Linux 系统信息命令

1. **系统信息查询**
   1. **系统参数**
2. 查看linux操作系统

cat /etc/issue

1. 查看内核版本

uname –a

1. 查看cpu

cat /proc/cpuinfo

1. 查看内存

cat /proc/meminfo

* 1. **配置IP地址**

方法一：修改配置文件，永久生效

1、 vi /etc/sysconfig/network-scripts/ifcfg-eth0

|  |
| --- |
| DEVICE=eth0 HWADDR=00:0c:29:cc:fe:bd  BOOTPROTO=static IPV6INIT=yes  BROADCAST=192.168.10.255  IPADDR=192.168.10.244  NETMASK=255.255.255.0  GATEWAY=192.168.10.1  ONBOOT=yes |

2、vi /etc/resolv.conf

|  |
| --- |
| nameserver 192.168.10.3  nameserver 192.168.10.4 |

3、sudo /etc/init.d/network restart 重新启动网卡

方法二、ifconfig临时配置

Ifconfig eth0 192.168.10.10 netmask 255.255.255.0

方法三、图形界面配置netconfig

方法四、同一网卡上配置多个IP

在/etc/sysconfig/network-scripts 目录下创建文件ifcfg-eth0:1

|  |
| --- |
| DEVICE=eth0:1 BROADCAST=192.168.10.255  IPADDR=192.168.10.244  NETMASK=255.255.255.0  ONBOOT=yes |

然后重启网卡

* 1. **防火墙**

关闭防火墙 chkconfig iptables off

or

/etc/init.d/iptables stop

开启防火墙 chkconfig iptables on

or

# /etc/init.d/iptables start

* 1. **修改主机名**

第一步：

#hostname oratest

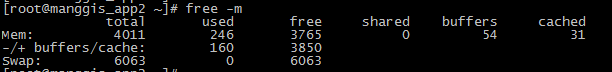
第二步：

修改/etc/sysconfig/network中的hostname

第三步：

修改/etc/hosts,增加ip hostname

* 1. **系统资源监控**
     1. **free命令**



Total:物理内存总量

Used:已使用物理内存大小。

Free:剩余可用物理内存大小。

Shared:多个进程共享的内存总额。

Buffers:有待写入磁盘的缓存数据。

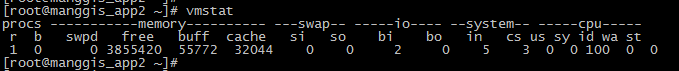
Cached:已经从磁盘读取，有待后续使用的缓存数据。如果cached的值很大，说明缓存的文件数很多，如果频繁访问到的文件都能cached，那么磁盘的读I/O会

计算说明：

第3行used: 160 = 246 -54 - 31

第3行free: 3850 = 3765 + 54 +31

* + 1. **Vmstat命令**



**(1)进程procs：**   
r：在运行队列中等待的进程数 。对一个CPU而言，运行队列不要超过3，如双核CPU不要超过6。  
b：在等待io的进程数 。  
(2**)内存memoy：**  
swpd：现时可用的交换内存（单位KB）。   
free：空闲的内存（单位KB）。  
buff: 缓冲去中的内存数（单位：KB）。  
cache：被用来做为高速缓存的内存数（单位：KB）。  
**(3) swap交换页面**  
si: 从磁盘交换到内存的交换页数量，单位：KB/秒。  
so: 从内存交换到磁盘的交换页数量，单位：KB/秒。

**(4) io块设备:**  
bi: 发送到块设备的块数  
bo: 从块设备接收到的快熟。  
**(5) system系统：**  
in: 每秒的中断数，包括时钟中断。  
cs: 每秒的环境（上下文）转换次数。

上面这2个值越大，会看到由内核消耗的CPU时间会越多

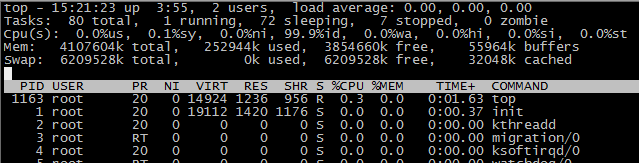
**(6) cpu：**  
cs：表示上下文切换数量。例如我们调用系统函数，就要进行上下文切换，线程的切换，也要 进程上下文切换，这个值要越小越好，太大了，要考虑调低线程或者进程的数目,例如在apache和nginx这种web服务器中，我们一般做性能测试时会进行几千并发甚至几万并发的测试，选择web服务器的进程可以由进程或者线程的峰值一直下调，压测，直到cs到一个比较小的值，这个进程和线程数就是比较合适的值了。系统调用也是，每次调用系统函数，我们的代码就会进入内核空间，导致上下文切换，这个是很耗资源，也要尽量避免频繁调用系统函数。上下文切换次数过多表示你的CPU大部分浪费在上下文切换，导致CPU干正经事的时间少了，CPU没有充分利用，是不可取的。

us：用户进程使用的时间 。以百分比表示。  
sy：系统进程使用的时间。 以百分比表示。  
id：中央处理器的空闲时间 。以百分比表示。

wa:表示由于IO等待而使CPU处于idle状态的时间。

st：来自于一个虚拟机偷取的CPU时间的百分比。

* + 1. **Top**



1. 第一行是任务队列信息

|  |  |
| --- | --- |
| load average: 0.06, 0.60, 0.48 | 系统负载，即任务队列的平均长度。 三个数值分别为  1分钟、5分钟、15分钟前到现在的平均值。 |

2. 第二、三行为进程和CPU的信息

当有多个CPU时，这些内容可能会超过两行。内容如下：

|  |  |
| --- | --- |
| Tasks | 进程总数 |
| running | 正在运行的进程数 |
| sleeping | 睡眠的进程数 |
| stopped | 停止的进程数 |
| zombie | 僵尸进程数 |
| Cpu(s): us | 用户空间占用CPU百分比 |
| sy | 内核空间占用CPU百分比 |
| ni | 用户进程空间内改变过优先级的进程占用CPU百分比 |
| id | 空闲CPU百分比 |
| wa | 等待输入输出的CPU时间百分比 |
| hi | 硬件中断 |
| si | 软件中断 |

3. 第四五行为内存信息。

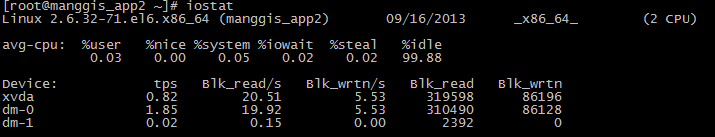
内容如下：

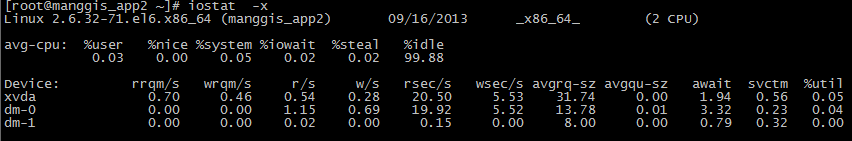
|  |  |
| --- | --- |
| Mem: total | 物理内存总量 |
| used | 使用的物理内存总量 |
| free | 空闲内存总量 |
| buffers | 用作内核缓存的内存量 |
| Swap | 交换区总量 |
| used | 使用的交换区总量 |
| free | 空闲交换区总量 |
| cached | 缓冲的交换区总量。 内存中的内容被换出到交换区，而后又被换入到内存，但使用过的交换区尚未被覆盖， 该数值即为这些**内容已存在于内存中**的交换区的大小。相应的内存再次被换出时可不必再对交换区写入。 |

二.  进程信息

|  |  |
| --- | --- |
| 列名 | 含义 |
| PID | 进程id |
| PPID | 父进程id |
| RUSER | Real user name |
| UID | 进程所有者的用户id |
| USER | 进程所有者的用户名 |
| GROUP | 进程所有者的组名 |
| TTY | 启动进程的终端名。不是从终端启动的进程则显示为 ? |
| PR | 优先级 |
| NI | nice值。负值表示高优先级，正值表示低优先级 |
| P | 最后使用的CPU，仅在多CPU环境下有意义 |
| %CPU | 上次更新到现在的CPU时间占用百分比 |
| TIME | 进程使用的CPU时间总计，单位秒 |
| TIME+ | 进程使用的CPU时间总计，单位1/100秒 |
| %MEM | 进程使用的物理内存百分比 |
| VIRT | 进程使用的虚拟内存总量，单位kb。VIRT=SWAP+RES |
| SWAP | 进程使用的虚拟内存中，被换出的大小，单位kb。 |
| RES | 进程使用的、未被换出的物理内存大小，单位kb。RES=CODE+DATA |
| CODE | 可执行代码占用的物理内存大小，单位kb |
| DATA | 可执行代码以外的部分(数据段+栈)占用的物理内存大小，单位kb |
| SHR | 共享内存大小，单位kb |
| nFLT | 页面错误次数 |
| nDRT | 最后一次写入到现在，被修改过的页面数。 |
| S | 进程状态。             D=不可中断的睡眠状态             R=运行             S=睡眠             T=跟踪/停止             Z=僵尸进程 |
| COMMAND | 命令名/命令行 |
| WCHAN | 若该进程在睡眠，则显示睡眠中的系统函数名 |
| Flags | 任务标志，参考 sched.h |

* + 1. **iostat**





**cpu属性值说明：**

%user：CPU处在用户模式下的时间百分比。

%nice：CPU处在带NICE值的用户模式下的时间百分比。

%system：CPU处在系统模式下的时间百分比。

%iowait：CPU等待输入输出完成时间的百分比。

%steal：管理程序维护另一个虚拟处理器时，虚拟CPU的无意识等待时间百分比。

%idle：CPU空闲时间百分比。

**备注：**如果%iowait的值过高，表示硬盘存在I/O瓶颈，%idle值高，表示CPU较空闲，如果%idle值高但系统响应慢时，有可能是CPU等待分配内存，此时应加大内存容量。%idle值如果持续低于10，那么系统的CPU处理能力相对较低，表明系统中最需要解决的资源是CPU。

**disk属性值说明：**

rrqm/s:  每秒进行 merge 的读操作数目。即 rmerge/s

wrqm/s:  每秒进行 merge 的写操作数目。即 wmerge/s

r/s:  每秒完成的读 I/O 设备次数。即 rio/s

w/s:  每秒完成的写 I/O 设备次数。即 wio/s

rsec/s:  每秒读扇区数。即 rsect/s

wsec/s:  每秒写扇区数。即 wsect/s

rkB/s:  每秒读K字节数。是 rsect/s 的一半，因为每扇区大小为512字节。

wkB/s:  每秒写K字节数。是 wsect/s 的一半。

avgrq-sz:  平均每次设备I/O操作的数据大小 (扇区)。

avgqu-sz:  平均I/O队列长度。

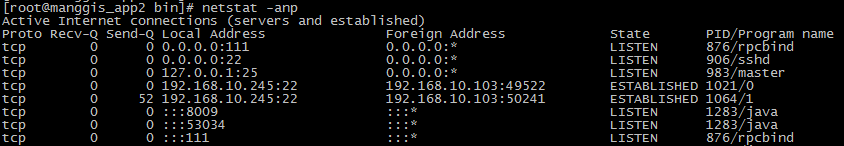
await:  平均每次设备I/O操作的等待时间 (毫秒)。

svctm: 平均每次设备I/O操作的服务时间 (毫秒)。

%util:  一秒中有百分之多少的时间用于 I/O 操作，即被io消耗的cpu百分比

**备注：**如果 %util 接近 100%，说明产生的I/O请求太多，I/O系统已经满负荷，该磁盘可能存在瓶颈。如果 svctm 比较接近 await，说明 I/O 几乎没有等待时间；如果 await 远大于 svctm，说明I/O 队列太长，io响应太慢，则需要进行必要优化。如果avgqu-sz比较大，也表示有当量io在等待。

* + 1. **Sar监控网络**



列出当前网络连接。包括协议/IP/端口号/进程号/状态等。

**State 说明**

LISTEN：侦听来自远方的TCP端口的连接请求

SYN-SENT：再发送连接请求后等待匹配的连接请求

SYN-RECEIVED：再收到和发送一个连接请求后等待对方对连接请求的确认

ESTABLISHED：代表一个打开的连接

FIN-WAIT-1：等待远程TCP连接中断请求，或先前的连接中断请求的确认

FIN-WAIT-2：从远程TCP等待连接中断请求

CLOSE-WAIT：等待从本地用户发来的连接中断请求

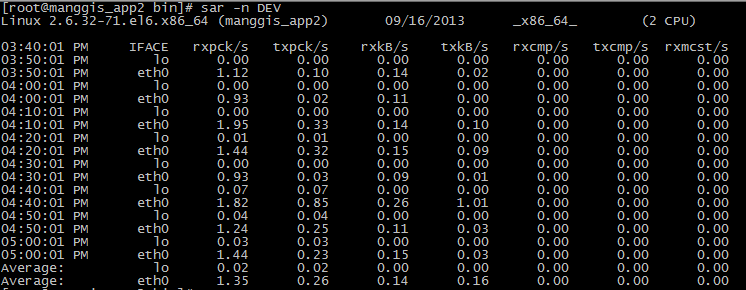
CLOSING：等待远程TCP对连接中断的确认

LAST-ACK：等待原来的发向远程TCP的连接中断请求的确认

TIME-WAIT：等待足够的时间以确保远程TCP接收到连接中断请求的确认

CLOSED：没有任何连接状态

* + 1. **Netstat**



该命令会显示系统中所有网络接口的统计信息，并在最后显示这段时间统计结果的平均值。其中输出结果中各字段的说明如下所示。

IFACE：网络接口的名字。

rxpck/s：每秒钟接收的数据包。

txpck/s：每秒钟发送的数据包。

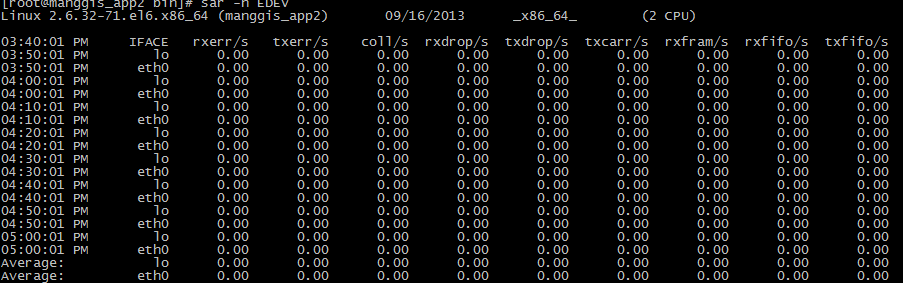
rxbyt/s：每秒钟接收的字节数。

txbyt/s：每秒钟发送的字节数。

rxcmp/s：每秒钟接收的压缩数据包。

txcmp/s：每秒钟发送的压缩数据包。

rxmcst/s：每秒钟接收的多播数据包。



正常情况下是不应该存在网络冲突和错误的，但是当网络流量不断增大的时候，就可能会因为网卡过载而出现丢包等情况。对于网络的错误统计信息，可以通过如下命令获取。

IFACE：网络接口名称。

rxerr/s：每秒钟接收的坏数据包。

txerr/s：每秒钟发送的坏数据包。

coll/s：每秒冲突数。

rxdrop/s：因为缓冲充满，每秒钟丢弃的已接收数据包数。

txdrop/s：因为缓冲充满，每秒钟丢弃的已发送数据包数。

txcarr/s：发送数据包时，每秒载波错误数。

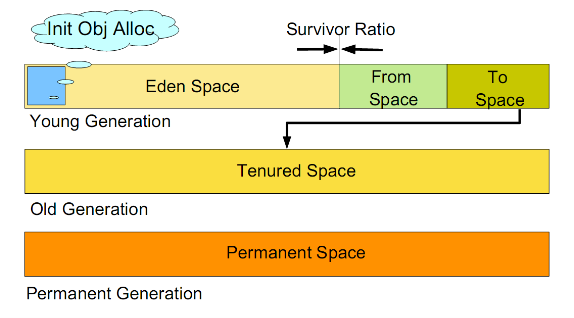
rxfram/s：每秒接收数据包的帧对齐错误数。

rxfifo/s：接收的数据包每秒FIFO过速的错误数。

txfifo/s：发送的数据包每秒FIFO过速的错误数。

1. **JVM性能调优**
   1. **原理**

JVM Heap = permanent Generation + young(new) Generation + tenured (old) Generation.



**permGen:**

    JVM自用区域，是内存永久保存区。用于存放反射代理和class，class在装载时即放在permGen区。

**newGen:**

    所有新生成的对象首先都是放在newGen的。newGen的目标就是尽可能快速的收集掉那些生命周期短的对象。newGen分三个区。一个Eden区，两个Survivor区(一般而言)。Eden用于存放新建对象。当Eden区满时，JVM执行垃圾回收GC。还存活的对象将被复制到Survivor区（两个中的一个），当这个Survivor区满时，此区的存活对象将被复制到另外一个Survivor区，当这个Survivor去也满了的时候，从第一个Survivor区复制过来的并且此时还存活的对象，将被复制“oldGen”。需要注意，Survivor的两个区是对称的，没先后关系，所以同一个区中可能同时存在从Eden复制过来 对象，和从前一个Survivor复制过来的对象，而复制到年老区的只有从第一个Survivor去过来的对象。而且，Survivor区总有一个是空的。同时，根据程序需要，Survivor区是可以配置为多个的（多于两个），这样可以增加对象在newGen中的存在时间，减少被放到oldGen的可能。

**oldGen:**

    在newGen中经历了N次垃圾回收后仍然存活的对象，就会被放到oldGen中。因此，可以认为oldGen中存放的都是一些生命周期较长的对象。

1. GC触发条件

|  |  |  |  |
| --- | --- | --- | --- |
| **GC类型** | **触发条件** | **触发时发生了什么** | **注意** |
| Minor Gc | eden空间不足 | 清空Eden+from survivor中所有no ref的对象占用的内存 将eden+from sur中所有存活的对象copy到to sur中 一些对象将晋升到old中:     to sur放不下的     存活次数超过turning threshold中的 重新计算tenuring threshold(serial parallel GC会触发此项)  重新调整Eden 和from的大小(parallel GC会触发此项) | 全过程暂停应用 是否为多线程处理由具体的GC决定 |
| Full GC | old空间不足 perm空间不足 显示调用System.GC, RMI等的定时触发 YGC时的悲观策略 dump live的内存信息时(jmap –dump:live) | 清空heap中no ref的对象 permgen中已经被卸载的classloader中加载的class信息  如配置了CollectGenOFirst,则先触发YGC(针对serial GC) 如配置了ScavengeBeforeFullGC,则先触发YGC(针对serial GC) | 全过程暂停应用 是否为多线程处理由具体的GC决定  是否压缩需要看配置的具体GC |

* 1. **参数说明**

|  |  |  |  |
| --- | --- | --- | --- |
| **参数名称** | **含义** | **默认值** |  |
| -Xms | 初始堆大小 | 物理内存的1/64(<1GB) | 设置JVM Heap大小的初始值。空余堆内存小于40%时，JVM就会增大堆直到-Xmx的最大限制. |
| -Xmx | 最大堆大小 | 物理内存的1/4(<1GB) | 默认(MaxHeapFreeRatio参数可以调整)空余堆内存大于70%时，JVM会减少堆直到 -Xms的最小限制 |
| -Xmn | 年轻代大小(1.4or lator) |  | **注意**：此处的大小是（eden+ 2 survivor space).与jmap -heap中显示的New gen是不同的。 整个堆大小=年轻代大小 + 年老代大小 + 持久代大小. 增大年轻代后,将会减小年老代大小.此值对系统性能影响较大,Sun官方推荐配置为整个堆的3/8 |
| -XX:NewSize | 设置年轻代大小(for 1.3/1.4) |  | 设置newGen大小的初始值。 |
| -XX:MaxNewSize | 年轻代最大值(for 1.3/1.4) |  |  |
| -XX:PermSize | 设置持久代(perm gen)初始值 | 物理内存的1/64 |  |
| -XX:MaxPermSize | 设置持久代最大值 | 物理内存的1/4 |  |
| -Xss | 每个线程的堆栈大小 |  | JDK5.0以后每个线程堆栈大小为1M,以前每个线程堆栈大小为256K.更具应用的线程所需内存大小进行 调整.在相同物理内存下,减小这个值能生成更多的线程.但是操作系统对一个进程内的线程数还是有限制的,不能无限生成,经验值在3000~5000左右 一般小的应用， 如果栈不是很深， 应该是128k够用的 大的应用建议使用256k。这个选项对性能影响比较大，需要严格的测试。 和threadstacksize选项解释很类似,官方文档似乎没有解释,在论坛中有这样一句话:"” -Xss is translated in a VM flag named ThreadStackSize” 一般设置这个值就可以了。 |
| -XX:ThreadStackSize | Thread Stack Size |  | (0 means use default stack size) [Sparc: 512; Solaris x86: 320 (was 256 prior in 5.0 and earlier); Sparc 64 bit: 1024; Linux amd64: 1024 (was 0 in 5.0 and earlier); all others 0.] |
| -XX:NewRatio | 年轻代(包括Eden和两个Survivor区)与年老代的比值(除去持久代) |  | -XX:NewRatio=4表示年轻代与年老代所占比值为1:4,年轻代占整个堆栈的1/5 Xms=Xmx并且设置了Xmn的情况下，该参数不需要进行设置。 |
| -XX:SurvivorRatio | Eden区与Survivor区的大小比值 |  | 设置为8,则两个Survivor区与一个Eden区的比值为2:8,一个Survivor区占整个年轻代的1/10 |
| -XX:LargePageSizeInBytes | 内存页的大小不可设置过大， 会影响Perm的大小 |  | =128m |
| -XX:+UseFastAccessorMethods | 原始类型的快速优化 |  |  |
| -XX:+DisableExplicitGC | 关闭System.gc() |  | 这个参数需要严格的测试 |
| -XX:MaxTenuringThreshold | 垃圾最大年龄 |  | 如果设置为0的话,则年轻代对象不经过Survivor区,直接进入年老代. 对于年老代比较多的应用,可以提高效率.如果将此值设置为一个较大值,则年轻代对象会在Survivor区进行多次复制,这样可以增加对象再年轻代的存活 时间,增加在年轻代即被回收的概率 该参数只有在串行GC时才有效. |
| -XX:+AggressiveOpts | 加快编译 |  |  |
| -XX:+UseBiasedLocking | 锁机制的性能改善 |  |  |
| -Xnoclassgc | 禁用垃圾回收 |  |  |
| -XX:SoftRefLRUPolicyMSPerMB | 每兆堆空闲空间中SoftReference的存活时间 | 1s | softly reachable objects will remain alive for some amount of time after the last time they were referenced. The default value is one second of lifetime per free megabyte in the heap |
| -XX:PretenureSizeThreshold | 对象超过多大是直接在旧生代分配 | 0 | 单位字节 新生代采用Parallel Scavenge GC时无效 另一种直接在旧生代分配的情况是大的数组对象,且数组中无外部引用对象. |
| -XX:TLABWasteTargetPercent | TLAB占eden区的百分比 | 1% |  |
| -XX:+CollectGen0First | FullGC时是否先YGC | false |  |

**并行收集器相关参数**

|  |  |  |  |
| --- | --- | --- | --- |
| -XX:+UseParallelGC | Full GC采用parallel MSC (此项待验证) |  | 选择垃圾收集器为并行收集器.此配置仅对年轻代有效.即上述配置下,年轻代使用并发收集,而年老代仍旧使用串行收集.(此项待验证) |
| -XX:+UseParNewGC | 设置年轻代为并行收集 |  | 可与CMS收集同时使用 JDK5.0以上,JVM会根据系统配置自行设置,所以无需再设置此值 |
| -XX:ParallelGCThreads | 并行收集器的线程数 |  | 此值最好配置与处理器数目相等 同样适用于CMS |
| -XX:+UseParallelOldGC | 年老代垃圾收集方式为并行收集(Parallel Compacting) |  | 这个是JAVA 6出现的参数选项 |
| -XX:MaxGCPauseMillis | 每次年轻代垃圾回收的最长时间(最大暂停时间) |  | 如果无法满足此时间,JVM会自动调整年轻代大小,以满足此值. |
| -XX:+UseAdaptiveSizePolicy | 自动选择年轻代区大小和相应的Survivor区比例 |  | 设置此选项后,并行收集器会自动选择年轻代区大小和相应的Survivor区比例,以达到目标系统规定的最低相应时间或者收集频率等,此值建议使用并行收集器时,一直打开. |
| -XX:GCTimeRatio | 设置垃圾回收时间占程序运行时间的百分比 |  | 公式为1/(1+n) |
| -XX:+ScavengeBeforeFullGC | Full GC前调用YGC | true | Do young generation GC prior to a full GC. (Introduced in 1.4.1.) |

**CMS相关参数**

|  |  |  |  |
| --- | --- | --- | --- |
| -XX:+UseConcMarkSweepGC | 使用CMS内存收集 |  | 测试中配置这个以后,-XX:NewRatio=4的配置失效了,原因不明.所以,此时年轻代大小最好用-Xmn设置.??? |
| -XX:+AggressiveHeap |  |  | 试图是使用大量的物理内存 长时间大内存使用的优化，能检查计算资源（内存， 处理器数量） 至少需要256MB内存 大量的CPU／内存， （在1.4.1在4CPU的机器上已经显示有提升） |
| -XX:CMSFullGCsBeforeCompaction | 多少次后进行内存压缩 |  | 由于并发收集器不对内存空间进行压缩,整理,所以运行一段时间以后会产生"碎片",使得运行效率降低.此值设置运行多少次GC以后对内存空间进行压缩,整理. |
| -XX:+CMSParallelRemarkEnabled | 降低标记停顿 |  |  |
| -XX+UseCMSCompactAtFullCollection | 在FULL GC的时候， 对年老代的压缩 |  | CMS是不会移动内存的， 因此， 这个非常容易产生碎片， 导致内存不够用， 因此， 内存的压缩这个时候就会被启用。 增加这个参数是个好习惯。 可能会影响性能,但是可以消除碎片 |
| -XX:+UseCMSInitiatingOccupancyOnly | 使用手动定义初始化定义开始CMS收集 |  | 禁止hostspot自行触发CMS GC |
| -XX:CMSInitiatingOccupancyFraction=70 | 使用cms作为垃圾回收 使用70％后开始CMS收集 | 92 | 为了保证不出现promotion failed(见下面介绍)错误,该值的设置需要满足以下公式[**CMSInitiatingOccupancyFraction计算公式**](http://www.cnblogs.com/redcreen/archive/2011/05/04/2037057.html#CMSInitiatingOccupancyFraction_value) |
| -XX:CMSInitiatingPermOccupancyFraction | 设置Perm Gen使用到达多少比率时触发 | 92 |  |
| -XX:+CMSIncrementalMode | 设置为增量模式 |  | 用于单CPU情况 |
| -XX:+CMSClassUnloadingEnabled |  |  |  |

**辅助信息**

|  |  |  |  |
| --- | --- | --- | --- |
| -XX:+PrintGC |  |  | 输出形式:  [GC 118250K->113543K(130112K), 0.0094143 secs] [Full GC 121376K->10414K(130112K), 0.0650971 secs] |
| -XX:+PrintGCDetails |  |  | 输出形式:[GC [DefNew: 8614K->781K(9088K), 0.0123035 secs] 118250K->113543K(130112K), 0.0124633 secs] [GC [DefNew: 8614K->8614K(9088K), 0.0000665 secs][Tenured: 112761K->10414K(121024K), 0.0433488 secs] 121376K->10414K(130112K), 0.0436268 secs] |
| -XX:+PrintGCTimeStamps |  |  |  |
| -XX:+PrintGC:PrintGCTimeStamps |  |  | 可与-XX:+PrintGC -XX:+PrintGCDetails混合使用 输出形式:11.851: [GC 98328K->93620K(130112K), 0.0082960 secs] |
| -XX:+PrintGCApplicationStoppedTime | 打印垃圾回收期间程序暂停的时间.可与上面混合使用 |  | 输出形式:Total time for which application threads were stopped: 0.0468229 seconds |
| -XX:+PrintGCApplicationConcurrentTime | 打印每次垃圾回收前,程序未中断的执行时间.可与上面混合使用 |  | 输出形式:Application time: 0.5291524 seconds |
| -XX:+PrintHeapAtGC | 打印GC前后的详细堆栈信息 |  |  |
| -Xloggc:filename | 把相关日志信息记录到文件以便分析. 与上面几个配合使用 |  |  |
| -XX:+PrintClassHistogram | garbage collects before printing the histogram. |  |  |
| -XX:+PrintTLAB | 查看TLAB空间的使用情况 |  |  |
| XX:+PrintTenuringDistribution | 查看每次minor GC后新的存活周期的阈值 |  | Desired survivor size 1048576 bytes, new threshold 7 (max 15) new threshold 7即标识新的存活周期的阈值为7。 |

* 1. **配置文件修改**

apache-tomcat-7.0.41/bin/ catalina.sh ，修改格式如下：

JAVA\_OPTS="-Djava.awt.headless=true -Dfile.encoding=UTF-8 -server -Xms4000m -Xmx4000m -Xmn2500m -XX:PermSize=512m -XX:MaxPermSize=512m"

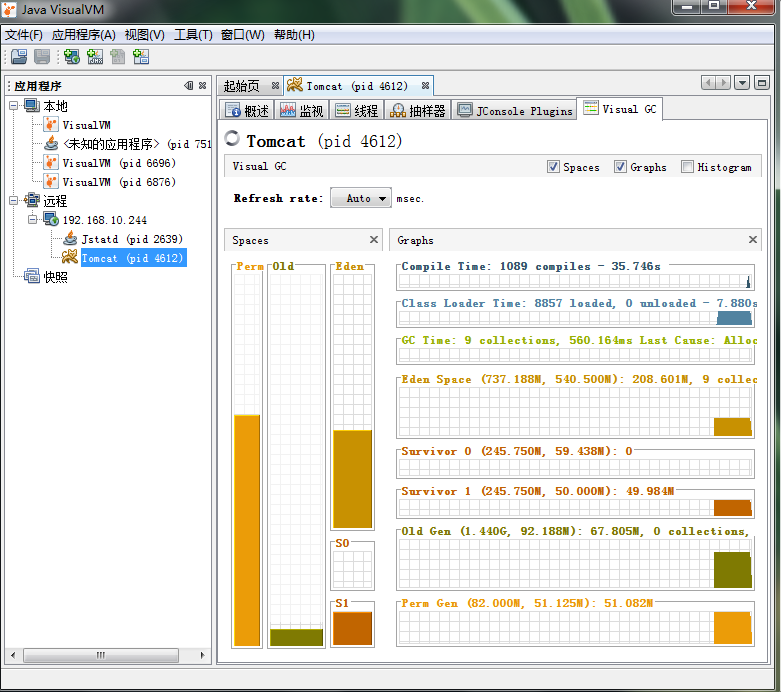
打印GC日志参数

JAVA\_OPTS="-verbose:gc -Xloggc:gc.log -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintHeapAtGC "

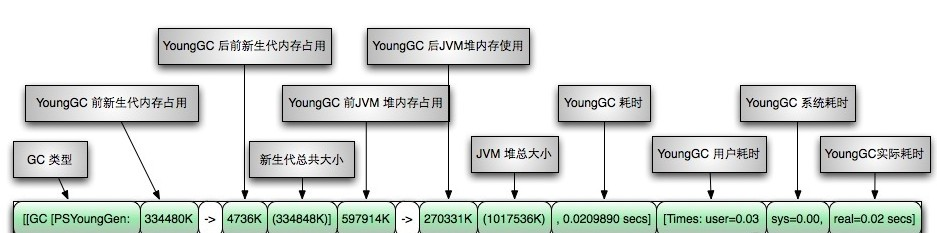
* 1. **JVM监控**

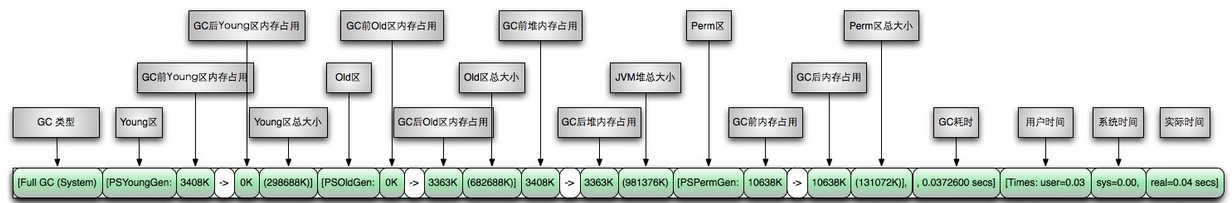
1. Windows操作系统

直接运行JDK自带的\jdk1.6.0\_43\bin\jvisualvm.exe程序即可。



**GC Log 介绍**





二、Linux系统下远程监控

1、配置JXM参数

在apache-tomcat-7.0.41/bin/ catalina.sh，增加参数JAVA\_OPTS。

|  |
| --- |
| JAVA\_OPTS="-Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.port=9004 -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false" |

2、设置jstatd参数

新建jstatd.all.policy文件，内容如下

|  |
| --- |
| grant codebase "file:${java.home}/../lib/tools.jar" {     permission java.security.AllPermission;  }; |

然后运行命令

jstatd -J-Djava.security.policy=jstatd.all.policy &

3、在jvisualvm远程连接linux JVM

在windows上打开jvisualvm程序，选择远程，并输入linux的ip地址。

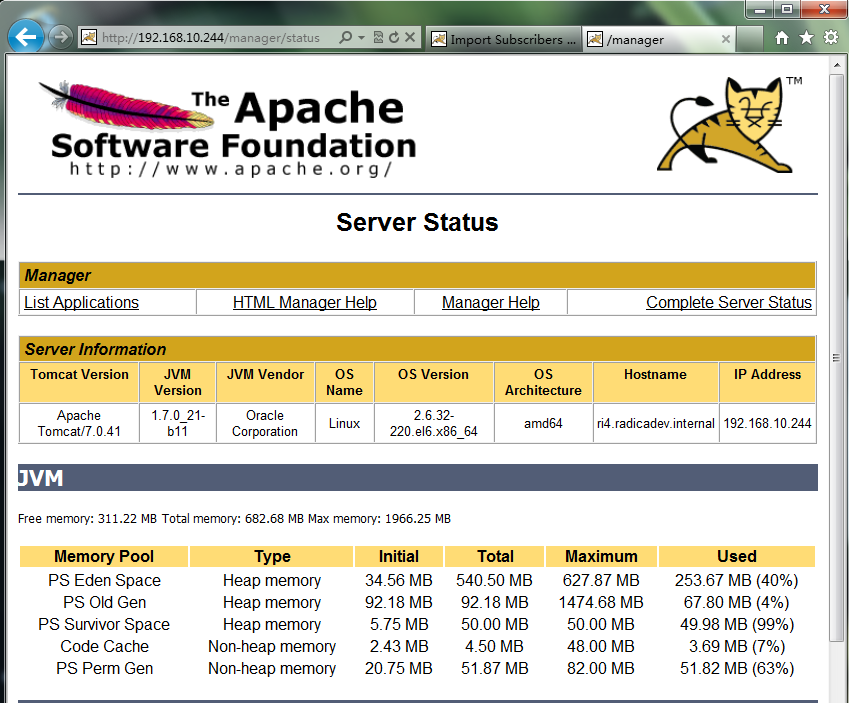
4、需要保证/etc/hosts文件，有本机IP地址和主机名。

三、tomcat自带的网页管理

在/apache-tomcat-7.0.41/conf/ tomcat-users.xml文件添加监控权限。

|  |
| --- |
| <role rolename=" manager-gui"/>  <user username="tomcat" password="tomcat" roles="tomcat,manager-gui"/> |

然后访问<http://IP:port/manager/status>页面即可。



1. **JVM高级特性**
   1. **JAVA内存区域**

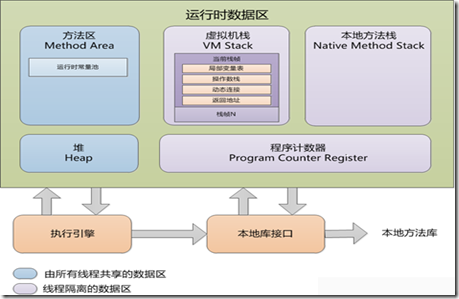


Figure 1 JAVA虚拟机运行时数据区

实例：object obj=new object();

“object obj”部分会反映到java栈的本地变量表，作为一个reference数据类型存在。

“new object()”部分会反映到java堆，形成一颗存储了object类型所有实例数据值(Instance Data,对象中各个实例字段的数据)的结构化内存。

Java堆中还必须包含能查找到此对象类型数据(如对象类型、父类、实现的接口、方法等)的地址信息，这些类型数据存储在方法区中。



Figure 通过句柄访问对象

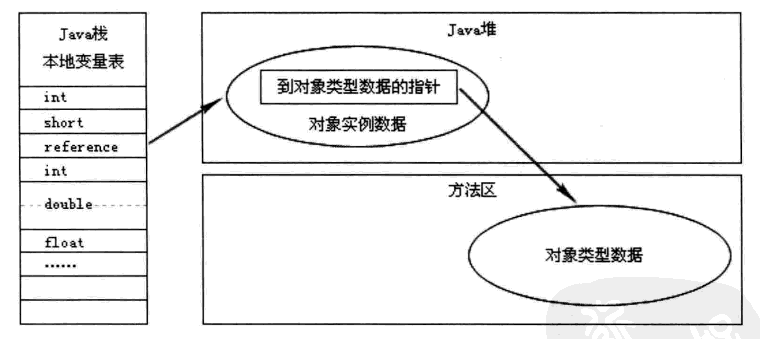


Figure 通过直接指针访问对象

* 1. **内存溢出异常**
     1. **Java堆溢出**

在JVM中如果98%的时间用于GC，且可用的Heap大小不足2%时，则抛出异常信息”java.lang.OutOfMemoryError:Java heap space”

**promotion failed**

一般可能是两种原因产生，第一个原因是救助空间不够，救助空间里的对象还不应该被移动到年老代，但年轻代又有很多对象需要放入救助空间；第二个原因是年老代没有足够的空间接纳来自年轻代的对象；这两种情况都会转向Full GC。

* + 1. **方法区溢出(perm Gen)**

异常堆栈信息”java.lang.OutOfMemoryError:permGen space” 由于GC不会在主程序运行期间对PermGen进行处理，如果应用程序有很多class的话，就可能出现permGen space错误。

* + 1. **虚拟机栈和本地方法栈溢出**

如果线程请求的栈深度大于虚拟机所允许的最大深度，则抛出StackOverflowError异常。

如果虚拟机在扩展栈时无法申请到足够内存空间，则抛出OutOfMemoryError:unable to create new native thread异常。

* + 1. **本机直接内存溢出**

DirectMemory容量可通过-XX:MaxDirectMemorySize指定。如果不指定，默认与Java堆的最大值(-Xmx指定)一样。异常堆栈信息”java.lang.OutOfMemoryError”或者”OutOfMemoryError:direct buffer memory”

* 1. **OutOfMemory异常信息诊断**

1、Exception in thread “main” java.lang.OutOfMemoryError: Java heap space

1. 这个信息表明无法在java 堆内存中分配对象空间。不过这个问题很可能是由于配置不当。例如使用- Xmx参数指定堆内存的最大值，这个值的设置（或者默认设置）不能满足应用的要求。
2. 另外，特别是对于那些长时间运行的系统，这可能表明应用或者应用使用的API始终持有堆中对象的引用，这使得垃圾回收器无法将其内存释放。这在java语言中等同于内存泄漏。
3. 另外一个引起OutOfMemoryError的潜在原因在于过多的使用finalizer。如果一个类覆盖了Object类的finalize方法，并且该类的对象在垃圾回收期间没有被回收。那么在垃圾回收之后的某个时候对象排队等待进行finalization。在sun虚拟机的实现中，finalizer是一个处理finalization队列的精灵线程。如果这个finalizer的线程不能持续处理finalization队列，很有可能造成堆内存溢出并抛出OutOfMemoryError错误信息。

有这样一个例子可以产生这样的错误：当应用程序创建了许多的高优先级别的线程，引起了finalization队列大量的增长，其增长的速度要大于finalizer线程处理finalization队列的速度。

2、Exception in thread “main” java.lang.OutOfMemoryError: PermGen space

1. 这说明permanent generation已经填满。permanent generation是class和method对象在堆中存储的区域。如果一个应用程序加载了大量的类，那么需要调高-XX:MaxPermSize参数选项。
2. 当String对象处于interned状态（share unique instances）的时候也会占用permanent generation内存。如果一个应用使用了大量的interned String，那么也需要调高PermSize的内存设定。

3、Exception in thread “main” java.lang.OutOfMemoryError: Requested array size exceeds VM limit

这样的错误表明应用或者应用使用的API尝试去分配比堆内存容量更大的数组变量。例如如果应用想要分配512M的内存空间，但堆内存最大容量为256M，就会引起该错误。大部分的情况下这是一个配置的问题（堆容量过小），或者是应用的bug使得应用尝试去创建一个超大的数组。

4、Exception in thread “main” java.lang.OutOfMemoryError: request <size> bytes for <reason>. Out of swap space?

尽管这看起来好像是抛出OutOfMemoryError错误，但是显然是HotSpot虚拟机代码在本地堆分配内存失败并且本地堆耗尽的时候报告出来的异常。这个信息表明了请求失败的内存数量<size>，也表明了请求的是什么内存。一些情况下会打印出相关的<reason>，但是大部分情况下报告分配内存失败的源模块的名称。

这种情况下需要使用操作系统的工具去诊断问题。产生这个问题的一个案例就是操作系统设置的过小的swap空间，或者操作系统的其他进程消耗了大量的内存资源。如果都不是以上的原因，那么可能是由于本地的内存泄漏造成了应用的失败；举个例子，应用或者本地库不停的分配内存但是从不释放给操作系统。

5、Exception in thread "main" java.lang.OutOfMemoryError: <reason> <stack trace>(Native method)

如果抛出OutOfMemoryError错误信息，同时打印出stack trace的栈顶是Native方法，那么这表明一个native方法引起了内存的分配失败。这种状况和上面的信息不同的地方在于是在JNI/native方法中发现的分配失败，而不是VM code。和上面的信息一样，需要使用操作系统的工具来诊断问题放生的原因。

6、OutOfMemoryError还没有抛出JVM就已经崩溃

有的时候虚拟机本地堆的内存分配失败直接造成应用的瞬间崩溃。这就造成本地代码没有去检查由内存分配函数返回是否为NULL。如果系统没有可用内存，malloc系统调用会返回NULL。如果没有检查返回的内存地址，应用就去访问不可用的内存就会造成瞬间崩溃。这完全取决于环境，并且这种问题很难定位。但是通常致命错误日志信息或者crash dump信息可以用来诊断问题发生的原因。如果crash被诊断为内存分配失败未被检查返回结果，必须去查找内存分配失败的原因。如果是由于本地堆内存的原因，这可能是由于配置了不足的swap空间，系统的其他进程消耗了所有的内存资源，或者应用中存在内存泄漏引起系统内存不足。

* 1. **垃圾收集**
     1. **垃圾收集算法**

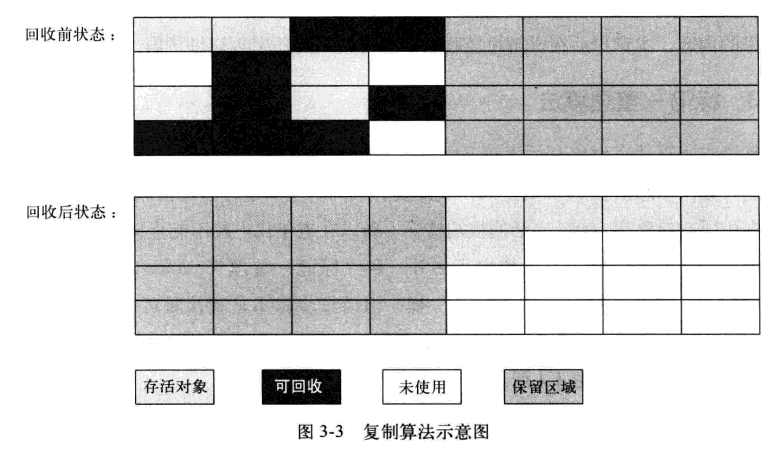
**1、标记-清除算法Mark-Clean**

通过GC Roots遍历被引用的对象，标记为可达，未被遍历到的对象就是不再被引用的对象，然后遍历一遍把不再被引用的对象占用的内存回收掉。但是这种算法会带来一个问题，就是内存碎片问题，如果每次按照对象大小分配，内存用完之后回收，可用空间零散散布于这个堆中，下次申请比较大的内存的时候，可能就申请不到很大的连续空间，只有一堆不连续的小空间，此时申请内存就会失败，而从外部来看，堆中总共还有足够的空闲空间来分配。



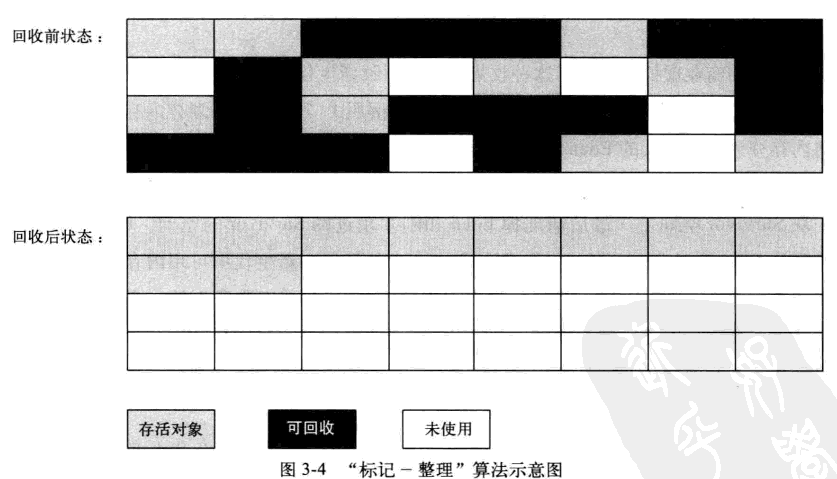
**2、拷贝算法Copy**

针对标记-清除算法的碎片问题，出现了一种改进的算法，把堆内存划分为两块，每次使用一块，当一块满的时候，就去标记使用中的对象，然后把使用中的对象拷贝到另外一个块里连续的内存空间中，然后把这个块的内存全部回收，后续创建对象申请空间就直接在另外一个块中申请。然后另外一个空间满的时候重复此过程。此算法解决了内存碎片问题，但是会降低内存的使用效率，一直都只有一半的空间被使用。同时考虑这样一个问题，如果有些对象一直被使用，如java.lang.Object这种在jvm的生命周期中一直会被使用的对象，copy来copy去的还是效率很低的。



**3、标记-清理算法（也叫压缩算法）Mark-Sweep**

针对拷贝算法的空间使用效率问题，标记整理算法给出了解决办法。在内存空间占满的时候，先通过GC Roots标记出存活的对象，然后把存活的对象从堆空间的一端，移动到另外一端，并使得他们占用的内存空间是连续的。移动完后，另一端就是一大块连续的空间，后续分配就在连续的空间进行，如果再次满了，就重复标记整理过程。但是这种算法依然存在拷贝算法存在的一个问题，就是长期存活的对象会被移动来移动去的。

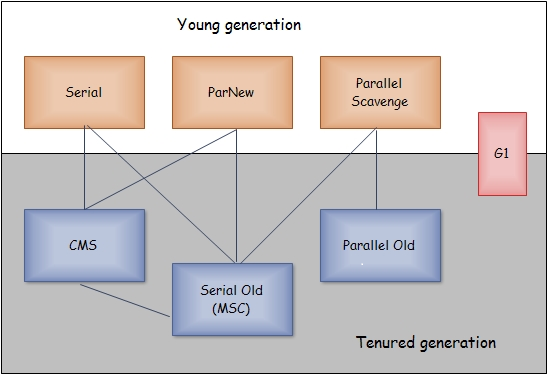


**4、分代收集算法**

针对标记-清理算法中出现的问题，分代收集算法给出了解决办法。分代收集算法把根据对象存活的时间把堆内存空间多个部分，一般有新生代、中生代和老生代，不同的世代中的对象一般会用不同的算法来收集，如新生代和中生代很多朝生夕死的对象用拷贝算法拷贝效率也比较高，收集频率也比较高。老生代因为存活时间都很长了，并且可能一直存活下去，拷贝来拷贝去拷贝量很大，所以一般采用标记整理算法，收集的频率一般也比较低。

* + 1. **垃圾收集器**

垃圾收集器是特定于实现的收集器，都是基于以上算法实现的。不同的虚拟机，实现的细节也不太相同，此处只是一些概念和思想上的讲解。

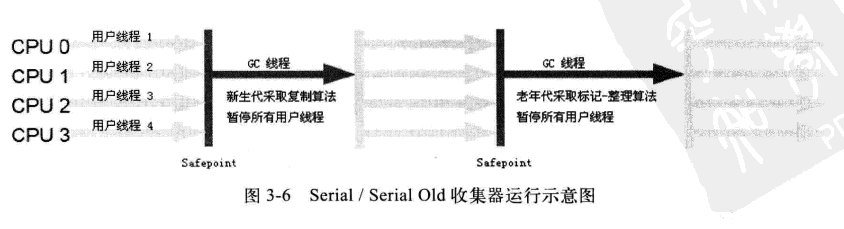


1、Serial

Serial是新生代的单线程垃圾收集器，使用Copy算法。当进行垃圾收集的时候，别的线程都停止工作，Stop the World。这种收集器在多cpu上无法充分利用cpu资源，另外由于是单线程工作，停止时间也比较长。

在HotSpot JVM中使用-XX:+UserSerialGC会启动该收集器，同时老生代用SerialOld。是client模式默认的收集器。

该收集器可以跟老生代的Serial Old、CMS协同工作。

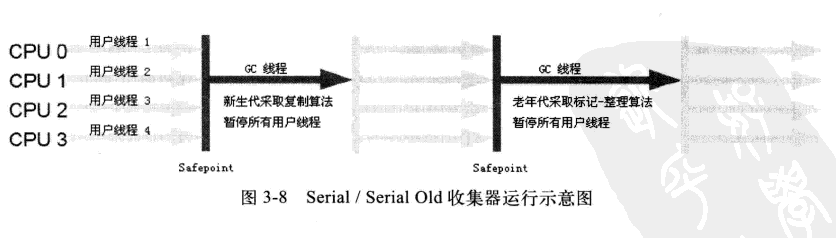


2、SerialOld

使用Mark-Sweep算法，是Serial的老生代GC收集器版本，单线程，Stop the World。

该收集器可以跟新生代的Serial、SerNew、Parallel Scavanage 协同工作。

该收集器还作为CMS失败时候的后备收集器。



3、ParNew

新生代收集器，是Serial的多线程版本，可以充分利用机器的cpu，收集的时候Stop the World。在单线程下也可以利用多线程，但是效率可能不如Serial，如果是多cpu，效率会很高。

在HotSpot JVM中使用-XX:UseParNew会启用该收集器，同时老生代使用SerialOld。

该收集器可以跟老生代的SerialOld、CMS协作收集。



4、Parallel Scavenge

新生代的收集器，也叫PS Scavenge，比起ParNew，此收集器的目的是达到一个可控的吞吐量，所以也叫做吞吐量优先收集器。可以通过参数-XX:+UseAdaptiveSizePolicy使用自适应内存调节策略，同时设置上-XX:MaxGCPauseMillis设置最大停机时间，通过-XX:GCTimeRatio设置吞吐量。

通过参数-XX:+UseParallelGC启用该收集器，此时老生代使用Serial Old（也叫PS MarkSweep，是在Serial Old的外边加了一层壳，本质上还是serial，很多地方也这么叫）

此收集器可以跟老生代Serial Old和Parallel Old协作。

5、Parallel Old

老生代并行收集器，Parallel Scavenge的老生代版本，吞吐量优先（但是没有相关参数设置老生代度量的？）

通过参数-XX:+UseParallelOldGC启用，新生代是Parallel Scavenage收集器。

注意：在jconsole中看到，-XX:+UseParallelOldGC时候老生代的内存收集器是PS MarkSweep，不是Parallel Old这个名字，[原因这里有](http://hllvm.group.iteye.com/group/topic/27629)。



6、CMS/ConcurrentMarkSweep

老生代收集器，Concurrent Low Pause Collector，收集阶段分为四个阶段：

a、初始标记（initial mark），串行执行。从GC roots开始标记由roots直接关联的对象

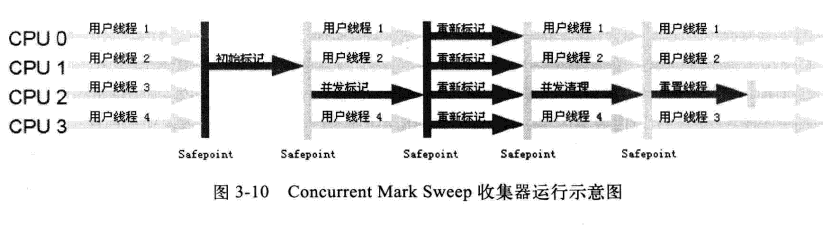
b、并发标记（concurrent mark），并发执行。用户线程和标记线程同时工作，根据上一步标记出的对象进一步标记整个引用链。

c、重新标记（remark），串行执行。在并发标记完成后，再进行一次并行标记，处理在并发标记阶段引用改变的对象。

d、并发收集（concurrent mark），并发收集。gc线程跟用户线程一起运行，收集内存。

此收集器用的是Mark-Sweep算法，会导致内存随便，可能会出现总内存足够，但是分配某个大小的内存时候没有空间的情况，此时就是用Serial Old收集器作为失败后的收集器来收集。

通过参数-XX:+UseConcMarkSweepGC启用该收集器，新生代收集器默认是ParNew，并发模式失败后是用Serial Old收集器。



7、G1收集器

Garbage First，将在jdk1.7中发布。比起cms有两个优势，一是g1基于mark-compact算法，不会产生内存碎片；二是可以精确控制停顿。该收集器是把整个堆（新生代、老生代）划分成多个固定大小的区域，每次根据允许停顿的时间去收集垃圾最多的区域。

* 1. **性能参数设置**

通过上面的介绍，已经讲明了JVM参数与性能稳定之间的关系。接下来将进行一些实验验证。

环境：CentOS6.0，Intel(R) Xeon(R) CPU E5-2630 0 @ 2.30GHz \*2, Memory 4G

测试步骤：通过性能测试工具LoadRunner模拟200个用户并发访问RI系统，运行相同的时间。

* + 1. **实验一: 减少分配内存**

JAVA\_OPTS="-Djava.awt.headless=true -Dfile.encoding=UTF-8 -server -Xms100m -Xmx100m -Xmn40m -XX:PermSize=64m -XX:MaxPermSize=64m "

说明：当分配给JVM的堆内存较少时，程序不能申请到足够的堆内存，几乎一直处于gc状态。此时程序无法正常运作，报java.lang.OutOfMemoryError。



* + 1. **实验二：默认配置**

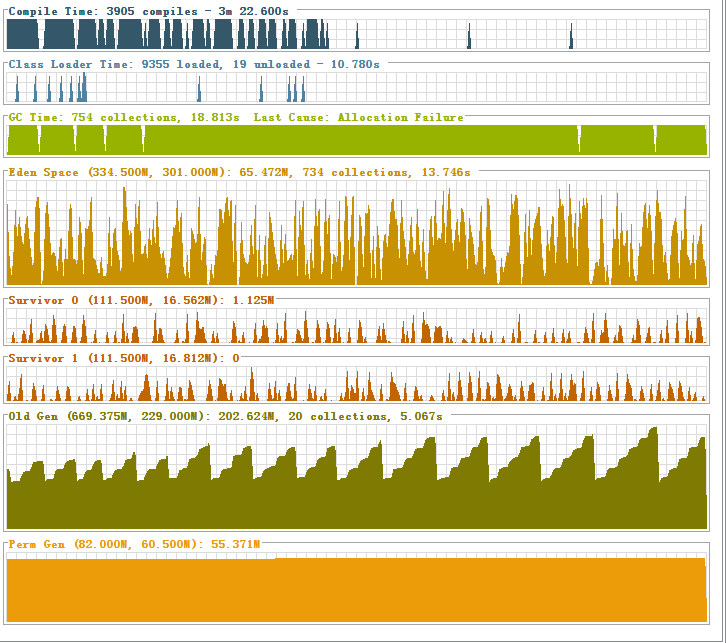
即xms约为1/64物理内存，xmx约为1/4物理内存。Perm约为1/64物理内存。

说明：jvm采用默认的设置。在实验一的基础上，增加了堆内存。同样的业务压力下，实验二的参数配置，业务能够正常的运行。

但是通过下图曲线，可以看到因为xms和xmx参数不一致，jvm一致在动态调整eden和old Gen的大小，增加额外的开销。Xmn的值设置较小，导致minor gc数量较多。同时看到old Gen在full Gc后，仍有部分的内存没有释放。长时间运行后，仍存在较大内存溢出的可能。

Perm Gen分配的1/64内存也不能满足系统要求，已经占用了90%。也存在较大内存溢出可能。

在默认的参数配置下，系统不能充分使用。



* + 1. **实验三：增加分配内存**

JAVA\_OPTS="-Djava.awt.headless=true -Dfile.encoding=UTF-8 -server –Xms2048m –Xmx2048m –Xmn800m -XX:PermSize=256m -XX:MaxPermSize=256m"

说明：进一步调整JVM的内存分配。Xms和Xmx配置为同样大小，分配1/2 物理内存，xmn分配xmx的40%内存。Perm分配512m内存。

下图可见，系统GC次数和总时间与实验二相比，都有减少，特别是full gc。并且old Gen在full gc后，内存能够清空。Perm Gen也有足够的内存。

由此可见，实验三的设置方案是优于实验二的。那么是否分配给JVM越多内存就越好呢？这个不是必然的。

当分配给JVM的内存越大时，单个GC耗时会增大。对比full gc单次时间实验二要少于实验三。当分配的内存很大时，full gc耗时继续增大，会造成程序暂停响应，影响用户使用。同时，分配给JVM的内存多，则本地内存就会减少，当程序启动线程较多时，会增大本地内存溢出的可能。

**推荐的参数设置：建议xms与xmx设置一样大小，约占物理内存的50%。Xmn约占xmx的40%，要小于old Gen。perm Gen根据实际使用设定。**

**参数的配置不是一成不变，一定要根据实际的硬件情况和软件运行情况，进行试验调整。**



* + 1. **实验四：垃圾收集优化**

备注：实验四服务端有变化，数据只在同组内比较。

（4-1）JAVA\_MEM\_OPTS=" -server -Xmx2g -Xms2g **-Xmn512m** -XX:PermSize=256m -Xss256k -XX:SurvivorRatio=8 -XX:MaxTenuringThreshold=7 -XX:GCTimeRatio=19  -Xnoclassgc -XX:+UseParNewGC -XX:+DisableExplicitGC -XX:+UseConcMarkSweepGC -XX:+CMSPermGenSweepingEnabled -XX:+CMSParallelRemarkEnabled -XX:+UseCMSCompactAtFullCollection -XX:CMSFullGCsBeforeCompaction=0 -XX:+CMSClassUnloadingEnabled -XX:CMSInitiatingOccupancyFraction=70 -XX:SoftRefLRUPolicyMSPerMB=0 -XX:+UseFastAccessorMethods -XX:+UseCMSInitiatingOccupancyOnly -XX:+UseCompressedOops -XX:-HeapDumpOnOutOfMemoryError"

（4-2）JAVA\_MEM\_OPTS=" -server -Xmx2g -Xms2g **–Xmn850m** -XX:PermSize=256m -Xss256k -XX:SurvivorRatio=8 -XX:MaxTenuringThreshold=7 -XX:GCTimeRatio=19  -Xnoclassgc -XX:+UseParNewGC -XX:+DisableExplicitGC -XX:+UseConcMarkSweepGC -XX:+CMSParallelRemarkEnabled -XX:+UseCMSCompactAtFullCollection -XX:CMSFullGCsBeforeCompaction=0 -XX:+CMSClassUnloadingEnabled -XX:CMSInitiatingOccupancyFraction=70 -XX:SoftRefLRUPolicyMSPerMB=0 -XX:+UseFastAccessorMethods -XX:+UseCMSInitiatingOccupancyOnly -XX:+UseCompressedOops -XX:-HeapDumpOnOutOfMemoryError"

参数详解：

-XX:PermSize=256m，分配perm内存256M。

-Xmx2g -Xms2g -Xmn512m ，分配堆内存(new+old) 共2G。其中new内存512M。

-XX:SurvivorRatio=8，eden与survivor from 与survivor to比例为8:1:1。

-XX:MaxTenuringThreshold=7，指new区对象在survivor多次(7)复制后再进入old区。可以增加在new区被回收的概率。减少进入old区。

-Xss256k，线程堆栈的大小。内存相同的情况下，减少这个值可以生成更多的线程。

-XX:GCTimeRatio=19 ，设置最大的垃圾回收时间占程序运行时间的百分比。1/(1+19)=5%。仅在使用parallel Scavenge回收器时生效。

-Xnoclassgc，缺省情况下，当一个类没有任何活动实例时，JVM就会从内存中卸载该类。卸载类会使性能下降。如果关闭类垃圾回收，就可以消除由于多次装入和卸载同一个类而造成地开销。

-XX:+UseParNewGC，设置new区垃圾收集器为parNew。

-XX:+DisableExplicitGC，关闭System.gc()。

-XX:+UseConcMarkSweepGC ，设置old区垃圾收集器为CMS。

-XX:+CMSParallelRemarkEnabled，降低标记停顿。

-XX:+UseCMSCompactAtFullCollection，在FULL GC的时候， 对old区进行压缩。

-XX:CMSFullGCsBeforeCompaction=0 ，多少次FULL GC后进行old区压缩

-XX:+CMSPermGenSweepingEnabled，对perm区进行垃圾的回收。JDK1.5使用

-XX:+CMSClassUnloadingEnabled，对perm区进行垃圾的回收。JDK1.6使用

-XX:CMSInitiatingOccupancyFraction=70，使用cms作为垃圾回收，使用70％后开始CMS收集。

-XX:SoftRefLRUPolicyMSPerMB=0 ，指定每兆堆空闲空间的 soft reference 保持存活（一旦它不强可达了）的毫秒数，这意味着每兆堆中的空闲空间中的 soft reference 会（在最后一个强引用被回收之后）存活1秒钟。注意，这是一个近似的值，因为 soft reference 只会在垃圾回收时才会被清除，而垃圾回收并不总在发生。

-XX:+UseFastAccessorMethods ，原始类型的快速优化。get,set 方法转成本地代码。

-XX:+UseCMSInitiatingOccupancyOnly，只有在设置初始化比例后，才启动cms收集器。

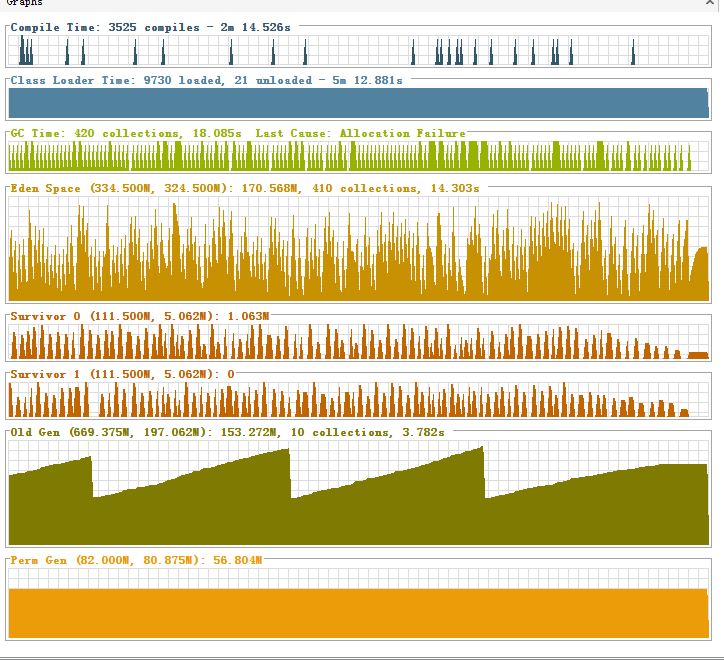
-XX:+UseCompressedOops，提供对象引用压缩功能。利用这些选项，即可在 32 位（而非 64 位）中存储对象字段和对象元数据值。在应用程序从 32 位 Java 运行时迁移到 64 位 Java 运行时的时候，这能消除 Java 堆内存使用量增加 70% 的负面影响。

HeapDumpOnOutOfMemoryError ，出现OutOfMemory时，将当时JVM内所有heap dump出来

说明：配置中增加了垃圾收集器相关配置，New Gen 设置为ParNew收集器，Old Gen设置为CMS收集器。垃圾最大回收阈值设置为7，增加资源在new Gen被回收的可能，减少进入old Gen，减少Full Gc的可能。

通过测试结果可以看到，实验四的配置情况下，GC次数和时间总数都有所减少，特别是full gc。保证了系统的稳定运行和用户响应体验。

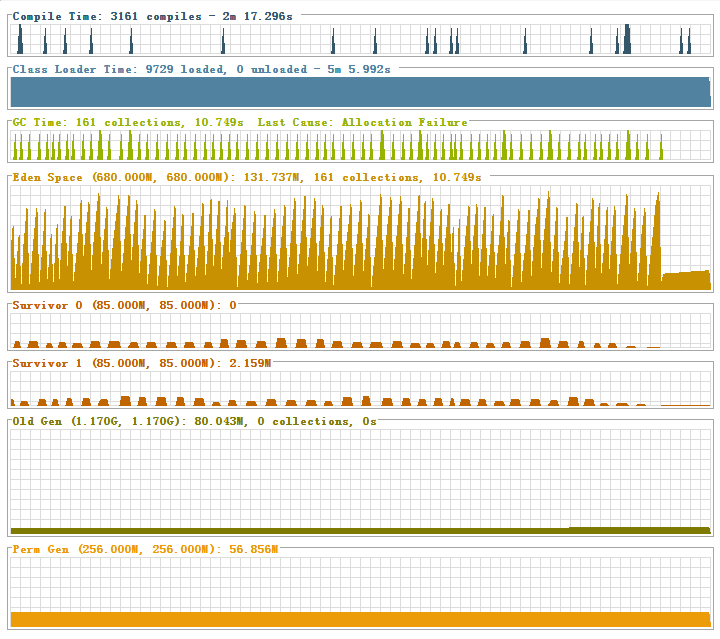
**通过实验三和四的对比可以看出，同样内存分配条件下，对垃圾收集器等参数的合理选择，对系统的稳定运行和有效运转有很大的提升。**



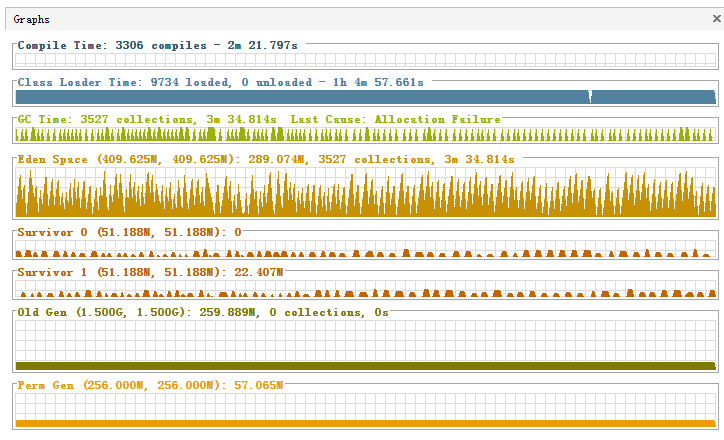
JVM默认配置结果（10min）



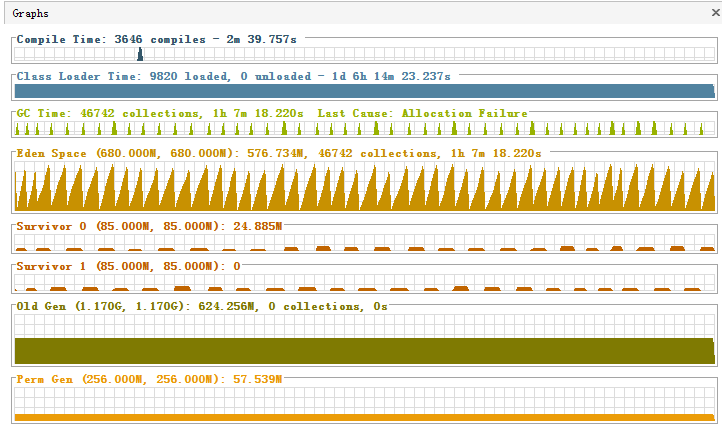
JVM4-1配置结果（10min）



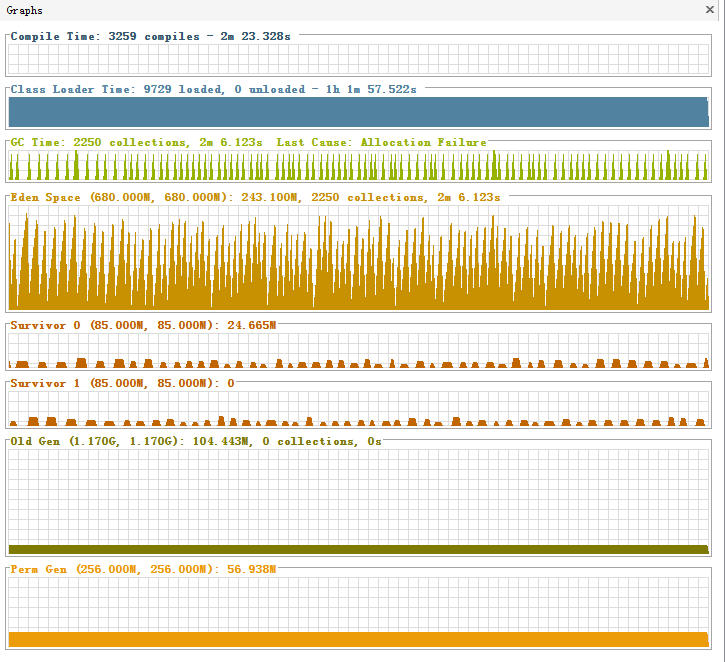
JVM4-2配置结果（10min）



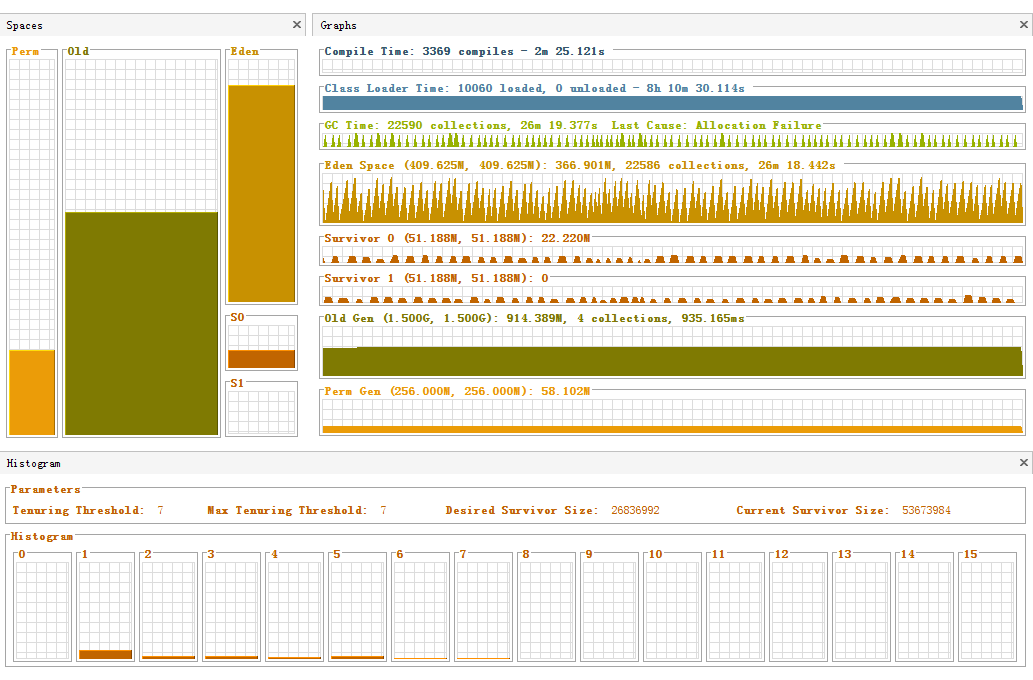
JVM4-1配置结果（2.5h）



JVM4-2配置结果（70h）



JVM4-2配置结果（2.5h）



JVM4-1配置结果（18h）

* + 1. **实验五：tomcat和jetty比较**

将同一服务分别部署在同样硬件条件下的Jetty和tomcat服务器。采用JVM配置为

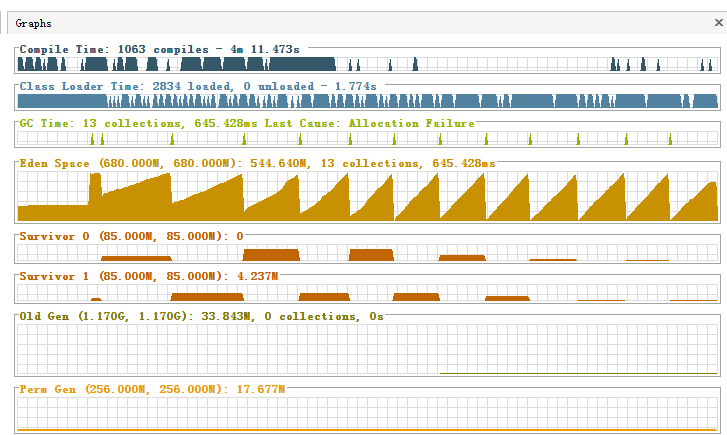
JAVA\_MEM\_OPTS=" -server -Xmx2g -Xms2g **–Xmn850m** -XX:PermSize=256m -Xss256k -XX:SurvivorRatio=8 -XX:MaxTenuringThreshold=7 -XX:GCTimeRatio=19  -Xnoclassgc -XX:+UseParNewGC -XX:+DisableExplicitGC -XX:+UseConcMarkSweepGC -XX:+CMSParallelRemarkEnabled -XX:+UseCMSCompactAtFullCollection -XX:CMSFullGCsBeforeCompaction=0 -XX:+CMSClassUnloadingEnabled -XX:CMSInitiatingOccupancyFraction=70 -XX:SoftRefLRUPolicyMSPerMB=0 -XX:+UseFastAccessorMethods -XX:+UseCMSInitiatingOccupancyOnly -XX:+UseCompressedOops -XX:-HeapDumpOnOutOfMemoryError"

通过性能测试工具LR启动200个用户，并发向服务器端发送一个请求（返回一个字符串，没有业务逻辑）。对比tomcat和jetty的gc记录，可以看到tomcat的运行更稳定，gc的次数和时间都远少于jetty。

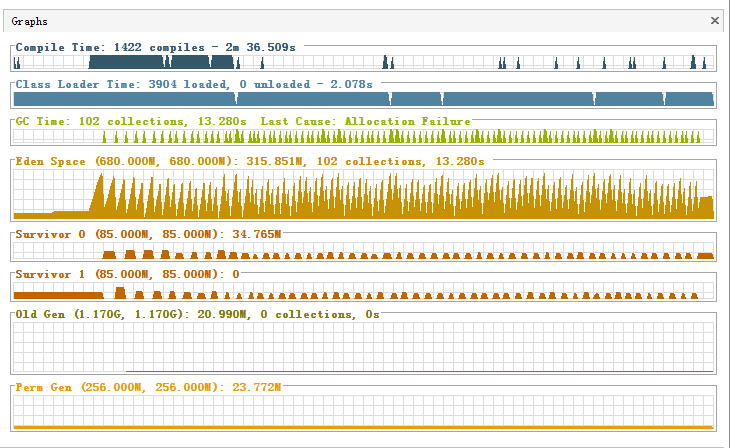
对比运行结果，更可以看出tomcat的(每秒点击数)要远大于jetty，每个事务的处理时间也快于jetty。

对比运行状态的JVM监控，由此可见，该业务情况下tomcat要优于jetty。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **tomcat** | **jetty** | **tomcat2** | **jetty3** |
| 并发用户 | 200 | 200 |  |  |
| [Average Hits per Second](file:///C:\Users\ivy.fan\AppData\Local\Temp\AppData\Local\Temp\HitsperSecond) | 12,521 | 4,502 |  |  |
| 事务运行平均时间 |  |  |  |  |
| Test(s) | 0.016 | 0.043 |  |  |



Tomcat 200用户运行5min



Jetty 200用户运行5min

通过性能工具ab对服务器进行一个简单的get请求的压力测试。不断的增加并发数量来测试jetty和tomcat处理请求的压力。从测试结果来看，Jetty并发处理速度却略优于tomcat。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Concurrent Requests** | **Connect(ms)** | | **Processing(ms)** | | **Waiting(ms)** | | **Requests per second** | |
|  | Jetty | tomcat | Jetty | tomcat | Jetty | tomcat | Jetty | tomcat |
| 10 | 2 | 3 | 40 | 46 | 39 | 44 | 178.25 | 189.04 |
| 50 | 3 | 3 | 222 | 380 | 211 | 375 | 211.15 | 182.22 |
| 100 | 3 | 3 | 473 | 520 | 435 | 466 | 201.57 | 185.91 |
| 200 | 4 | 4 | 816 | 1085 | 582 | 1022 | 207.99 | 167.22 |
| 400 | 4 | 7 | 1723 | 2803 | 1084 | 2443 | 194.12 | 134.14 |
| 500 | 4 | 7 | 2259 | 3417 | 1472 | 2634 | 175.86 | 128.93 |

**Jetty与 Tomcat 的比较**

Tomcat 和 Jetty 都是作为一个 Servlet 引擎应用的比较广泛，虽然 Jetty 正常成长为一个优秀的 Servlet 引擎，但是目前的 Tomcat 的地位仍然难以撼动。相比较来看，它们都有各自的优点与缺点。

Tomcat 经过长时间的发展，它已经广泛的被市场接受和认可，相对 Jetty 来说 Tomcat 还是比较稳定和成熟，尤其在企业级应用方面，Tomcat 仍然是第一选择。但是随着 Jetty 的发展，Jetty 的市场份额也在不断提高，至于原因就要归功与 Jetty 的很多优点了，而这些优点也是因为 Jetty 在技术上的优势体现出来的。

**架构比较**

从架构上来说，显然 Jetty 比 Tomcat 更加简单。

Jetty 的架构所有组件都是基于 Handler 来实现，当然它也支持 JMX。但是主要的功能扩展都可以用 Handler 来实现。可以说 Jetty 是面向 Handler 的架构，就像 Spring 是面向 Bean 的架构，iBATIS 是面向 statement 一样，而 Tomcat 是以多级容器构建起来的，它们的架构设计必然都有一个“元神”，所有以这个“元神“构建的其它组件都是肉身。

从设计模板角度来看 Handler 的设计实际上就是一个责任链模式，接口类 HandlerCollection 可以帮助开发者构建一个链，而另一个接口类 ScopeHandler 可以帮助你控制这个链的访问顺序。另外一个用到的设计模板就是观察者模式，用这个设计模式控制了整个 Jetty 的生命周期，只要继承了 LifeCycle 接口，你的对象就可以交给 Jetty 来统一管理了。所以扩展 Jetty 非常简单，也很容易让人理解，整体架构上的简单也带来了无比的好处，Jetty 可以很容易被扩展和裁剪。

相比之下，Tomcat 要臃肿很多，Tomcat 的整体设计上很复杂，前面说了 Tomcat 的核心是它的容器的设计，从 Server 到 Service 再到 engine 等 container 容器。作为一个应用服务器这样设计无口厚非，容器的分层设计也是为了更好的扩展，这是这种扩展的方式是将应用服务器的内部结构暴露给外部使用者，使得如果想扩展 Tomcat，开发人员必须要首先了解 Tomcat 的整体设计结构，然后才能知道如何按照它的规范来做扩展。这样无形就增加了对 Tomcat 的学习成本。不仅仅是容器，实际上 Tomcat 也有基于责任链的设计方式，像串联 Pipeline 的 Vavle 设计也是与 Jetty 的 Handler 类似的方式。要自己实现一个 Vavle 与写一个 Handler 的难度不相上下。表面上看，Tomcat 的功能要比 Jetty 强大，因为 Tomcat 已经帮你做了很多工作了，而 Jetty 只告诉，你能怎么做，如何做，有你去实现。

打个比方，就像小孩子学数学，Tomcat 告诉你 1+1=2，1+2=3，2+2=4 这个结果，然后你可以根据这个方式得出 1+1+2=4，你要计算其它数必须根据它给你的公式才能计算，而 Jetty 是告诉你加减乘除的算法规则，然后你就可以根据这个规则自己做运算了。所以你一旦掌握了 Jetty，Jetty 将变得异常强大。

**性能比较**

单纯比较 Tomcat 与 Jetty 的性能意义不是很大，只能说在某种使用场景下，它表现的各有差异。因为它们面向的使用场景不尽相同。从架构上来看 Tomcat 在处理少数非常繁忙的连接上更有优势，也就是说连接的生命周期如果短的话，Tomcat 的总体性能更高。

而 Jetty 刚好相反，Jetty 可以同时处理大量连接而且可以长时间保持这些连接。例如像一些 web 聊天应用非常适合用 Jetty 做服务器，像淘宝的 web 旺旺就是用 Jetty 作为 Servlet 引擎。

另外由于 Jetty 的架构非常简单，作为服务器它可以按需加载组件，这样不需要的组件可以去掉，这样无形可以减少服务器本身的内存开销，处理一次请求也是可以减少产生的临时对象，这样性能也会提高。另外 Jetty 默认使用的是 NIO 技术在处理 I/O 请求上更占优势，Tomcat 默认使用的是 BIO，在处理静态资源时，Tomcat 的性能不如 Jetty。

**特性比较**

作为一个标准的 Servlet 引擎，它们都支持标准的 Servlet 规范，还有 Java EE 的规范也都支持，由于 Tomcat 的使用的更加广泛，它对这些支持的更加全面一些，有很多特性 Tomcat 都直接集成进来了。但是 Jetty 的应变更加快速，这一方面是因为 Jetty 的开发社区更加活跃，另一方面也是因为 Jetty 的修改更加简单，它只要把相应的组件替换就好了，而 Tomcat 的整体结构上要复杂很多，修改功能比较缓慢。所以 Tomcat 对最新的 Servlet 规范的支持总是要比人们预期的要晚。

* + 1. **实验六：tomcat和jetty比较 RI**

在同样硬件环境和JVM配置情况下，对RI在Tomcat和Jetty环境下进行测试。（RI重构了，测试数据不与前面实验比较。）

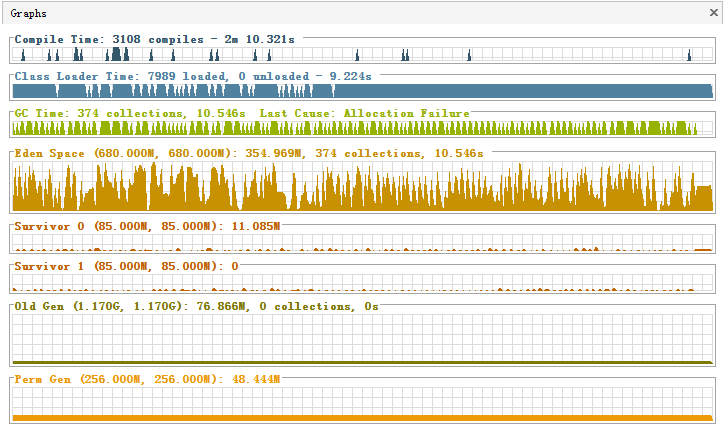
JAVA\_MEM\_OPTS=" -server -Xmx2g -Xms2g **–Xmn850m** -XX:PermSize=256m -Xss256k -XX:SurvivorRatio=8 -XX:MaxTenuringThreshold=7 -XX:GCTimeRatio=19  -Xnoclassgc -XX:+UseParNewGC -XX:+DisableExplicitGC -XX:+UseConcMarkSweepGC -XX:+CMSParallelRemarkEnabled -XX:+UseCMSCompactAtFullCollection -XX:CMSFullGCsBeforeCompaction=0 -XX:+CMSClassUnloadingEnabled -XX:CMSInitiatingOccupancyFraction=70 -XX:SoftRefLRUPolicyMSPerMB=0 -XX:+UseFastAccessorMethods -XX:+UseCMSInitiatingOccupancyOnly -XX:+UseCompressedOops -XX:-HeapDumpOnOutOfMemoryError"

**从以下运行结果看，tomcat和jetty在200个用户并行操作时，性能没有明显区别。但随着并发数的增多，400用户并发运行时，jetty事务处理速度要优于tomcat。**

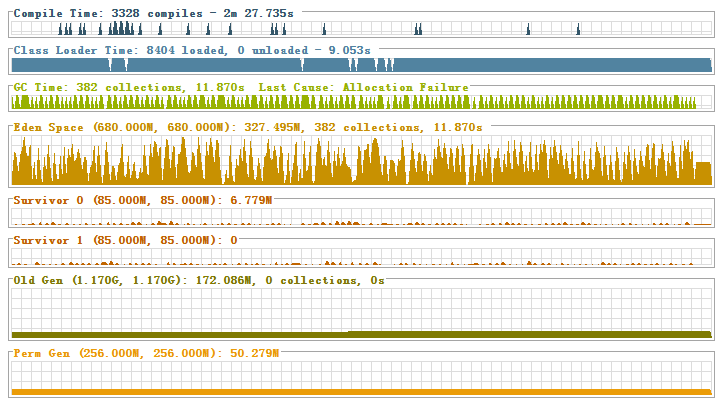
**从长时间运行（17h）运行JVM监控图来看，jetty长时间运行时，内存开销稳定性要差于tomcat。**

业务运行10min 指标

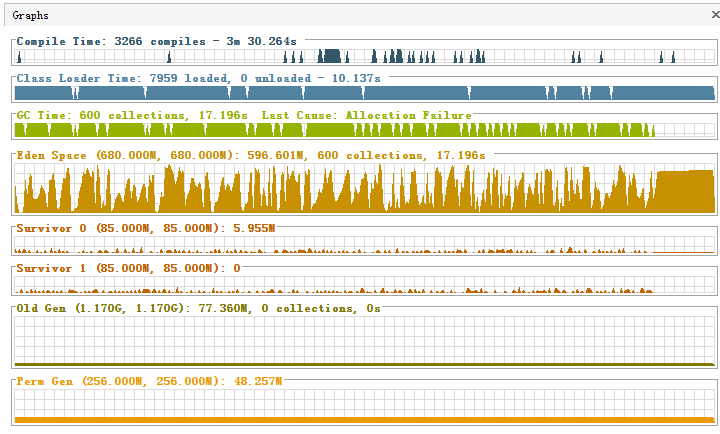
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **tomcat** | **jetty** | **tomcat2** | **jetty3** |
| 并发用户 | 200 | 200 | 400 | 400 |
| [Average Hits per Second](file:///C:\Users\ivy.fan\AppData\Local\Temp\AppData\Local\Temp\HitsperSecond) | 414 | 416 | 501 | 535 |
| 事务运行平均时间 |  |  |  |  |
| [group\_subcriber\_list](file:///C:\Users\ivy.fan\AppData\Local\Temp\AppData\Local\Temp\ResponseTime0000(group_subcriber_list)0000)(s) | 0.333 | 0.323 | 1.221 | 0.976 |
| [login](file:///C:\Users\ivy.fan\AppData\Local\Temp\AppData\Local\Temp\ResponseTime0000(login)0000)(s) | 0.287 | 0.313 | 1.085 | 0.768 |
| [subgroup\_add](file:///C:\Users\ivy.fan\AppData\Local\Temp\AppData\Local\Temp\ResponseTime0000(subgroup_add)0000)(s) | 0.052 | 0.053 | 0.14 | 0.119 |
| [subgroup\_del](file:///C:\Users\ivy.fan\AppData\Local\Temp\AppData\Local\Temp\ResponseTime0000(subgroup_del)0000)(s) | 0.527 | 0.487 | 1.715 | 1.547 |
| [subgroup\_edit](file:///C:\Users\ivy.fan\AppData\Local\Temp\AppData\Local\Temp\ResponseTime0000(subgroup_edit)0000)(s) | 0.111 | 0.118 | 0.402 | 0.245 |
| [subgroup\_list](file:///C:\Users\ivy.fan\AppData\Local\Temp\AppData\Local\Temp\ResponseTime0000(subgroup_list)0000)(s) | 0.568 | 0.563 | 1.499 | 1.487 |



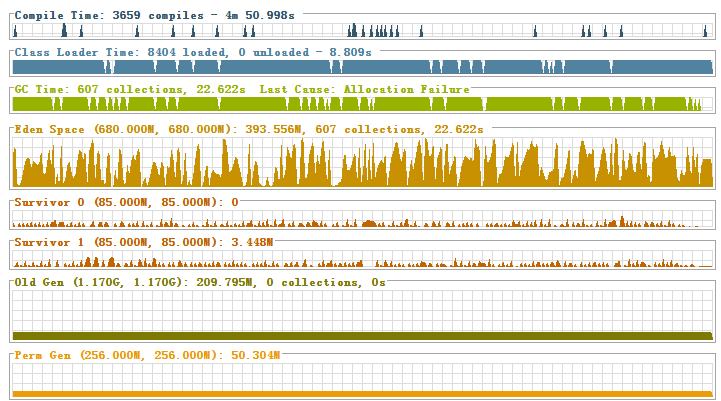
Tomcat 200用户运行10min



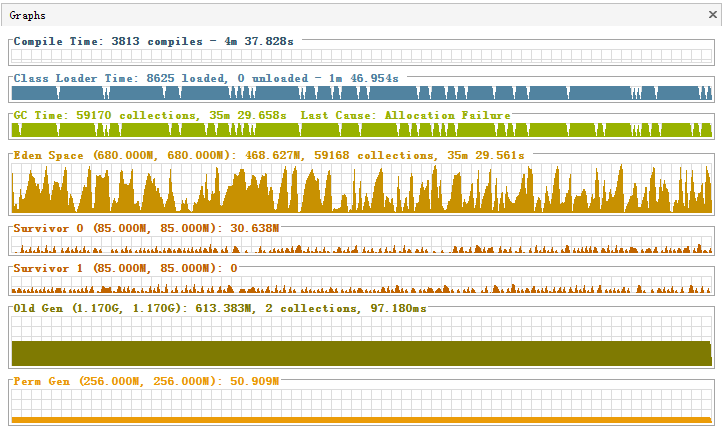
Jetty 200用户运行10min



Tomcat 400用户运行10min



Jetty 400用户运行10min



Jetty 400用户运行17h