第五章：调优分析与实战

1. **大内存硬件**上的程序部署策略

一个在线文档类型网站更换了硬件系统，物理内存更换为16GB，因此堆内存大小由原来的1.5GB提升到12GB。

问题：**不定期长时间失去响应**。

原因：**过大的堆内存回收时长时间停顿。**

垃圾回收停顿导致的，虚拟机以服务端模式运行，默认使用吞吐量优先收集器，回收12GB的堆时，一次Full GC停顿时间高达14s。程序设计原因，访问文档时会把文档从磁盘提取到内存中，导致内存出现许多因文档序列化产生的大对象，直接进入老年代，内存很快耗尽使得Full GC频繁。

在之前分配1.5GB内存时并没有明显的长时间停顿，修改当前堆内存为2GB也会减少停顿，但是会浪费内存。

单体应用在较大内存硬件上主要部署方式有两种：

1 **一个单独的虚拟机管理堆内存**

2 **同时使用若干虚拟机，建立逻辑集群利用硬件资源**。

第一种方式：

可以使用Parallel Scavenge/Old收集器，并分配较大堆内存。前提是把**Full GC频率控制的足够低**，一天只发生一次Full GC，并且**在深夜执行定时任务的方式触发Full GC**。

控制Full GC频率关键是老年代稳定，即**不能有太多长时间存活对象和太多的大对**象。

第二种方式：**建立逻辑集群利用硬件资源。**

在一个物理机器上启动多个服务器进程，分配不同端口，在前端搭建一个负载均衡器，以反向代理模式分配访问请求。需求是利用硬件资源，因此使用无Session复制的亲和式集群，均衡器按照一定规则将固定用户请求永远分配到一个固定集群节点。

本例解决方案：**建立5个32位JDK的逻辑集群**，每个进程按2GB内存计算（堆为1.5GB），共10GB内存。前端使用Apache服务作为均衡代理访问。用户对响应速度有要求，且文档服务压力集中在**磁盘和内存访问**，因此改用**CMS收集器**。调整后，再没出现长时间停顿。

1. **集群间同步导致内存溢出**

使用亲合式集群，节点间无Session同步，但是有需求实现部分数据在各个节点间共享。最开始数据在数据库中，但是读写频繁竞争激烈使用了一个全局缓存。启用后使用了较长一段时间后出现不定期内存溢出问题。

在溢出后查看了堆转储dump快照，发现存在大量org.jgroups.protocols.pbcast.NAKACK对象，这个对象是JBossCache缓存保障各个包有效顺序及重发。

在服务使用时，一个页面产生数十次请求，导致各个集群节点之间网络交互频繁，网络情况不满足传输要求时，重发数据请求会不断在内存中累计，最终内存溢出。

解决：**可以读操作频繁，避免写操作频繁**。

1. **堆外内存导致溢出错误**

一个B/S电子考试系统，使用了逆向AJAX技术（Comet或Server Side Push）

问题：服务端不定时抛出内存溢出异常

尝试把堆内存调到最大，但是没有用，加入-XX:+HeapDumpOnOutOfMemoryError参数没有反应，抛出内存溢出异常时无任何文件产生。挂着jstat，发现垃圾收集器不频繁，各个区域内存都稳定，最后在系统日志中找到异常堆栈异常解释：java.lang.OutOfMemoryError:null，即直接内存溢出。

这个案例共2GB内存，1.6GB给了堆内存，直接内存只能分配在剩下0.4中的一部分。直接内存不会主动被回收，只能在Full GC时顺带回收，因此直接内存满了会直接抛出异常，本例使用的CometD 1.1.1有大量NIO操作需要使用直接内存，因此会造成溢出。

1. **外部命令造成系统缓慢**

一个校园应用请求响应慢，通过操作系统mpstat工具发现处理器使用率高，但是占用绝大数处理器资源的不是应用本身。

通过工具发现是fork系统调用最消耗处理器资源，产生新进程消耗资源。

问题：每个用户请求处理都要执行一个外部Shell脚本获得一些系统信息，调用时会复制一个和当前虚拟机拥有一样环境变量的进程，然后利用这个进程执行外部命令，频繁创建消耗很大。

解决：**去掉这个Shell脚本执行，改用使用JavaAPI获取信息**。

1. **服务器虚拟机进程崩溃**

使用异步方式调用web服务，但是两边服务速度不对等，时间长了累计了大量web服务没有调用，超出虚拟承受能力崩溃。

解决：**将异步调用改为生产者/消费者模式的消息队列实现**。

1. 不恰当数据结构导致内存占用过大

内存中有超过100万个HashMap<Long,Long>Entry，这个结构储存数据文件空间效率太低。

LongX2=16字节，包装成对象后，分别有8字节MarkWord、8字节Klass指针、8字节储存数据的long值。两个Long组成Map.Entry后，多了16字节对象头，一个8字节next和4字节int型hash字段，对齐还要加4字节对齐填充，还有HashMap对这个Entry的8字节引用，增加两个长整型数字实际耗费内存88字节。16/88=18%，效率太低。

1. **安全点导致的长时间停顿**

User：进程执行用户态代码耗费处理器时间

Sys：进程执行核心态代码所耗费处理器时间（指所有处理器总花费时间）

Real：执行动作从开始到结束耗费的时钟时间（指一共花费多长时间执行完）

这个问题就是部分线程走到了安全点，但是有些特别慢的没到，导致垃圾收集线程无法工作，只能自旋等待。

解决：找出最慢的几个线程，添加-XX：SafepointTimeout和-XX：SafepointTimeoutDelay=2000两个参数，在虚拟机等到线程进入安全点时间超过2000毫秒时认定超时，输出慢线程名称。

安全点选取是以是否具有让程序长期执行特征为标准选取，所以**方法调用、循环跳转、异常跳转**都可能设置安全点。但是虚拟机优化了一条，循环次数较少执行时间不会太长，所以**使用int类型或范围更小的数据类型作为索引值的循环默认不会被放置安全点**，这种循环称为**可数循环**。**使用long或范围更大的数据类型作为索引值的循环**称为**不可数循环**，会被放置安全点。通常是可行的，但是**如果循环单体执行很慢，那可数循环也会耗费较多时间**。

问题：清理大量链接的**索引值是int类型**，是可数循环，**不会插入安全点**，垃圾回收发生时，**必须等待这个位置的循环跑完才能进入安全点**，此时其他线程也必须等待，现象上看就是长时间停顿。

解决：**循环索引的数据类型从int改为long，此时不可数循环会放置安全点**。

1. **调优实战：**

第一步：**升级JDK版本**

出现问题时解决思路：

发生内存溢出后：打开VisualVM，监视内存页签，查看堆使用曲线和永久代使用曲线。查看问题，如本例永久代曲线和永久代容量几乎重合，然后查看永久代最大容量64M。

因为升级JDK版本了，所以需要对比两个版本对永久代的参数设置，发现JDK5运行时启动参数有个设置永久代最大容量256M，而后序没有了，所以会出现内存溢出。

第二步：**优化编译时间和类加载时间**

字节码验证阶段可能浪费时间，如果之前已经保证了字节码安全，可以禁用字节码验证，加快速度。

即时编译器：Java代码被调用了一定次数，会判定为热代码交给即时编译器编译为本地代码，提高运行速度。

第三步：**调整内存设置控制垃圾收集频率**

第四步：**选择收集器降低延迟**

**JVM内存调优:**

对JVM内存的系统级的调优主要的⽬的是**减少GC的频率和Full GC**的次数。过多的GC和

Full GC是会占⽤很多的系统资源（主要是CPU），影响系统的吞吐量。

使⽤JDK提供的内存查看⼯具，⽐如**JConsole和Java VisualVM**。 1）监控GC的状态，使⽤各种JVM⼯具，查看当前⽇志，并且**分析当前堆内存快照和gc⽇志**，根据实际的情况看是否需要优化。

2）通过**JMX的MBean或者Java的jmap**⽣成**当前的Heap信息**，并使⽤**Visual VM**或者Eclipse⾃带的Mat**分析dump⽂件**

3）如果参数设置合理，没有超时⽇志，GC频率GC耗时都不⾼则没有GC优化的必要，如果**GC时间超过1秒或者频繁GC，**则必须优化

4）**调整GC类型和内存分配**，使⽤1台和多台机器进⾏测试，进⾏性能的对⽐。再做修改，最后通过不断的试验和试错，分析并找到最合适的参数

**调优命令:**

Sun JDK监控和故障处理命令有jps jstat jmap jhat jstack jinfo

• **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是⽤于监视虚拟机运⾏时状态信息的命令，它可

以显示出虚拟机进程中的**类装载、内存、垃圾收集、JIT编译**等运⾏数据。

• **jmap**，JVM Memory Map命令⽤于**⽣成heap dump**⽂件

• **jhat**，JVM Heap Analysis Tool命令是与jmap搭配使⽤，⽤来**分析jmap⽣成的dump**，jhat内置了⼀个微型的HTTP/HTML服务器，⽣成dump的分析结果后，可以在浏览器中查看

• **jstack**，⽤于⽣成java虚拟机当前时刻的线程快照。 • jinfo，JVM Configuration info 这个命令作⽤是实时查看和调整虚拟机运⾏参数。

**调优⼯具**:

常⽤调优⼯具分为两类, jdk⾃带监控⼯具：**jconsole 和 jvisualvm**，第三⽅有：

MAT(Memory Analyzer Tool)、GChisto。 **•** jconsole，Java Monitoring and Management Console是从java5开始，在JDK中⾃带的java监控和管理控制台，⽤于**对JVM中内存，线程和类等的监控**