**Java性能优化原则：**代码运算性能、内存回收、应用配置（影响Java程序主要原因是垃圾回收，下面会重点介绍这方面）

**代码层优化：**避免过多循环嵌套、调用和复杂逻辑。

**Tomcat调优主要内容如下：**

**1、增加最大连接数**

**2、调整工作模式**

**3、启用gzip压缩**

**4、调整JVM内存大小**

**5、作为Web服务器时，与Apache整合或Nginx**

**6、合理选择垃圾回收算法**

**7、尽量使用较新JDK版本**

**生产配置实例：**

<Connectorport="8080"protocol="org.apache.coyote.http11.Http11NioProtocol"  
               maxThreads="1000"  
               minSpareThreads="100"  
               maxSpareThreads="200"  
               acceptCount="900"  
               disableUploadTimeout="true"  
              connectionTimeout="20000"  
               URIEncoding="UTF-8"  
               enableLookups="false"  
               redirectPort="8443"  
               compression="on"  
              compressionMinSize="1024"  
              compressableMimeType="text/html,text/xml,text/css,text/javascript"/>

**参数说明：**

org.apache.coyote.http11.Http11NioProtocol：调整工作模式为Nio

maxThreads：最大线程数，默认150。增大值避免队列请求过多，导致响应缓慢。

minSpareThreads：最小空闲线程数。

maxSpareThreads：最大空闲线程数，如果超过这个值，会关闭无用的线程。

acceptCount：当处理请求超过此值时，将后来请求放到队列中等待。

disableUploadTimeout：禁用上传超时时间

connectionTimeout：连接超时，单位毫秒，0代表不限制

URIEncoding：URI地址编码使用UTF-8

enableLookups：关闭dns解析，提高响应时间

compression：启用压缩功能

compressionMinSize：最小压缩大小，单位Byte

compressableMimeType：压缩的文件类型

**Tomcat有三种工作模式：Bio、Nio和Apr，下面简单了解下他们工作原理：**

Bio(Blocking I/O)：默认工作模式，阻塞式I/O操作，没有任何优化技术处理，性能比较低。

Nio(New I/O or Non-Blocking)：非阻塞式I/O操作，有Bio有更好的并发处理性能。

Apr(Apache Portable Runtime，Apache可移植运行库)：首选工作模式，主要为上层的应用程序提供一个可以跨越多操作系统平台使用的底层支持接口库。

tomcat利用基于Apr库tomcat-native来实现操作系统级别控制，提供一种优化技术和非阻塞式I/O操作，大大提高并发处理能力。但是需要安装apr和tomcat-native库。

**工作模式原理涉及到了网络I/O模型知识：**

**阻塞式I/O模型：**应用进程调用recv函数系统调用时，如果等待要操作的数据没有发送到内核缓冲区，应用进程将阻塞，不能接收其他请求。反之，内核recv端缓冲区有数据，内核会把数据复制到用户空间解除阻塞，继续处理下一个请求。（内核空间(缓冲区)--用户空间(系统调用)）

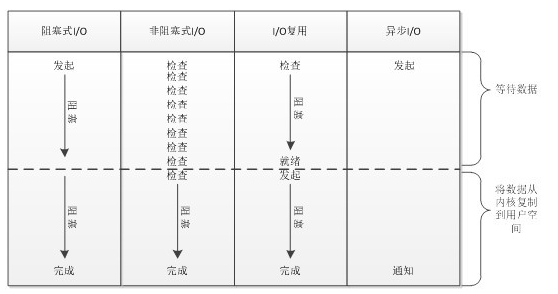
**非阻塞式I/O模型：**应用进程设置成非阻塞模式，如果要操作的数据没有发送到内核缓冲区，recv系统调用返回一个错误，应用进程利用轮询方式不断检查此操作是否就绪，如果缓冲区中有数据则返回，I/O操作同时不会阻塞应用进程，期间会继续处理新请求。

**I/O复用模型：**阻塞发生在select/poll的系统调用上，而不是阻塞在实际的I/O系统调用上。能同时处理多个操作，并检查操作是否就绪，select/epoll函数发现有数据就绪后，就通过实际的I/O操作将数据复制到应用进程的缓冲区中。

**异步I/O模型：**应用进程通知内核开始一个异步I/O操作，并让内核在整个操作（包括数据复制缓冲区）完成后通知应用进程，期间会继续处理新请求。

**I/O操作分为两个阶段：**第一个阶段等待数据可用，第二个阶段将数据从内核复制到用户空间。

**前三种模型的区别：**第一阶段阻塞式I/O阻塞在I/O操作上，非阻塞式I/O轮询，I/O复用阻塞在select/poll或epoll上。第二阶段都是一样的。而异步I/O的两个阶段都不会阻塞进程。



**Java性能问题主要来自于JVM，JVM GC也比较复杂，再调优之前了解下相关基础概念是必要的：**

1）JVM内存划分分为年轻代（Young Generation）、老年代（Old Generation）、永久代（Permanent Generation）。

2）年轻代又分为Eden和Survivor区。Survivor区由FromSpace和ToSpace组成。Eden区占大容量，Survivor两个区占小容量，默认比例大概是8:2。

3）堆内存（Heap）=年轻代+老年代。非堆内存=永久代。

4）堆内存用途：存放的是对象，垃圾收集器就是收集这些对象，然后根据GC算法回收。

5）非堆内存用途：JVM本身使用，存放一些类、方法、常量、属性等。

6）年轻代：新生成的对象首先放到年轻代的Eden区中，当Eden满时，经过GC后，还存活的对象被复制到Survivor区的FromSpace中，如果Survivor区满时，会再被复制到Survivor区的ToSpace区。如果还有存活对象，会再被复制到老年代。

7）老年代：在年轻代中经过GC后还存活的对象会被复制到老年代中。当老年代空间不足时，JVM会对老年代进行完全的垃圾回收（Full GC）。如果GC后，还是无法存放从Survivor区复制过来的对象，就会出现OOM（Out of Memory）。

8）永久代：也称为方法区，存放静态类型数据，比如类、方法、属性等。

**垃圾回收（GC，Garbage Collection）算法：**

1）标记-清除（Mark-Sweep）

GC分为两个阶段，标记和清除。首先标记所有可回收的对象，在标记完成后统一回收所有被标记的对象。同时会产生不连续的内存碎片。碎片过多会导致以后程序运行时需要分配较大对象时，无法找到足够的连续内存，而不得已再次触发GC。

2）复制（Copy）

将内存按容量划分为两块，每次只使用其中一块。当这一块内存用完了，就将存活的对象复制到另一块上，然后再把已使用的内存空间一次清理掉。这样使得每次都是对半个内存区回收，也不用考虑内存碎片问题，简单高效。缺点需要两倍的内存空间。

3）标记-整理（Mark-Compact）

也分为两个阶段，首先标记可回收的对象，再将存活的对象都向一端移动，然后清理掉边界以外的内存。此方法避免标记-清除算法的碎片问题，同时也避免了复制算法的空间问题。

一般年轻代中执行GC后，会有少量的对象存活，就会选用复制算法，只要付出少量的存活对象复制成本就可以完成收集。而老年代中因为对象存活率高，没有额外过多内存空间分配，就需要使用标记-清理或者标记-整理算法来进行回收。

**垃圾收集器：**

1）串行收集器（Serial）

比较老的收集器，单线程。收集时，必须暂停应用的工作线程，直到收集结束。

2）并行收集器（Parallel）

多条垃圾收集线程并行工作，在多核CPU下效率更高，应用线程仍然处于等待状态。

3）CMS收集器（Concurrent Mark Sweep）

CMS收集器是缩短暂停应用时间为目标而设计的，是基于标记-清除算法实现，整个过程分为4个步骤，包括：

* 初始标记（Initial Mark）
* 并发标记（Concurrent Mark）
* 重新标记（Remark）
* 并发清除（Concurrent Sweep）

其中，初始标记、重新标记这两个步骤仍然需要暂停应用线程。初始标记只是标记一下GC Roots能直接关联到的对象，速度很快，并发标记阶段是标记可回收对象，而重新标记阶段则是为了修正并发标记期间因用户程序继续运作导致标记产生变动的那一部分对象的标记记录，这个阶段暂停时间比初始标记阶段稍长一点，但远比并发标记时间段。

由于整个过程中消耗最长的并发标记和并发清除过程收集器线程都可以与用户线程一起工作，所以，CMS收集器内存回收与用户一起并发执行的，大大减少了暂停时间。

4）G1收集器（Garbage First）

G1收集器将堆内存划分多个大小相等的独立区域（Region），并且能预测暂停时间，能预测原因它能避免对整个堆进行全区收集。G1跟踪各个Region里的垃圾堆积价值大小（所获得空间大小以及回收所需时间），在后台维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的Region，从而保证了再有限时间内获得更高的收集效率。

G1收集器工作工程分为4个步骤，包括：

* 初始标记（Initial Mark）
* 并发标记（Concurrent Mark）
* 最终标记（Final Mark）
* 筛选回收（Live Data Counting and Evacuation）

初始标记与CMS一样，标记一下GC Roots能直接关联到的对象。并发标记从GC Root开始标记存活对象，这个阶段耗时比较长，但也可以与应用线程并发执行。而最终标记也是为了修正在并发标记期间因用户程序继续运作而导致标记产生变化的那一部分标记记录。最后在筛选回收阶段对各个Region回收价值和成本进行排序，根据用户所期望的GC暂停时间来执行回收。

**了解了JVM基础知识，下面配置下相关Java参数，将下面一段放到catalina.sh里面：**

|  |
| --- |
| JAVA\_OPTS="-server -Xms1024m -Xmx1536m -XX:PermSize=256m -XX:MaxPermSize=512m -XX:+UseConcMarkSweepGC -XX:+UseParallelGCThreads=8 XX:CMSInitiatingOccupancyFraction=80 -XX:+UseCMSCompactAtFullCollection -XX:CMSFullGCsBeforeCompaction=0 -XX:-PrintGC -XX:-PrintGCDetails -XX:-PrintGCTimeStamps -Xloggc:../logs/gc.log" |

|  |  |
| --- | --- |
| **参数** | **描述** |
| -Xms | 堆内存初始大小，单位m、g |
| -Xmx | 堆内存最大允许大小，一般不要大于物理内存的80% |
| -XX:PermSize | 非堆内存初始大小，一般应用设置初始化200m，最大1024m就够了 |
| -XX:MaxPermSize | 非堆内存最大允许大小 |
| -XX:+UseParallelGCThreads=8 | 并行收集器线程数，同时有多少个线程进行垃圾回收，一般与CPU数量相等 |
| -XX:+UseParallelOldGC | 指定老年代为并行收集 |
| -XX:+UseConcMarkSweepGC | CMS收集器（并发收集器） |
| -XX:+UseCMSCompactAtFullCollection | 开启内存空间压缩和整理，防止过多内存碎片 |
| -XX:CMSFullGCsBeforeCompaction=0 | 表示多少次Full GC后开始压缩和整理，0表示每次Full GC后立即执行压缩和整理 |
| -XX:CMSInitiatingOccupancyFraction=80% | 表示老年代内存空间使用80%时开始执行CMS收集，防止过多的Full GC |

**注意：**不是JVM内存设置越大越好，具体还是根据项目对象实际占用内存大小而定，可以通过Java自带的分析工具来查看。如果设置过大，会增加回收时间，从而增加暂停应用时间。

博客地址：http://lizhenliang.blog.51cto.com

QQ群：323779636（Shell/Python运维开发群）

**gzip压缩作用：**节省服务器流量和提高网站访问速度。客户端请求服务器资源后，服务器将资源文件压缩，再返回给客户端，由客户端的浏览器负责解压缩并浏览。

使用Apache与Tomcat整合，因为Tomcat处理静态文件能力远不足Apache，因此让Apache来处理静态文件，Tomcat处理动态jsp文件，可以有效提高处理速度。

在集群架构下，会涉及到一个问题，怎么保存Session？

**TomcatSessionID持久化三种方法：**

   Session粘性：通过浏览器Cookie绑定SessionID，通过sticky模式将同一Session请求分配到同一Tomcat上。

   Session复制：Tomcat通过广播形式将Session同步到其他Tomcat节点，并且Linux下要手动开启开放广播地址。不易后端节点过多

 Session保存数据库（memcache、redis）：将SessionID保存在共享的数据库中。

**OOM（Out of Memory）异常常见有以下几个原因：**

1）老年代内存不足：java.lang.OutOfMemoryError:Javaheapspace

2）永久代内存不足：java.lang.OutOfMemoryError:PermGenspace

3）代码bug，占用内存无法及时回收。

前两种情况通过加大内存容量，可以得到解决。如果是代码bug，就要通过jstack、jmap、jstat自带的工具分析问题，定位到相关代码，让开发解决。