## 第5章性能调优

### 5.2调优

在寻找到系统的性能瓶颈之后，接下来的步骤就是调优，以提高系统的性能。调优通常可以从硬件、操作系统、JVM以及程序四个方面来着手。下面就从JVM及程序方面来介绍一些常用的调优方法。

### 5.2.1 JVM调优

JVM调优主要是从内存管理方面的调优，包括各个代的大小、GC策略等。由于GC动作会挂起应用线程，严重影响性能。根据应用的情况选择不同的内存管理策略，这些策略都是为了尽量降低GC所导致的应用暂停时间。

**代大小的调优**

在不采用G1（G1不区分minor GC和Full GC）的情况下，**通常minor GC会远快于Full FC，各个代的大小设置决定了minor GC和Full GC触发的时机，**在代大小的调优上，最为关键的参数：-Xms -Xmx -Xmn -XX:SurvivorRatio -XX:MaxTenuringThreshold。

-Xms和-Xmx通常设置为相同的值，避免运行时要不断地扩展JVM内存空间，这个值决定了JVM Heap所能用的空间大小。在内存不够用的情况下可以适当加大此值。这个值能调整到多大取决于操作系统位数以及CPU能力。

-Xmn决定了新生代的大小。适用于调整新生代的大小，新生代的大小决定了多少比例的对象有机会在minor GC阶段被回收。但此值相应的也决定了老年代的大小。新生代越大，通常意味着多数对象能在minor GC阶段被回收掉。但同时意味着老年代的空间会变小，可能会造成更加频繁的Full GC，甚至是OutOfMemoryError。

-XX:SurvivorRatio决定了新生代（New Generation）空间的大小，新生代中Eden、S0和S1三个区域的比率可以通过这个参数进行控制。Eden区越大通常意味着minor GC发生的频率越低。但有可能会造成Survivor区太小，导致对象经过minor GC后直接进入老年代了，从而更加频繁触发Full GC，这取决于当Eden区满的时候其中对象存活的比例。

-XX:MaxTenuringThreshold控制对象在经历多少次Minor GC后才转入老年代，此参数只有在串行GC时有效，其他方式GC时间由Sun JDK自行决定。

**代大小的调优的调优策略：**

**1.避免新生代大小设置过小。**当新生代大小设置过小时，会产生两种比较明显的现象，一是minor GC的次数更加频繁；二是有可能导致minor GC对象直接进入老年代。

当minor GC过于频繁，或发现经常出现minor GC时，Survivor的一个区域空间满。且老年代增长超过了Survivor区域大小时，就要考虑新生代大小的调整了。调整时的原则是在不能调大JVM Heap的情况下，尽可能放大新生代空间，尽量让对象在minor GC阶段被回收。

**2.避免新生代大小设置过大。**新生代设置过大会带来两个典型的现象，一是老年代变小了，有可能导致Full GC频繁执行；二是minor GC的耗时大幅度增加。新生代通常不能设置太大，大多数情况下设置应当比老年代小，通常推荐的比例是新生代占JVM Heap区大小的33%左右。

**3.避免Survivor区过小或过大。**在采用串行GC时，默认情况下Eden、S0、S1的大小比例是8：1：1。在无法调整JVM Heap以及新生代的大小时，合理调整Survivor区的大小也能带来一些效果。调大SurvivorRatio值意味Eden区域变大，minor GC的触发次数会降低，但此时Survivor区域的空间变小了，如有超过了Survivor空间大小的对象在minor GC后仍没有被回收，则会直接进入老年代；调小SurvivorRatio则意味Eden区域变小，minor GC的触发次数会增加，Survivor区域变大，意味着可以存储更多在minor GC后仍然存活的对象，避免其进入老年代。

**4.合理设置新生代存活周期。**-XX:MaxTenuringThreshold控制对象在经历多少次Minor GC后才转入老年代，其默认为15次。增大了存活周期后，对象在Minor GC阶段被回收的机会就增加了，但同时带来的是survivor区被占用

**GC策略的调优**

在实际情况中，串行GC性能太差，因此实际场景中使用到的主要行为并发和并行GC。

大部分Web应用在处理请求时设置了一个最大可同时处理的请求数，当超过此请求数时，会将之后的请求放入等待队列之中，而这个等待的队列也限制了大小。当等待的队列满了之后仍然有请求进入，那么这些请求会直接被丢弃。此外，所有的请求又都是有超时限制的。在这种情况下如果触发了造成应用暂停的Full GC，那么有可能在这次Full GC后，应用中很多请求就超时或被丢弃了。例如请求的超时时间为3秒，在排队时应用中触发了一次Full GC，造成了3秒的暂停，那么之前在此应用上等待处理请求就全部超时了。从上面看，Web应用需要一个对应用造成暂停时间短的GC，再加上大部分Web应用的瓶颈都不在CPU 上。因此对于Web应用而言，并发和并行GC是不错的选择。

**JVM调优案例**

书中案例总结：

1.缩小Eden Space，扩大Survivor Space，这样进入老年代的对象就变少了，从而减少了Full GC的几率。更好的方法就是调大整个JVM Heap区域，并同比例增大新生代的空间。在增大新生代空间后，要注意观察Minor GC消耗的时间。

2.在进行参数调整时，可以根据目前收集到的顶峰时系统请求次数、响应时间以及GC的信息，来估计系统每次请求需要消耗的内存，以及每次Minor GC时存活的对象所占的内存，从而估计需要设置多大的Survivor才能够避免对象进如老年代。由于参数估计是以请求次数和响应时间为基准的，因此一旦系统的响应速度下降或请求的次数上升，就可能仍然会导致大量对象进入老年代。从这个层面上讲，根本要做的调优是提升请求速度以及降低每次请求需要分配的内存，只有这样才能使得应用能够支撑更高的并发量。

### 5.2.2程序调优

**CPU消耗严重的解决方法**

1.CPU us高的解决方法

根据之前的分析，CPU us高的原因主要执行线程时无任何挂起动作，且一直执行，导致CPU没有任何机会去调度执行其他的线程，造成线程饿死的现象，对于这种情况，常见的一种优化方法就是对这种线程的动作增加Thread.sleep，以释放CPU的执行权，降低CPU的消耗。

在实际的Java应用中会有很多的类似的场景，例如多线程的任务执行管理器，它通常要通过扫描任务集合列表来执行任务，对于这些类似的场景，都可以通过增加一定的sleep时间来避免消耗过多的CPU。

除了上述场景外，还要一种经典的场景是状态的扫描，例如某线程要等待其他线程改变了值之后才能继续执行。对于这种场景，最佳的方式就是采用wait/notify机制。

对于其他类似循环次数太多、正则、计算等造成的CPU us过高的状况，则需要结合业务需求来进行调优。

对于GC频繁造成的CPU us高的现象，则需要通过JVM调优或者程序调优，降低GC的执行次数。

2.CPU sy高的解决方法

CPU sy高的原因主要是线程的运行状态要经常切换，对于这种情况，最简单的优化方法就是减少线程数。

造成CPU sy高的原因除了启动的线程过多以外，还要一个重要的原因就是线程之间锁竞争激烈，造成的线程状态要经常切换，因此尽可能降低线程间锁的竞争也是常见的优化方法。锁竞争降低后，线程的状态切换的次数也就会下降。

除了上述两种情况以外，对于分布式Java应用而言，还有一种典型的现象是应用中有较多的网络IO操作确实是需要一些锁竞争机制，但是为了能够支撑高的并发量，在Java应用中又只能借助启动更多的线程来支撑。在这样的一种情况下当并发量增长到一定程度后，可能会造成CPU sy高的现象，对于这样的现象可以采用协程来支撑更高的并发量。

在目前的Sun JDK中，创建启动一个Thread对象就意味着运行了一个原生线程，当这个线程中有任何的阻塞动作（例如同步文件IO、同步网络IO、锁等待、Thread sleep等）时，这个线程就会被高高挂起，但仍然会占据线程的资源。当线程中阻塞动作完成时，由操作系统来恢复线程的上下文。当并发量上涨后，有可能出现的现象是启动的大量线程处于浪费状态。

采用协程后，能做到当线程等待数据库执行结果时，就立刻释放此线程资源给其他请求，等到数据库执行结果返回后才继续执行，在Java中目前主要可用于实现协程的框架为kilim。

**文件IO消耗严重的解决方法**

从程序角度而言，造成文件IO消耗严重的原因主要是多个线程在写大量的数据到同一文件，导致文件变得很大，从而写入速度越来越慢，并造成各线程激烈争抢文件锁，对于这类情况，常用有以下几种。

* 异步写文件

将写文件的同步动作改为异步动作，避免应用由于写文件慢而导致性能下降太多，例如写日志

* 批量读写

频繁的读写操作会IO消耗会严重，批量操作将会大幅度提升IO操作的性能

限流

频繁读写的另外一种调优方式就是限流，从而将文件IO消耗控制到一个可以接受的范围。

* 限制文件大小

操作太大的文件也是造成文件IO效率低一个原因，因此对于每个输出的文件，都应做大小的限制，在超出最大值后生成一个新的文件。

除了以上这些外，还有就是尽可能采用缓冲区等方式读取文件内容，避免不断与操作系统交互。

**网络IO消耗很重的解决方法**

从程序角度来看，造成网络IO消耗很重的原因是同时需要发送或接收的包太多，对于这类情况，常用的调优方法为进行限流，限流通常是限制发送packet的频率，从而在网络IO消耗可以接受的范围内来发送packet。

**对于内存消耗严重的情况**

在内存消耗方面，最明显的在于消耗了过多的JVM Heap内存，造成GC频繁执行的现象。除了JVM的调优以外，在寻找内存消耗严重的代码后，可从代码本身进行优化，避免资源内存消耗过多。

1. 释放不必要的引用

内存消耗严重的情况中最典型的一种情况是代码中有了不需要的对象引用，造成了这些对象无法被GC，从而占据了JVM堆内存。这种情况最典型的一个例子是在复用线程的情况下使用ThreadLocal，由于线程复用，ThreadLocal中存放的对象如未主动释放的话则不会被GC。

1. 使用对象缓存池

创建对象缓存的实例要耗费一些CPU以及内存，使用对象缓存池一定程度上可以降低JVM Heap内存的使用。在内存消耗严重的情况下，采用对象缓存池可大幅度提升性能，避免创建对象所耗费的时间以及频繁的GC。

1. 采用合理的缓存失效算法

上面说到采用对象缓存池来降低内存的消耗，但如果放入太多的对象在缓存池中，反而会造成内存消耗严重。同时由于缓存池一直对这些对象持有引用，从而造成Full GC增多，对于这种情况要合理地控制缓存池的大小。

控制缓存池的大小问题在于当达到缓存池的最大容量后，如果要加入新的对象该如何处理，有一些经典的缓存失效算法来清除缓存池中的对象。例如FIFO、LRU、LFU等，这些算法可以控制缓存池中对象的数日，避免缓存池中的对象数量无限上涨。

1. 合理使用SoftReference和WeakReferenc

对于占据内存但又不是必须存在的对象，例如缓存对象，也可以基于SoftReference和WeakReferenc的方式来进行缓存。SoftReference的对象会在内存不够用的时候进行回收。WeakReferenc的对象会在Full GC的时候进行回收。

### 5.2.3对于资源消耗不多，但程序执行慢的情况

造成这种情况的主要原因通常有锁竞争激烈及未充分发挥硬件资源两种。

**锁竞争激烈**

锁是影响性能的重要因素，但为了保证资源的一致性，多线程应用中锁的使用是必不可免的，只能尽量的去降低线程间的锁竞争，常见的方法如下：

1. 使用并发包中的类

并发包中很多的类都采用了lock-free、nonblocking算法，减少了多线程情况下资源的锁竞争。因此对于多线程要共享操作的资源而言，应尽量使用并发包中的类来实现。

1. 使用Treiber算法
2. 使用Michael-Scott非阻塞算法
3. 尽可能少用锁

尽可能让锁出现在需要他的地方，通常没必要对整个方法加锁，而只对需要控制的资源做加锁的操作。因此在编写多线程程序时，要仔细考虑

1. 拆分锁
2. 去除读写操作的互斥锁

**未充分发挥硬件资源**