01 JVM性能优化

> JVM的性能优化可以分为代码层面和非代码层面。
> 在代码层面，大家可以结合字节码指令进行优化，比如一个循环语句，可以将循环不相关的代码提取到循环体之外，这样在字节码层面就不需要重复执行这些代码了。
> 在非代码层面，一般情况可以从参数、内存、GC以及cpu占用率等方面进行优化。
> 注意，JVM调优是一个漫长和复杂的过程，而在很多情况下，JVM是不需要优化的，因为JVM本身已经做了很多的内部优化操作，大家千万不要为了调优和调优。

1.1 代码优化

1.1.1 获取属性

```
class Person{
 public int age;
 ...
}
```

```
void print(Person p){
 int lastyear=p.age-1;
 int afteryear=p.age+1;
 System.out.println(afteryear-lastyear);
}
```

1.1.2 属性赋值

```
class Calc{
 int i=0;
 int add(){
 i=1;
 i=i+1;
 }
}
```

1.1.3 循环中的代码

```
class Calc{
 void calc(){
 for(int i=0;i<10;i++){
 int result=0;
 result=result+i;
 }
 }
}
```

1.2 参数优化

参数官网：https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html

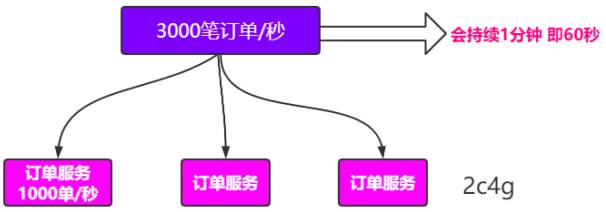
```
（1）-XX:MaxTenuringThreshold
 Sets the maximum tenuring threshold for use in adaptive GC sizing. The largest value is 15. The default value is 15 for the parallel (throughput) collector, and 6 for the CMS collector.

（2）-XX:PretenureSizeThreshold
 超过多大的对象直接在老年代分配，避免在新生代的Eden和S区不断复制

（3）-XX:+/- UseAdaptiveSizePolicy
 Enables the use of adaptive generation sizing. This option is enabled by default.

 （4）-XX:SurvivorRatio
 默认值为8

 （5）-XX:ConcGCThreads
 Sets the number of threads used for concurrent GC. The default value depends on the number of CPUs available to the JVM.

 （6）-Xsssize
 Sets the thread stack size (in bytes). Append the letter k or K to indicate KB

```

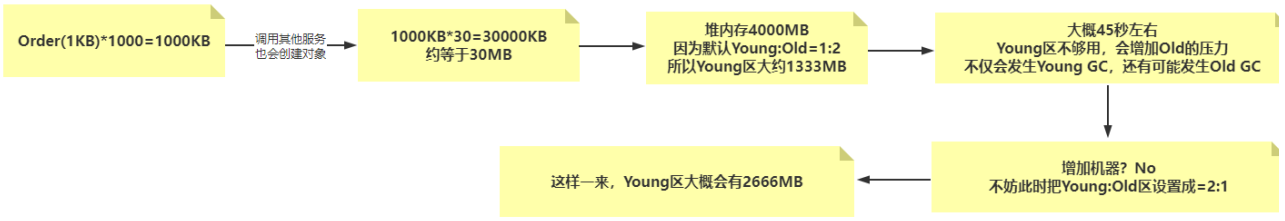
```
（7）-Xms和-Xmx
两者值一般设置成一样大，防止内存空间进行动态扩容

（8）-XX:ReservedCodeCacheSize
Sets the maximum code cache size (in bytes) for JIT-compiled code. Append the letter k or K to indicate kilobytes, m or M to indicate megabytes, g or G to indicate gigabytes.
```

1.3 内存调优

1.3.1 内存分配

> 每台机器配置2c4G，以每秒3000笔订单为例，整个过程持续60秒





1.3.2 内存泄露导致内存溢出

（1）代码

```
public class TLController {
 @RequestMapping(value = "/tl")
 public String tl(HttpServletRequest request) {
 ThreadLocal<Byte[]> tl = new ThreadLocal<Byte[]>();
 // 1MB
 tl.set(new Byte[1024*1024]);
 return "ok";
 }
}
```

（2）排查问题

```
01 启动
 java -jar -Xms1000M -Xmx1000M -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=jvm.hprof jvm-case-0.0.1-SNAPSHOT.jar

02 使用jmeter模拟并发

03 使用top命令查看
 top
 top -Hp PID

04 jstack查看有没有死锁或者IO阻塞
 jstack PID

05 查看堆内存使用情况
 jmap -heap PID
 java -jar arthas.jar ---> dashboard

06 获取到heap的文件，比如jvm.hprof，用相应的工具来分析，比如heaphero.io
```

1.4 GC调优

> 重点分析G1垃圾收集器

1.4.1 是否选用G1

官网：https://docs.oracle.com/javase/8/docs/technotes/guides/vm/G1.html#use_cases

```
（1）50%以上的堆被存活对象占用
（2）对象分配和晋升的速度变化非常大
（3）垃圾回收时间比较长
```

1.4.2 G1日志文件

> 参数：-XX:+UseG1GC -Xloggc:g1-gc.log

1.4.3 G1调优实战

（1）使用 G1 GC垃圾收集器，获取到日志文件：-Xms100M -Xmx100M

```
Throughput Min Pause Max Pause Avg Pause GC count
99.16% 0.00016s 0.0137s 0.00559s 12
```

（2）调整堆内存大小：-Xms300M -Xmx300M

（3）调整最大停顿时间：-XX:MaxGCPauseMillis=200 设置最大GC停顿时间指标

（4）启动并发GC时堆内存占用百分比：-XX:InitiatingHeapOccupancyPercent=45

> G1用它来触发并发GC周期,基于整个堆的使用率,而不只是某一代内存的使用比例。值为0则表示“一直执行GC循环”. 默认值为 45 (例如, 全部的 45% 或者使用了45%).

1.4.3 G1调优最佳实战

官网:

https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/g1_gc_tuning.html#recomm

（1）不要手动设置新生代和老年代的大小，只要设置整个堆的大小

```
G1收集器在运行过程中，会自己调整新生代和老年代的大小
其实是通过adapt代的大小来调整对象晋升的速度和年龄，从而达到为收集器设置的暂停时间目标，如果手动设置了大小就意味着放弃了G1的自动调优。
```

（2）不断调优暂停时间目标

```
一般情况下这个值设置到100ms或者200ms都是可以的(不同情况下会不一样)，但如果设置成50ms就不太合理。暂停时间设置的太短，就会导致出现G1跟不上垃圾产生的速度。最终退化成Full GC。
所以对这个参数的调优是一个持续的过程，逐步调整到最佳状态。暂停时间只是一个目标，并不能总是得到满足。
```

（3）使用-XX:ConcGCThreads=n来增加标记线程的数量

（4）适当增加堆内存大小

（5）不正常的Full GC

```
有时候会发现系统刚刚启动的时候，就会发生一次Full GC，但是老年代空间比较充足，一般是由Metaspace区域引起的。可以通过MetaspaceSize适当增加其大家，比如256M。
```

1.5 CPU占用率过高

```
（1）top
（2）top -Hp PID
查看进程中占用CPU高的线程id，即tid
（3）jstack PID | grep tid（1）top
```

1.6 大并发场景

> 在大并发场景下，除了对JVM本身进行调优之外，还需要考虑业务架构的整体调优

```
（1）浏览器缓存、本地缓存、验证码
（2）CDN静态资源服务器
（3）集群+负载均衡
（4）动静态资源分离、限流[基于令牌桶、漏桶算法]
（5）应用级别缓存、接口防刷限流、队列、Tomcat性能优化
（6）异步消息中间件
（7）Redis热点数据对象缓存
（8）分布式锁、数据库锁
（9）5分钟之内没有支付，取消订单、恢复库存等
```

1.7 JVM性能优化指南

02 JVM常见面试题

(1) 内存泄漏与内存溢出的区别

内存泄漏是指不再使用的对象无法得到及时的回收，持续占用内存空间，从而造成内存空间的浪费。

内存泄漏很容易导致内存溢出，但内存溢出不一定是内存泄漏导致的。

(2) Young GC会有stw吗?

不管什么GC，都会发送 stop-the-world，区别是发生的时间长短，主要取决于不同的垃圾收集器。

(3) Major gc和Full gc的区别

Major GC在很多参考资料中是等价于Full GC 的，我们也可以发现很多性能监测工具中只有Minor GC 和Full GC。一般情况下，一次 Full GC 将会对年轻代、老年代、元空间以及堆外内存进行垃圾回收。

触发Full GC的原因其实有很多：

当年轻代晋升到老年代的对象大小，并比目前老年代剩余的空间大小还要大时，会触发 Full GC；当老年代的空间使用率超过某阈值时，会触发 Full GC；当元空间不足时（JDK1.7永久代不足），也会触发 Full GC；当调用 System.gc() 也会安排一次 Full GC。

(4) 判断垃圾的方式

引用计数

可达性分析

(5) 为什么要区分新生代和老年代

当前虚拟机的垃圾收集都采用分代收集算法，这种算法没有什么新的思想，只是根据对象存活周期的不同将内存分为几块。

一般将 java 堆分为新生代和老年代，这样我们就可以根据各个年代的特点选择合适的垃圾收集算法。

比如在新生代中，每次收集都会有大量对象死去，所以可以选择复制算法，只需要付出少量对象的复制成本就可以完成每次垃圾收集。

而老年代的对象存活几率是比较高的，而且没有额外的空间对它进行分配担保，所以我们必须选择“标记-清除”或“标记-整理”算法进行垃圾收集。

(6) 方法区中的类信息会被回收吗?

方法区主要回收的是无用的类，那么如何判断一个类是无用的类的呢？判定一个常量是否是“废弃常量”比较简单，而要判定一个类是否是“无用的类”的条件则相对苛刻许多。类需要同时满足下面 3 个条件才能算是 “无用的类”

a-该类所有的实例都已经被回收，也就是 Java 堆中不存在该类的任何实例。

b-加载该类的 ClassLoader 已经被回收。

c-该类对应的 java.lang.Class 对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

(7) CMS与G1的区别

CMS 主要集中在老年代的回收，而 G1 集中在分代回收，包括了年轻代的 Young GC 以及老年代的 Mixed GC。

G1 使用了 Region 方式对堆内存进行了划分，且基于标记整理算法实现，整体减少了垃圾碎片的产生。

在初始化标记阶段，搜索可达对象使用到的 Card Table，其实现方式不一样。

G1可以设置一个期望的停顿时间。

(8) 为什么需要Survivor区

如果没有Survivor,Eden区每进行一次Minor GC,存活的对象就会被送到老年代。

这样一来，老年代很快被填满,触发Major GC(因为Major GC一般伴随着Minor GC,也可以看做触发了Full GC)。

老年代的内存空间远大于新生代,进行一次Full GC消耗的时间比Minor GC长得多。执行时间长有什么坏处?频发的Full GC消耗的时间很长,会影响大型程序的执行和响应速度。

(9) 为什么需要两个Survivor区

最大的好处就是解决了碎片化。也就是说为什么一个Survivor区不行?第一部分中,我们知道了必须设置Survivor区。

假设现在只有一个Survivor区,我们来模拟一下流程:

刚刚新建的对象在Eden中,一旦Eden满了,触发一次Minor GC,Eden中的存活对象就会被移动到Survivor区。

这样继续循环下去,下一次Eden满了的时候,问题来了,此时进行Minor GC,Eden和Survivor各有一些存活对象,

如果此时把Eden区的存活对象硬放到Survivor区,很明显这两部分对象所占有的内存是不连续的,也就导致了内存碎片化。

永远有一个Survivor是空的,另一个非空的Survivor无碎片。

(10) 为什么Eden:S1:S2=8:1:1

新生代中的对象大多数是“朝生夕死”的

(11) 堆内存中的对象都是所有线程共享的吗?

JVM默认为每个线程在Eden上开辟一个buffer区域，用来加速对象的分配，称之为TLAB，全称:Thread Local Allocation Buffer。

对象优先会在TLAB上分配，但是TLAB空间通常会比较小，如果对象比较大，那么还是在共享区域分配。

(12) Java虚拟机栈的深度越大越好吗?

线程栈的大小是个双刃剑，如果设置过小，可能会出现栈溢出，特别是在该线程内有递归、大的循环时出现溢出的可能性更大，如果该值设置过大，就有影响到创建栈的数量，如果是多线程的应用，就会出现内存溢出的错误。

(13) 垃圾收集器一般如何选择

a-用默认的

b-关注吞吐量：使用Parallel

c-关注停顿时间：使用CMS、G1

d-内存超过8GB：使用G1

e-内存很大或希望停顿时间几毫秒级别：使用ZGC

(14) 什么是方法内联?

正常调用方法时，会不断地往Java虚拟机栈中存放新的栈帧，这样开销比较大，其实jvm虚拟机内部为了节省这样开销，可以把一些方法放到同一个栈帧中执行。

(15) 什么样的情况下对象会进入Old区?

a-大对象

b-到了GC年龄阈值

c-担保机制

d-动态对象年龄判断

(16) 聊聊Minor GC、Major GC、Full GC发生的时机

Minor GC：Eden或S区空间不足或发生了Major GC

Major GC：Old区空间不足

Full GC：Old空间不足，元空间不足，手动调用System.gc())