，避免在GC后调整堆大小时带来额外的压力。

模拟：不停的创建对象，并且GC Roots到对象之间有可达路径，超过堆的最大内存容量就会OOM。如，在while(true)循环中不停的创建对象并将对象add到List集合中

Java堆内存溢出时异常堆栈信息会在“java.lang.OutOfMemoryError”后提示Java heap space

如何定位?

1）设置参数-XX:+HeapDumpOnOutOfMemoryError 来配置当内存耗尽时记录下当时的内存快照，一般根据MAT分析具体原因是内存泄漏还是内存溢出

2）内存泄漏：根据工具查看泄漏对象到GC Roots的引用链，找到泄漏对象时通过怎么样的路径与GC Roots相关联并导致垃圾收集器无法自动回收它们的。根据对象类型和GC Roots引用链就可以准确定位到泄漏代码的位置。

3）内存溢出：检查虚拟机参数堆参数能否扩大，代码上检查是否存在某些对象生命周期过长、持有状态时间过长。

我的随笔
我的评论
我的参与
最新评论
我的标签

随笔分类
Dubbo(1)
ElasticSearch(1)
HTTP(9)
Java基础(23)
JDK源码解读(11)
JVM & JMM(10)
Maven(2)
MongoDB(1)
MyBatis(5)
MySQL(11)

2、虚拟机栈和本地方法栈溢出

HotSpot虚拟机不区分虚拟机栈和本地方法栈，因此设置本地方法栈的参数-Xoss实际上是无效的，栈的参数只能根据-Xss设定

如果线程请求的栈深度超过了JVM允许的最大深度，将抛出StackOverflowError

如果虚拟机在扩展栈时无法申请到足够的内存将跑出OutOfMemoryError

1）StackOverflowError 容易定位，可能是递归没有出口，正常的方法调用深度在1000-2000之间完全没有问题

2）OOM可能是建立的线程数量非常多，每个线程瓜分栈内存，当线程都存活着没有足够的内存去分配给线程时会抛出OOM

3、方法区和运行时常量池溢出

Java方法区和运行时常量池溢出异常堆栈信息会在“java.lang.OutOfMemoryError”后提示PermGen space

如String字符串存放在常量池中

4、本机直接内存溢出

Unsafe的allocateMemory可以申请分配内存（Unsafe实例可通过反射获取）

由DirectMemory导致的内存溢出在Heap Dump文件中看不出明显的异常，如果OOM之后的Dump文件很小，而程序中直接或者间接使用了NIO，那可能是这方面的原因。

DirectMemory-XX:MaxDirectMemory指定，如果不指定默认与Java堆内存最大值一样。

垃圾收集时，虚拟机 只有在Full GC时会顺便回收DirectMemory中废弃的对象。因此DirectMemory内存满了之后，只有等待系统的下一次Full GC，或者抛出内存异常在catch中调用System.gc()

内存溢出排查原因

1、GC日志分析

为了在内存溢出时排查原因，可以在JVM启动时加一些参数来控制，当JVM内存出问题时可以通过分析记录下来的GC日志，GC的频率和每次GC回收了哪些内存。

GC的日志输入有以下参数

1）-verbose:gc 可以辅助输出一些详细的GC信息

2）-XX:+PrintGCDetails 输出GC的详细信息

Nginx(1)

Redis(12)

Spring(33)

并发编程(13)

分布式服务相关(6)

更多

随笔档案

2021年2月(6)

2021年1月(9)

2020年12月(4)

2020年11月(2)

2020年10月(6)

2020年9月(4)

2020年8月(3)

2020年7月(1)

2020年6月(1)

3）-XX:+PrintGCApplicationStoppedTime 输出GC造成应用程序暂停的时间

4）-XX:+PrintGCDateStamps 输出GC发生的时间信息

5）-XX:PrintHeapAtGC 在GC前后输出堆中各个区域的大小

6）-Xloggc:[file] 将GC信息输出到单独的文件

每种GC方式输出日志的形式不同，除CMS的日志和其他GC方式差异较大外，其余GC方式的日志可以抽象成如下方式

[GC [<collector>: <starting occupancy1> -> <ending occupancy1> (total size1) , <pause time1> secs]

<starting occupancy2> -> <ending occupancy2> (total size2) , <pause time2> secs]]

说明如下

1）<collectot>GC 表示垃圾收集器的名称

2）<starting occupancy1> 表示Young区在GC前占用的内存

3）<ending occupancy1> 表示Young区在GC后占用的内存

4）(total size1) 表示Young区的总内存大小

5）<pause time1> 表示Young区局部收集时JVM暂停处理的时间 secs表示单位秒

6）<starting occupancy2> 表示Heap在GC前占用的内存

7）<ending occupancy2> 表示Heap在GC后占用的内存

8）(total size2) 表示Heap的总内存

9）<pause time2> 表示在GC过程中JVM暂停处理的总时间

可以根据日志来判断是否存在内存泄漏的问题：

<starting occupancy1> - <ending occupancy1> 和 <starting occupancy2> - <ending occupancy2> 比较

2020年5月(1)
2020年4月(1)
2020年3月(16)
2020年2月(3)
2019年12月(4)
2019年11月(2)
更多

最新评论

1. Re:ConcurrentHashMap详解

引用网友的例子描述：既然不能全锁（HashTable）又不能不锁（HashMap），所以就搞个部分锁，只锁部分，用到哪部分就锁哪部分。一个大仓库，里面有若干个隔间，每个隔间都有锁，同时只允许一个人进隔...

--杨七

阅读排行榜

1. SimpleDateFormat线程不安全原因及解决方案(19550)

1、如果前者差等于后者差，表明Young区GC 对象100%被回收，没有对象进入 Old区或者Perm区

2、如果前者大于后者，那么差值就是这次GC对象进入Old或者Perm区的大小

如果随着时间的的延长，<ending occupancy2>的大小一直在增长，而且Full GC很频繁，那么很可能就是内存泄漏导致的。

2、堆快照文件分析

如果是OOM，可能有两方面的原因

1、内存分配过小，不满足程序运行所需要的内存

2、内存泄漏（FullGC频繁，回收后Heap占用的内存不断增长）

以下Java命令在JDK的bin目录下执行

1、导出堆dump文件分析

1）jps -m -l 命令找到我们服务的进程

2）jmap -dump:format=b,file=[filename] [pid]　　jmap（Memory Map for Java）来记录下堆的内存快照，然后利用第三方工具如eclipse 插件MAT、IBM HeapAnalyzer来分析整个Heap的对象关联情况。

IBM HeapAnalyzer：Steps

① Download: https://public.dhe.ibm.com/software/websphere/appserv/support/tools/HeapAnalyzer/ha457.jar

② Open a terminal or command prompt and change directory to where you downloaded the JAR file.

③ Ensure that Java is on your PATH to run the tool.

④ Launch the tool (increase -Xmx based on your available RAM)：java -Xmx2g -jar ha*.jar

如果内存耗尽可直接导致JVM退出，可以通过参数

-XX:+HeapDumpOnOutOfMemoryError 来配置当内存耗尽时记录下当时的内存快照

-XX:HeapDumpPath 指定内存快照文件的路径 文件快照的名称格式为 java_[pid].hprof

2、在线分析

1）jps -m -l 命令找到我们服务的进程

2）使用jmap -heap pid查看进程堆内存使用情况，包括使用的GC算法、堆配置参数和各代中堆内存使用情况。如：

```
./jmap -heap 27675
Attaching to process ID 27675, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.202-b08

using parallel threads in the new generation.
using thread-local object allocation.
Concurrent Mark-Sweep GC

Heap Configuration:
   MinHeapFreeRatio         = 40
   MaxHeapFreeRatio         = 70
   MaxHeapSize              = 536870912 (512.0MB)
   NewSize                  = 268435456 (256.0MB)
   MaxNewSize               = 268435456 (256.0MB)
   OldSize                  = 268435456 (256.0MB)
   NewRatio                 = 2
   SurvivorRatio            = 3
   MetaspaceSize            = 21807104 (20.796875MB)
   CompressedClassSpaceSize = 939524096 (896.0MB)
   MaxMetaspaceSize         = 1073741824 (1024.0MB)
   G1HeapRegionSize         = 0 (0.0MB)

Heap Usage:
New Generation (Eden + 1 Survivor Space):
```

```
   capacity = 214761472 (204.8125MB)
   used     = 5953088 (5.67730712890625MB)
   free     = 208808384 (199.13519287109375MB)
   2.7719534349252366% used
Eden Space:
   capacity = 161087488 (153.625MB)
   used     = 5953088 (5.67730712890625MB)
   free     = 155134400 (147.94769287109375MB)
   3.695562004170057% used
From Space:
   capacity = 53673984 (51.1875MB)
   used     = 0 (0.0MB)
   free     = 53673984 (51.1875MB)
   0.0% used
To Space:
   capacity = 53673984 (51.1875MB)
   used     = 0 (0.0MB)
   free     = 53673984 (51.1875MB)
   0.0% used
concurrent mark-sweep generation:
   capacity = 268435456 (256.0MB)
   used     = 75364008 (71.87271881103516MB)
   free     = 193071448 (184.12728118896484MB)
   28.075280785560608% used

28582 interned Strings occupying 3079592 bytes.
```

3）使用 jmap -histo[:live] pid 查看堆内存中的对象数目、大小统计直方图，如果带上live则只统计活对象，如：

```
 ./jmap -histo:live 27675|less
 num     #instances         #bytes  class name
----------------------------------------------
   1:        277691       32177840  [C
   2:        198142        4755408  java.lang.String
```

3、JVM Crash 日志分析

TODO

JVM性能监控和故障处理

通过工具导出和处理分析 运行日志、异常堆栈、GC日志、线程快照（threaddump/javacore文件）、堆转储快照（headdump/hprof文件）等

jps：JVM process status tool，显示指定系统内所有的HotSpot虚拟机进程

jstat：JVM statistics Monitoring Tool，收集HotSpot虚拟机各方面的运行数据

jinfo：Configuration Info for java，显示虚拟机配置信息

jmap：Memory map for Java，生成虚拟机的内存转储快照（heapdump文件）

jhat：JVM Heap Dump Browser，用于分析heapdump文件，它会建立一个http/html服务器，让用户可以在浏览器上查看分析结果

jstack：Stack trace for Java，显示虚拟机的线程快照

jps：（Jvm Process Status ）和linux中ps命令相似，可列出正在运行的虚拟机进程，并显示虚拟机执行主类和这些进程的唯一ID（Local Virtual Machine Identifier LVMID）

格式：jps [option] [hostid(主机名)]

参数：

-l 输出主类的全路径

-v 输出虚拟机进程启动是的JVM参数

-m 输出传入main方法的参数

一般使用 jps -m -l 来查看虚拟机进程

jstat：用于监视虚拟机各种运行状态信息的命令行工具，它可以显示本地或者远程虚拟机进程中的类加载、内存、垃圾收集、JIT编译等运行数据，是定位虚拟机性能问题的首选工具。

格式：jstat [option vmid [interval] [count]]

interval和count是查询间隔和次数，如果忽略这两个参数则只查询一次

选项：

-class 监视类装载、卸载数量、总空间以及类装载消耗时间

-gc 监视Java堆状况，包含Eden区，两个Survivor区、老年代、永久代等的容量，已用空间，GC时间等信息。

如： jstat -gc 27675 250 4 采样时间间隔为250ms，采样数为4：

```
 jstat -gc 27675 250 4
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC         MU      CCSC     C
52416.0 52416.0  0.0    0.0   157312.0 37334.0   262144.0   73597.7   129932.0 77136.4 9824.0 91
52416.0 52416.0  0.0    0.0   157312.0 37334.0   262144.0   73597.7   129932.0 77136.4 9824.0 91
52416.0 52416.0  0.0    0.0   157312.0 37334.0   262144.0   73597.7   129932.0 77136.4 9824.0 91
52416.0 52416.0  0.0    0.0   157312.0 37334.0   262144.0   73597.7   129932.0 77136.4 9824.0 91
```

S0C、S1C、S0U、S1U：Survivor 0/1区容量（Capacity）和使用量（Used）
EC、EU：Eden区容量和使用量
OC、OU：年老代容量和使用量
PC、PU：永久代容量和使用量
YGC、YGT：年轻代GC次数和GC耗时
FGC、FGCT：Full GC次数和Full GC耗时
GCT：GC总耗时

jmap：Java内存映射工具，用于生成堆存储快照heapdump，生成堆快照还可以通过设置参数使在OOM异常之后自动生成堆dump文件。

jmap还可查询finalize执行队列、Java堆和永久代的详细信息（空间使用率，使用哪种收集器等）。

格式：jmap [option] vmid

option参数

-dump:format=b,file=[filename] 生成堆转储快照

-heap 显示Java堆详细信息，如使用哪种收集器、参数配置、分代状况等。

如：jmap -dump:format=b,file=/home/dump.hprof [pid]

jhat：与jmap搭配使用，分析jmap生成的堆转储快照文件。一般不会直接使用jhat命令分析dump文件，一是不会直接在应用服务器上分析dump文件，因为分析耗时耗资源，二是jhat分析结果比较简陋，可用VisualVM，MAT等工具

jstack：Java堆栈跟踪工具，用户生成虚拟机当前时刻的线程快照，线程快照就是当前虚拟机内每一条线程正在执行的方法堆栈的集合，**生成快照的主要目的是定位线程出现长时间等待的原因，如线程间死锁、死循环、请求外部资源（如sql）导致的长时间等待等。**

线程出现停顿的时候通过jstack来查看各个线程的调用堆栈，就可以知道没有响应的线程到底在后台做什么事情或者等待什么资源

格式：jstack [option] vmid

option参数：

-F：强制输出线程堆栈

-l：除堆栈外显示关于锁的附加信息

-m：可显示调用本地方法的堆栈

JDK1.5之后的Thread类新增了getAllStackTraces()方法用户获取虚拟机中所有线程的StackTraceElement对象。和jstack功能类似

public static Map<Thread,StackTraceElement[]> **getAllStackTraces**() 返回从 Thread 到 StackTraceElement 数

总结:

如果要定位OOM问题使用jps和jmap组合命令，先用jps或者linux的ps命令查看虚拟机进程的vmid，然后用jamp命令

如果要定位线程响应时间过长的问题，使用jps和jstack命令，先用jps或者linux的ps命令查看虚拟机进程的vmid，然后

JVM调优

JVM调优是通过分析GC日志等来分析java内存和垃圾回收的情况，来调整各内存区域内存占比和垃圾回收策略。

GC优化的根本原因

垃圾收集器的工作就是清除Java创建的对象，垃圾收集器需要清理的对象数量以及要执行的GC数量均取决于已创建的对

如：养成如下的良好的编码习惯

1、使用StringBuffer或StringBuilder来代替String

2、尽量少输出日志

GC优化的两个目的

1、减少Full GC的频率

2、减少Full GC的执行时间

1、减少Full GC的频率——将进入老年代的对象数量降到最低

充分使用系统资源，减少GC停顿时间和停顿次数，由于Full GC的停顿时间远比Minor GC的停顿时间长，因此要控制Full GC的频率。

控制Full GC的频率的关键是将进入老年代的对象数量降到最低。因此要看应用中的绝大多数对象是否符合“朝生夕灭”的原则，即大多数的对象的生存时间都不应太长，尤其是不能有成批量的、长时间存活的对象产生，这样这些对象在Minor GC就会被回收，不会进入老年代，这样才能保证老年代的稳定。

比如对于十几小时乃至一天才出现一次Full GC的系统可以通过定时任务的方式在夜间触发Full GC。

如果FullGC次数过多可能是下面的原因：

1、内存占用高：代码中创建了大量的对象导致内存泄漏，不能回收内存，创建新对象导致空间不足触发fullGC

2、内存占用不高：可能是显示的调用System.gc()次数太多导致的fullGC，可以通过添加-XX:+DisableExplicitGC来禁用JVM对显式GC的响应

2、减少Full GC的时间

Full GC的停顿时间远比Minor GC的停顿时间长，因此，如果在Full GC上花费过多的时间（超过1s）， 将可能出现超时错误。

1）如果通过减小老年代内存来减少Full GC时间，可能会引起OutOfMemoryError或者导致Full GC的频率升高。

2）如果通过增加老年代内存来降低Full GC的频率，Full GC的时间可能因此增加。

因此，你需要把老年代的大小设置成一个“合适”的值。

影响GC性能的因素：

1）-XX:NewRatio 新生代和老年代的内存比（默认1:2），这个参数将对GC性能产生重要的影响

2）垃圾收集器的类型

什么时候需要进行GC优化?

通过监控GC状态，然后分析GC监控结果来判断是否需要进行GC优化。

如果GC执行时间满足下列所有条件，就没有必要进行GC优化了：

1）Minor GC执行非常迅速（50ms以内）

2）Minor GC没有频繁执行（大约10s执行一次）

3）Full GC执行非常迅速（1s以内）

4）Full GC没有频繁执行（大约10min执行一次）

GC优化是一个不断尝试并逐渐调试的过程。在每次设置完GC参数后，要收集数据，并收集至少24个小时之后再进行结

分类: JVM & JMM

好文要顶 关注我 收藏该文

杨七
关注 - 4
粉丝 - 11
+加关注

0 0

« 上一篇： JVM内存结构
» 下一篇： 监听kafka消息

posted @ 2020-03-18 09:53 杨七 阅读(364) 评论(0) 编辑 收藏

刷新评论 刷新页面 返回顶部

登录后才能发表评论，立即 登录 或 注册， 访问 网站首页