性能瓶颈分析及案例总结

Testfan-北河

文件版本：	V0.1	文件编号：	
发布日期：		编制：	
审核：		批准：	

修订记录：

版本号	修订人	修订日期	修订描述

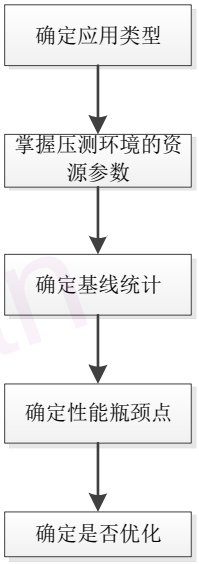
目录

1 文档说明

1.1 目的

提升学员性能问题分析能力

2 性能分析步骤



2.1.1 确定应用类型

应用大致分为 cpu 密集型和 IO 密集型。

Cpu 密集型：就是一个批量处理 CPU 请求、逻辑判断以及数学计算的过程，反应出来的现象是高负荷的 CPU 占用，load 值过高。典型的是 web server。

IO 密集型：一般都是高负荷的内存使用以及存储系统，此类应用通常使用 CPU 资源都是为了产生 IO 请求以及进入到内核调度的 sleep 状态。反映出来的现象是 CPU 在等 I/O (硬盘/内存) 的读/写，此时 CPU Loading 不高。典型的是：mysql、oracle 等

2.1.2 掌握压测环境的资源参数

包括需要了解用到的中间件相关的版本、参数配置情况及服务器 cpu、网卡、内存等硬件性能指标。

2.1.3 确定基线统计

明确单服务情况下的大概的性能情况，如单 helloword 请求在虚拟机下 tps 至少 1w，单 redis 在物理机下的单线程跑满，set 的 tps 应该至少在 8W，mysql 小字段类型的 insert 的 tps 至少在 2w。

在有基础性能的指标后，再结合压测的结果和情况，以明确性能瓶颈点大概出现在哪个节点。

2.1.4 确定性能的瓶颈点。

通过对基线的统计和资源监控情况的分析，确定 cpu、网卡、IO 等是否出现资源瓶颈。

2.1.5 确定是否优化

如果问题出现在涉及操作系统、中间件、JDK、驱动等核心工作机制，这些是不能进行代码修改的，只能考虑进行组件的升级。

如果涉及的是应用程序的代码层，则可以考虑进行以下优化：

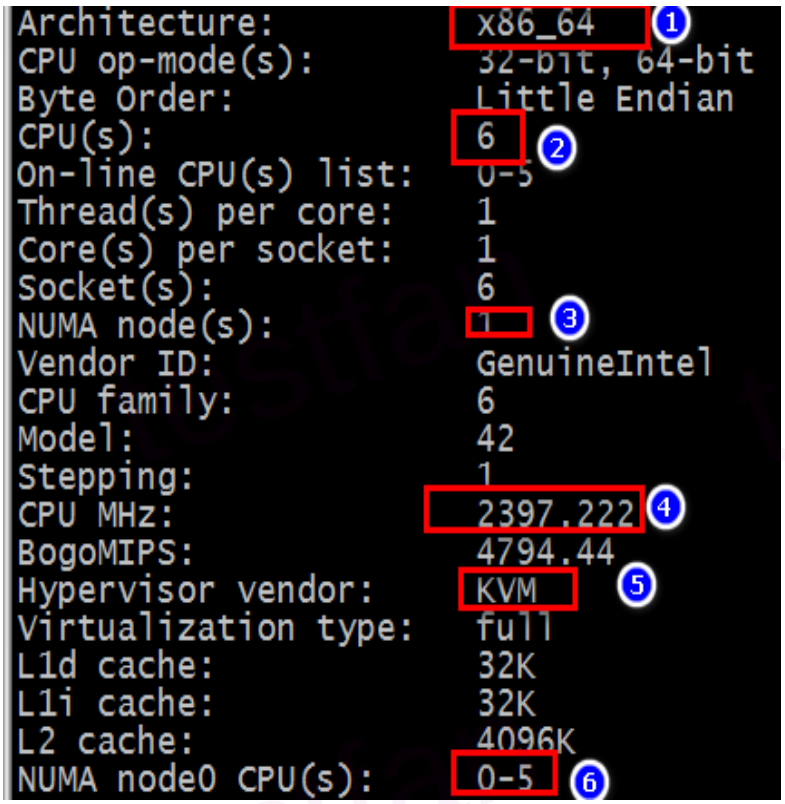
1) 使用对象池减少对重复对象的创建；
2) 调整对后端的连接
3) 增加本地缓存
4) 如果不涉及事务的情况下，考虑使用 Nosql 进行存储
5) 一次请求合并多次操作。
6) 由串行修改为并行操作
7) 同步修改为异步

3 性能分析命令

3.1 Cpu

3.1.1 了解 cpu 基本信息

命令：lscpu



1. Cpu 架构 64 位
2. Cpu 核心数 6
3. NUMA 节点数为 2 个（显示值加 1）
4. Cpu 的核心频率
5. 说明此服务器为虚拟机
6. 此服务器的 cpu 使用的是 numa node0 的 0-5 号 cpu

3.1.2 vmstat 工具的使用

命令：vmstat 1 60> /tmp/cpu.txt

说明：每秒采样 1 次，共采集 100 次

格式化显示：cat /tmp/cpu.txt|column -t

```
[18 ~]# cat /tmp/cpu.txt|column  -t
procs  -----------memory----------  ---swap--  -----io----  --system--  -----cpu-----
r      b                            swpd       free         buff        cache          si  so  bi  bo  in   cs   us  sy  id  wa  st
0      0                            7360       24457996     404676      102122480      0   0   0   3   0    0    0   0   99  0   0
0      0                            7360       24458344     404676      102122480      0   0   0   0   276  371  0   0   100 0   0
0      0                            7360       24458344     404676      102122480      0   0   0   0   257  533  0   0   100 0   0
0      0                            7360       24458620     404676      102122480      0   0   0   0   170  300  0   0   100 0   0
0      0                            7360       24458868     404676      102122480      0   0   0   0   177  269  0   0   100 0   0
0      0                            7360       24459240     404676      102122480      0   0   0   0   174  298  0   0   100 0   0
0      0                            7360       24459240     404676      102122480      0   0   0   24  182  266  0   0   100 0   0
0      0                            7360       24457168     404676      102122480      0   0   0   0   546  459  0   0   100 0   0
0      0                            7360       24457572     404676      102122480      0   0   0   0   295  513  0   0   100 0   0
0      0                            7360       24457752     404676      102122480      0   0   0   0   166  301  0   0   100 0   0
0      0                            7360       24457876     404676      102122480      0   0   0   0   177  268  0   0   100 0   0
0      0                            7360       24458124     404676      102122480      0   0   0   0   172  309  0   0   100 0   0
0      0                            7360       24458124     404676      102122480      0   0   0   20  209  466  0   0   100 0   0
0      0                            7360       24458492     404676      102122480      0   0   0   20  224  351  0   0   100 0   0
0      0                            7360       24458616     404676      102122480      0   0   0   0   178  267  0   0   100 0   0
0      0                            7360       24459112     404676      102122480      0   0   0   0   359  463  0   0   100 0   0
0      0                            7360       24458832     404676      102122480      0   0   0   0   180  262  0   0   100 0   0
1      0                            7360       24458844     404676      102122480      0   0   0   0   1219 520  0   0   100 0   0
0      0                            7360       24458844     404676      102122480      0   0   0   0   235  317  0   0   100 0   0
```

1）procs

a.r 列表示运行和等待 CPU 时间片的进程数，这个值如果长期大于系统 CPU 个数，就说明 CPU 资源不足，可以考虑增加 CPU；

b.b 列表示在等待资源的进程数，比如正在等待 I/O 或者内存交换等。

2）memory

a.swpd 列表示切换到内存交换区的内存数量（以 KB 为单位）。如果 swpd 的值不为 0 或者比较大，而且 si、so 的值长期为 0，那么这种情况一般不用担心，不会影响系统性能；

b.free 列表示当前空闲的物理内存数量（以 KB 为单位）；

c.buff 列表示 buffers cache 的内存数量，一般对块设备的读写才需要缓冲；

d.cache 列表示 page cached 的内存数量，一般作文件系统的 cached，频繁访问的文件都会被 cached。如果 cached 值较大，就说明 cached 文件数较多。如果此时 IO 中的 bi 比较小，就说明文件系统效率比较好。

3）swap

a.si 列表示由磁盘调入内存，也就是内存进入内存交换区的数量；

b.so 列表示由内存调入磁盘，也就是内存交换区进入内存的数量

c.一般情况下，si、so 的值都为 0，如果 si、so 的值长期不为 0，则表示系统内存不足，需要考虑是否增加系统内存。

4）IO

a.bi 列表示从块设备读入的数据总量（即读磁盘，单位 KB/秒）

b.bo 列表示写入到块设备的数据总量（即写磁盘，单位 KB/秒）

这里设置的 bi+bo 参考值为 1000，如果超过 1000，而且 wa 值比较大，则表示系统磁盘 IO 性能瓶颈。（需要根据具体硬件性能评估）

5）system

a.in 列表示在某一时间间隔中观察到的每秒设备中断数；

b.cs 列表示每秒产生的上下文切换次数。

上面这两个值越大，会看到内核消耗的 CPU 时间就越多。

6）CPU

a.us 列显示了用户进程消耗 CPU 的时间百分比。

b.sy 列显示了内核进程消耗 CPU 的时间百分比。sy 的值比较高时，就说明内核消耗的 CPU 时间多；如果 us+sy 超过 80%，就说明 CPU 的资源存在不足。

c.id 列显示了 CPU 处在空闲状态的时间百分比；

d.wa 列表示 IO 等待所占的 CPU 时间百分比。wa 值越高，说明 IO 等待越严重。如果 wa 值超过 20%，说明 IO 等待严重。

e.st 列一般不关注，虚拟机占用的时间百分比。

经验总结：

- cpu 一般被充分利用的大概的范围

us：65% - 70%

sy：30% - 35%

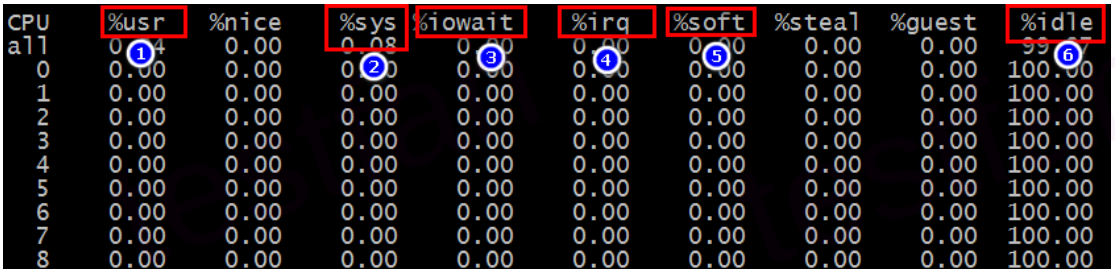
id：0% - 5%

- procs 中的 r 运行队列数不要超出每个处理器 3 个可运行状态线程的限制。
- sy 如果比较高一般是 in（中断）和 cs（上下文切换），可根据情况开启 RPS 或者减少应用的线程池。

3.1.3 mpstat 工具的使用

命令：mpstat -P ALL 1

介绍：每秒中采集一次 cpu 各个核心的使用资源情况



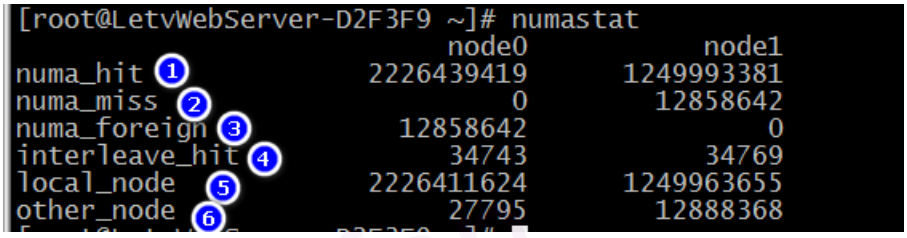
1. 表示处理用户进程所使用 CPU 的百分比
2. 表示内核进程使用的 CPU 百分比
3. 表示等待进行 I/O 所使用的 CPU 时间百分比
4. 表示用于硬件中断的 CPU 百分比。（硬中断是外部设备引起的中断）
5. 表示用于软件中断的 CPU 百分比。（软中断是操作系统执行中断指令引起的）
6. 显示 CPU 的空闲时间百分比。

经验总结：

- 如果监控中的个别 cpu，%soft 很高，%idle 很低，需要考虑软中断优化，软中断的优化必须基于物理机进行，且中断频率很高，基于小数据包的处理。
- 单队列网卡多 cpu 环境下，适合开启 RPS 和 RSS；多队列网卡多 cpu 环境，则可使用 SMP irq affinity 直接绑定硬中断。（见网卡优化）

3.1.4 numastat 工具的使用

命令：numastat



1. numa_hit 是打算在该节点上分配内存，最后从这个节点分配的次数;

2. **num_miss 是打算在该节点分配内存，最后却从其他节点分配的次数;**
3. num_foregin 是打算在其他节点分配内存，最后却从这个节点分配的次数;
4. interleave_hit 是采用 interleave 策略最后从该节点分配的次数;
5. local_node 该节点上的进程在该节点上分配的次数
6. other_node 是其他节点进程在该节点上分配的次数

经验总结：

- num_miss 是比较关键的参数如果此值过高的话，说明单个 node 的内存命中率比较低，此时可能需要使用绑定 cpu 的策略;
- 使用参数–p 可以查看单个进程在不同 node 下，内存的分布情况。
- 使用监控命令 watch -n1 --differences numastat，监控 miss 值变化，如果太高的时候可以考虑绑定 node;

3.1.5 numactl 工具使用

命令：　　【options】

说明：控制进程或内存指令

经验总结：

- 一般我们可以启动 numad 使系统自动监控 cpu、内存的分配情况，并平衡这些资源的访问，但有时候我们需要使用–cpunodebind（node 绑定）physcpubind（cpu 绑定）。

3.1.6 taskset 工具使用

命令：taskset -cp【cpulist】【pid】

说明:通用绑定 cpu 命令,对 numa 架构的建议使用 numa 自身的绑定命令处理。

举例：taskset -cp 1,2　25718。将 pid 为 25718 绑定在 1 号和 2 号 cpu 上

经验总结：

- 为了完成 cpu 的隔离应该将此 cpu 的中断绑定在非隔离的 cpu 上。

```
echo CPU_MASK > /proc/irq/<irq number>/smp_affinity
```

3.2 网卡

3.2.1 了解网卡基本信息

命令：ethtool eth0

```
Supported ports: [ TP ]
Supported link modes:   100baseT/Full
                        1000baseT/Full
                        10000baseT/Full
Supported pause frame use: No
Supports auto-negotiation: Yes
Advertised link modes:  100baseT/Full
                        1000baseT/Full
                        10000baseT/Full
Advertised pause frame use: Symmetric
Advertised auto-negotiation: Yes
Speed: 10000Mb/s
Duplex: Full
Port: Twisted Pair
PHYAD: 0
Transceiver: external
Auto-negotiation: on
MDI-X: Unknown
Supports Wake-on: umbg
Wake-on: g
Current message level: 0x00000007 (7)
                       drv probe link
Link detected: yes
```

1. 网卡带宽,显示万兆网卡。10000Mb/s=1250MB/s
2. 显示全双工通信.

3.2.2 网卡优化

网卡优化需要结合 cpu 资源使用情况进行调整，网卡的优化主要是对软中断处理的优化，虚拟机下不能对网卡进行优化。

- 命令：grep eth0 /proc/interrupts |awk '{print $1,$NF}'

  说明：显示网卡的队列个数

```
[root@tomcat45_87 ~]#  grep eth0 /proc/interrupts |awk '{print $1,$NF}'
113: eth0
114: eth0-TxRx-0
```

显示为网卡的队列个数是 2.网卡为单队列网卡。114 为对应的 irq 编号

经验总结：

1) 单队列网卡多 cpu 环境下，适合开启 RPS 和 RSS；

运行 sh set_irq.sh (见附件)，如果撤销则运行 sh unset_irq.sh(见附件)

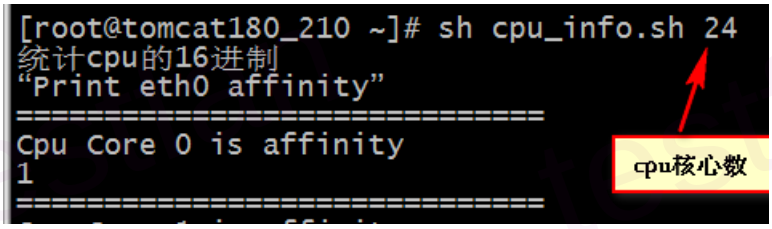
2) 多队列网卡多 cpu 环境，则可使用 SMP irq affinity 直接绑定硬中断。

注意：绑定 cpu 亲和性前需先备份对应网卡的 irq 数据文件，以备恢复使用。

i. 首先关闭 irqbalance 服务 service irqbalance stop（一定要 stop）

```
[root@tomcat180_210 ~]# service irqbalance stop
Stopping irqbalance:                                       [  OK  ]
[root@tomcat180_210 ~]#
```

ii. 获取 cpu 的 16 进制表达式（shcpu_Info.run 24）



iii. 绑定 cpu 与 irq

- 使用 16 进制设置 smp_affinity

echo "1" > /proc/irq/144/smp_affinity；“1”数字代表 cpu 的 16 进制编号

……

- 使用 10 进制设置 smp_affinity_list(推荐，相比较「smp_affinity」的十六进制，可读性更好些)，以下是多个 CPU 参与中断处理设置方式。

echo 3,5 > /proc/irq/144/smp_affinity_list；数字代表 cpu 的编号。

echo 0-7 > /proc/irq/144/smp_affinity_list

3.3 网络分析

3.3.1 fping 工具使用

命令：fping -e -c 30 -f servers.txt

说明：同时批量 ping 多个文件 servers.txt 中的 IP 地址，获取每个服务器的响应时间和成功失败情况。

```
[root@resin150_65 ~]#  fping  -e -c 30 -f servers.txt
10.150.150.66  : [0], 96 bytes, 0.17 ms (0.17 avg, 0% loss)
10.150.150.66  : [1], 96 bytes, 0.16 ms (0.16 avg, 0% loss)
10.150.150.66  : [2], 96 bytes, 0.17 ms (0.16 avg, 0% loss)
10.150.150.66  : [3], 96 bytes, 0.16 ms (0.16 avg, 0% loss)
ICMP Host Unreachable from 10.150.150.65 for ICMP Echo sent to 10.150.190.178
ICMP Host Unreachable from 10.150.150.65 for ICMP Echo sent to 10.150.190.178
ICMP Host Unreachable from 10.150.150.65 for ICMP Echo sent to 10.150.190.178
```

3.3.2 tcprstat 工具使用

命令：./tcprstat -p 20888 -t 1 -n 0

说明：此工具是通过抓包统计分析请求的响应时间。

```
[root@resin150_65 ~]# ./tcprstat -p 20888 -t 1 -n 0
timestamp       count   max     min     avg     med     stddev  95_max  95_avg  95_std  99_max  99_avg  99_std
1471955037      0       0       0       0       0       0       0       0       0       0       0       0
1471955038      0       0       0       0       0       0       0       0       0       0       0       0
1471955039      0       0       0       0       0       0       0       0       0       0       0       0
1471955040      0       0       0       0       0       0       0       0       0       0       0       0
1471955041      0       0       0       0       0       0       0       0       0       0       0       0
1471955042      0       0       0       0       0       0       0       0       0       0       0       0
1471955043      0       0       0       0       0       0       0       0       0       0       0       0
1471955044      0       0       0       0       0       0       0       0       0       0       0       0
1471955045      0       0       0       0       0       0       0       0       0       0       0       0
1471955046      0       0       0       0       0       0       0       0       0       0       0       0
1471955047      0       0       0       0       0       0       0       0       0       0       0       0
1471955048      0       0       0       0       0       0       0       0       0       0       0       0
1471955049      0       0       0       0       0       0       0       0       0       0       0       0
1471955050      0       0       0       0       0       0       0       0       0       0       0       0
1471955051      0       0       0       0       0       0       0       0       0       0       0       0
1471955052      2       162     120     141     162     21      120     120     0       120     120     0
1471955053      0       0       0       0       0       0       0       0       0       0       0       0
```

说明：时间单位是微秒

注意：-p：为端口号

3.3.1 nicstat 工具使用

命令：nicstat -t 1

说明：每秒采集一次网卡 tcp 连接信息，以 M 为单位显示。

```
[root@resin150_65 ~]# nicstat -t 1
20:49:51    InKB   OutKB   InSeg  OutSeg Reset  AttF %ReTX InConn OutCon Drops
TCP         0.00    0.00   815.5  2927.8  13.2  2.28 0.000   3.32   4.48  0.09
20:49:52    InKB   OutKB   InSeg  OutSeg Reset  AttF %ReTX InConn OutCon Drops
TCP         0.00    0.00    2.00    2.00  0.00  0.00 0.000   0.00   0.00  0.00
20:49:53    InKB   OutKB   InSeg  OutSeg Reset  AttF %ReTX InConn OutCon Drops
TCP         0.00    0.00    2.00    2.00  0.00  0.00 0.000   0.00   0.00  0.00
```

InKB：表示每秒接收到的千字节.

OutKB：表示每秒传输的千字节.

InSeg：表示每秒接收到的 TCP 数据段(TCP Segments).

OutSeg：表示每秒传输的 TCP 数据段(TCP Segments).

Reset：表示 TCP 连接从 ESTABLISHED 或 CLOSE-WAIT 状态直接转变为 CLOSED 状态的次数.

AttF ：表示 TCP 连接从 SYN-SENT 或 SYN-RCVD 状态直接转变为 CLOSED 状态的次数,再加上 TCP 连接从 SYN-RCVD 状态直接转变为 LISTEN 状态的次数

%ReTX：表示 TCP 数据段(TCP Segments)重传的百分比.即传输的 TCP 数据段包含有一个或多个之前传输的八位字节.

InConn：表示 TCP 连接从 LISTEN 状态直接转变为 SYN-RCVD 状态的次数.

OutCon：表示 TCP 连接从 CLOSED 状态直接转变为 SYN-SENT 状态的次数.

Drops：表示从完成连接(completed connection)的队列和未完成连接(incomplete connection)的队列中丢弃的连接次数.

命令：nicstat -x –n 1

说明：每秒采集一次网卡扩展信息情况，包括网卡的利用率、包量、错误率等信息。

```
^C
[root@resin150_65 ~]# nicstat -x -n 1
10:57:31      RdKB    WrKB   RdPkt   WrPkt   IErr  OErr  Coll  NoCP Defer  %Util
eth0         543.5   335.0  1239.0  2636.1  0.00  0.00  0.00  0.00  0.00   0.72
eth0         56.31   31.86   787.4   396.7  0.00  0.00  0.00  0.00  0.00   0.07
eth0         60.78   37.13   863.7   468.4  0.00  0.00  0.00  0.00  0.00   0.08
eth0         88.36   47.65  1221.2   588.6  0.00  0.00  0.00  0.00  0.00   0.11
eth0         81.87   55.57  1121.8   690.3  0.00  0.00  0.00  0.00  0.00   0.11
eth0         69.64   39.29   973.6   491.3  0.00  0.00  0.00  0.00  0.00   0.09
eth0         67.22   38.48   954.6   487.8  0.00  0.00  0.00  0.00  0.00   0.09
eth0         72.89   42.76  1009.9   524.0  0.00  0.00  0.00  0.00  0.00   0.09
eth0         50.97   31.83   722.0   402.6  0.00  0.00  0.00  0.00  0.00   0.07
eth0         75.38   40.70  1056.0   507.5  0.00  0.00  0.00  0.00  0.00   0.10
eth0         60.47   37.24   864.1   474.5  0.00  0.00  0.00  0.00  0.00   0.08
eth0         38.30   19.84   536.6   246.3  0.00  0.00  0.00  0.00  0.00   0.05
eth0         65.26   38.08   932.3   480.2  0.00  0.00  0.00  0.00  0.00   0.08
eth0         61.48   34.31   861.1   429.0  0.00  0.00  0.00  0.00  0.00   0.08
eth0         46.94   27.70   672.4   352.7  0.00  0.00  0.00  0.00  0.00   0.06
eth0         73.51   39.81  1021.4   492.7  0.00  0.00  0.00  0.00  0.00   0.09
eth0         72.85   43.41  1035.7   550.3  0.00  0.00  0.00  0.00  0.00   0.10
eth0         42.81   21.60   606.7   265.3  0.00  0.00  0.00  0.00  0.00   0.05
eth0         68.75   39.07   982.2   489.6  0.00  0.00  0.00  0.00  0.00   0.09
eth0         63.75   35.01   894.0   437.5  0.00  0.00  0.00  0.00  0.00   0.08
10:57:51      RdKB    WrKB   RdPkt   WrPkt   IErr  OErr  Coll  NoCP Defer  %Util
```

3.4 内存

3.4.1 了解内存基本信息

命令：free -g

说明：以 GB 为单位显示内存的使用情况

```
[root@LetvWebServer-D68741 ~]# free -g
             total       used       free     shared    buffers     cached
Mem:           125 (1)      65 (2)      60 (3)          0          0         13
-/+ buffers/cache:         51         74
Swap:            5          0          5
```

1. 总物理内存
2. 已使用内存
3. 剩余内存

3.4.2 vmstat 工具的使用

命令：vmstat 1 60> /tmp/cpu.txt

说明：每秒采样 1 次，共采集 100 次

格式化显示：cat /tmp/cpu.txt|column -t

```
18 ~]# cat /tmp/cpu.txt|column -t
procs  -----------memory----------  ---swap--  -----io----  --system--  -----cpu-----
r      b                            swpd       free         buff        cache        si  so  bi  bo  in   cs   us  sy  id  wa  st
0      0                            7360       24457996     404676      102122480    0   0   0   3   0    0    0   0   99  0   0
0      0                            7360       24458344     404676      102122480    0   0   0   0   276  371  0   0   100 0   0
0      0                            7360       24458344     404676      102122480    0   0   0   0   257  533  0   0   100 0   0
0      0                            7360       24458620     404676      102122480    0   0   0   0   170  300  0   0   100 0   0
0      0                            7360       24458868     404676      102122480    0   0   0   0   177  269  0   0   100 0   0
0      0                            7360       24459240     404676      102122480    0   0   0   0   174  298  0   0   100 0   0
0      0                            7360       24459240     404676      102122480    0   0   0   24  182  266  0   0   100 0   0
0      0                            7360       24457168     404676      102122480    0   0   0   0   546  459  0   0   100 0   0
0      0                            7360       24457572     404676      102122480    0   0   0   0   295  513  0   0   100 0   0
0      0                            7360       24457752     404676      102122480    0   0   0   0   166  301  0   0   100 0   0
0      0                            7360       24457876     404676      102122480    0   0   0   0   177  268  0   0   100 0   0
0      0                            7360       24458124     404676      102122480    0   0   0   0   172  309  0   0   100 0   0
0      0                            7360       24458124     404676      102122480    0   0   0   20  209  466  0   0   100 0   0
0      0                            7360       24458492     404676      102122480    0   0   0   20  224  351  0   0   100 0   0
0      0                            7360       24458616     404676      102122480    0   0   0   0   178  267  0   0   100 0   0
0      0                            7360       24459112     404676      102122480    0   0   0   0   359  463  0   0   100 0   0
0      0                            7360       24458832     404676      102122480    0   0   0   0   180  262  0   0   100 0   0
1      0                            7360       24458844     404676      102122480    0   0   0   0   1219 520  0   0   100 0   0
0      0                            7360       24458844     404676      102122480    0   0   0   0   235  317  0   0   100 0   0
```

1）memory

a.swpd 列表示切换到内存交换区的内存数量（以 KB 为单位）。如果 swpd 的值不为 0 或者比较大，而且 si、so 的值长期为 0，那么这种情况一般不用担心，不会影响系统性能；

b.free 列表示当前空闲的物理内存数量（以 KB 为单位）；

c.buff 列表示 buffers cache 的内存数量，一般对块设备的读写才需要缓冲；

d.cache 列表示 page cached 的内存数量，一般作文件系统的 cached，频繁访问的文件都会被 cached。如果 cached 值较大，就说明 cached 文件数较多。如果此时 IO 中的 bi 比较小，就说明文件系统效率比较好。

3）swap

a.si 列表示由磁盘调入内存，也就是内存进入内存交换区的数量；

b.so 列表示由内存调入磁盘，也就是内存交换区进入内存的数量

c.一般情况下，si、so 的值都为 0，如果 si、so 的值长期不为 0，则表示系统内存不足，需要考虑是否增加系统内存。

经验总结：

- 一般不出现 swap 内存是足够的。如果内存不足我们可以运行【top】输入 shift+m，将进程按照物理内存占用（“RES”列）从大到小进行排序，然后对排前面的进程逐一排查。

3.5 磁盘

3.5.1 iostat 磁盘监控

命令： iostat -x -k -d 1

说明：每秒以 kB 为单位，采样各个磁盘的 io 详细情况；

```
[root@LetvWebServer-8CFC66 ~]# iostat -x -k -d 1
Linux 2.6.32-926.504.30.3.letv.el6.x86_64 (LetvWebServer-8CFC66)        07/13/2016      _x86_64_        (24 CPU)

Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avgqu-sz   await  svctm  %util
sda               0.00     0.87    0.00    1.03     0.09    65.50   126.39     0.00    1.33   0.33   0.03
dm-0              0.00     0.00    0.00    1.00     0.05     3.98     8.07     0.00    0.44   0.22   0.02
dm-1              0.00     0.00    0.00    0.74     0.00     2.96     8.01     0.00    0.17   0.07   0.01
dm-2              0.00     0.00    0.00    0.16     0.04    58.55   722.68     0.00    6.60   0.40   0.01
sdb               0.00     0.00    0.00    0.00     0.00     0.09   450.12     0.00    0.38   0.38   0.00
```

rrqm/s: 每秒对该设备的读请求被合并次数，文件系统会对读取同块(block)的请求进行合并

wrqm/s: 每秒对该设备的写请求被合并次数

r/s: 每秒完成的读次数

w/s: 每秒完成的写次数

rkB/s: 每秒读数据量(kB 为单位)

wkB/s: 每秒写数据量(kB 为单位)

avgrq-sz:平均每次 IO 操作的数据量(扇区数为单位)

avgqu-sz: 平均等待处理的 IO 请求队列长度

await: 平均每次 IO 请求等待时间(包括等待时间和处理时间，毫秒为单位)

svctm: 平均每次 IO 请求的处理时间(毫秒为单位)

%util: 1 秒中有百分之多少的时间用于 I/O 操作。

经验总结：

- **提高 IO 效率原则：顺序写，随机读。**
- **监控时重点监控 rkB/s 和 wkB/s**
- **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、**
- **当 await 与 svctm 相差很大的时候，我们就要注意磁盘的 IO 性能**
- **吞吐量/读或写次数>32k，可以判断为顺序 IO**

3.5.1 ioprofile 工具使用

命令： pt-ioprofile -p pid [-c=times| sizes]

说明：pt-ioprofile 是 percona-toolkit 子集工具，它的原理是对某个 pid 附加一个 strace 进程进行 IO 分析，提供了直观的量化的数据来描述进程对 io 设备的真实读写量，默认是按 IO 占用时间比重排序。

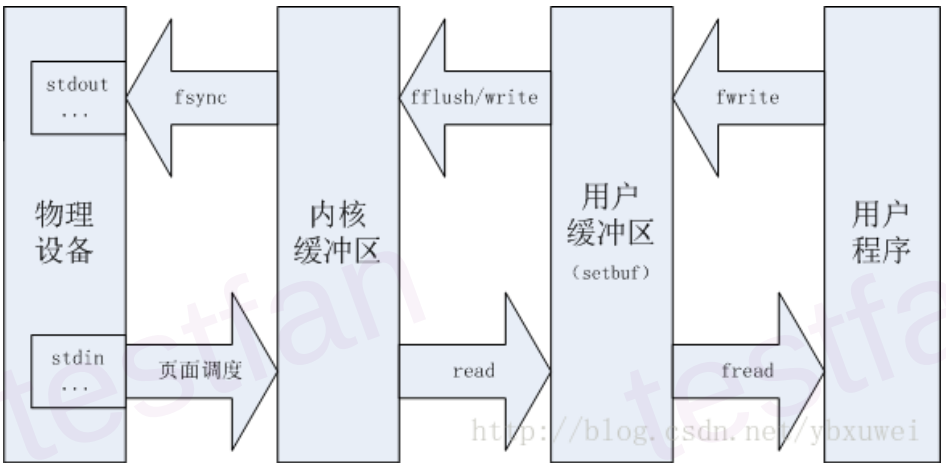
```
[root@nginx180_211 bin]# ./pt-ioprofile  --profile-pid  6804
Wed Aug 24 20:54:29 CST 2016
Tracing process ID 6804
     total      pwrite       write       fsync filename
  4.375978    4.272977    0.000000    0.103001 /letv/zx_test_mysql/data/ib_logfile0
  1.430520    0.000000    1.430520    0.000000 /letv/zx_test_mysql/data/zx_test-bin.000337
  0.697903    0.373180    0.000000    0.324723 /letv/zx_test_mysql/data/order_trade/orders.ibd
  0.487648    0.202927    0.000000    0.284721 /letv/zx_test_mysql/data/ibdata1
  0.077693    0.034989    0.000000    0.042704 /letv/zx_test_mysql/data/order_trade/order_logs.ibd
```

增加-c=sizes 该参数将结果已 B/s 的方式展示出来

```
[root@nginx180_211 bin]# ./pt-ioprofile  -p  6804 -c=sizes
Wed Aug 24 21:14:39 CST 2016
Tracing process ID 6804
     total      read      pwrite       write  fdatasync      fsync      open     close     lseek filename
 230834176         0   230834176           0          0          0         0         0         0 /letv/zx_test_mysql/data/order_trade/orders.ibd
 200572928         0   200572928           0          0          0         0         0         0 /letv/zx_test_mysql/data/ibdata1
  51540480         0    51540480           0          0          0         0         0         0 /letv/zx_test_mysql/data/ib_logfile1
  46734848         0    46734848           0          0          0         0         0         0 /letv/zx_test_mysql/data/ib_logfile0
  39983736         0           0    39983736          0          0         0         0         0 /letv/zx_test_mysql/data/zx_test-bin.000346
  22986752         0    22986752           0          0          0         0         0         0 /letv/zx_test_mysql/data/order_trade/order_logs.ibd
  14993442         0           1    14993441          0          0         0         0         0 /letv/zx_test_mysql/data/zx_test-bin.000345
      8610      2877           0          21          0          0         0         0      5712 /letv/zx_test_mysql/data/zx_test-bin.index
        21         0           0          21          0          0         0         0         0 /letv/zx_test_mysql/data/zx_test-bin.~rec~
         0         0           0           0          0          0         0         0         0 /letv/zx_test_mysql/data/
```

经验总结：

- **分析 IO 系统瓶颈前首先需要了解文件读写磁盘的原理**



read/write/fsync 涉及到进程上下文的切换，即用户态到核心态的转换，这是个比较消耗性能的操作，如果这个值太大说明存在读写瓶颈。

需要注意的是对于输出设备或磁盘文件，fflush 只能保证数据到达内核缓冲区，并不能保证数据到达物理设备，因此应该在调用 fflush 后，调用 fsync(fileno(stream))，确保数据存入磁盘。

3.6 连接

命令：ss -a|grep :16001

说明：分组统计 16001 端口不同连接状态的

```
[root@LetvWebServer-E32C18 sbin]# ss -a|grep  :16001
ESTAB      0      0          10.149.11.221:55606        10.140.45.107:16001
ESTAB      0      0          10.149.11.221:34445        10.140.45.108:16001
```

ESTAB：为连接状态

经验总结：

- 如果客户端访问服务端的端口一直保持不变，如上图中的 55606 端口，此连接为长连接否则为短连接
- 连接数的多少（Cons），取决于服务端所能提供的最大的连接数（TotalCons）及集群的数量（Cnt），一般的 Cons*Cnt<=TotalCons
- 如果后端服务能力很强如 memcached，尽可能使用单一连接；如果后端服务处理时间很长，需要扩大连接池中的连接数量。

3.7 Jvm 监控

3.7.1 Jmap 堆内存分配

命令：/usr/java/jdk1.7.0_76/bin/jmap –heap【进程 pid】

```
Heap Configuration:
   MinHeapFreeRatio = 40
   MaxHeapFreeRatio = 70
1  MaxHeapSize      = 85899345920 (81920.0MB)
2  NewSize          = 21474836480 (20480.0MB)
   MaxNewSize       = 21474836480 (20480.0MB)
   OldSize          = 5439488 (5.1875MB)
   NewRatio         = 2
   SurvivorRatio    = 8
3  PermSize         = 1342177280 (1280.0MB)
   MaxPermSize      = 2684354560 (2560.0MB)
   G1HeapRegionSize = 0 (0.0MB)

Heap Usage:
New Generation (Eden + 1 Survivor Space):
   capacity = 19327352832 (18432.0MB)
   used     = 14204437832 (13546.407539367676MB)
   free     = 5122915000 (4885.592460632324MB)
   73.49396451479859% used
Eden Space:
   capacity = 17179869184 (16384.0MB)
   used     = 14201204712 (13543.324195861816MB)
   free     = 2978664472 (2840.6758041381836MB)
   82.66189084388316% used
From Space:
   capacity = 2147483648 (2048.0MB)
   used     = 3233120 (3.083343505859375MB)
   free     = 2144250528 (2044.9166564941406MB)
   0.1505538821220398% used
To Space:
   capacity = 2147483648 (2048.0MB)
   used     = 0 (0.0MB)
   free     = 2147483648 (2048.0MB)
   0.0% used
concurrent mark-sweep generation:4
   capacity = 64424509440 (61440.0MB)
   used     = 220889568 (210.65670776367188MB)
   free     = 64203619872 (61229.34329223633MB)
   0.3428657352924347% used
Perm Generation:
   capacity = 1342177280 (1280.0MB)
```

1. 整个堆内存大小
2. 新生代内存大小
3. 永久代内存大小
4. 年老代内存大小

经验总结：

- HeapSize=Yung+Old,不包括 prem。
- （单服务器单应用）最大堆的设置建议在物理内存的 1/2 到 2/3 之间，例如 16G 的物理内存的话，最大堆的设置应该在 8000M-10000M 之间，Java 进程消耗的总内存肯定大于最大堆设置的内存：堆内存（Xmx）+ 方法区内存（MaxPermSize）+ 栈内存（Xss,包括虚拟机栈和本地方法栈）*线程数 + NIO direct memory + socket 缓存区（receive37KB，send25KB）+ JNI 代码 + 虚拟机和 GC 本身 = java 的内存。

3.7.2 Jstack 堆栈分析

命令：/usr/java/jdk1.7.0_76/bin/jstack【进程 pid】>/tmp/of.txt

说明：将某个进程的 java 程序的执行堆栈捕捉到特定文件中

```
[root@LetvWebServer-8CFC66 ~]# /usr/java/jdk1.7.0_76/bin/jstack 13929  >/tmp/of.txt
```

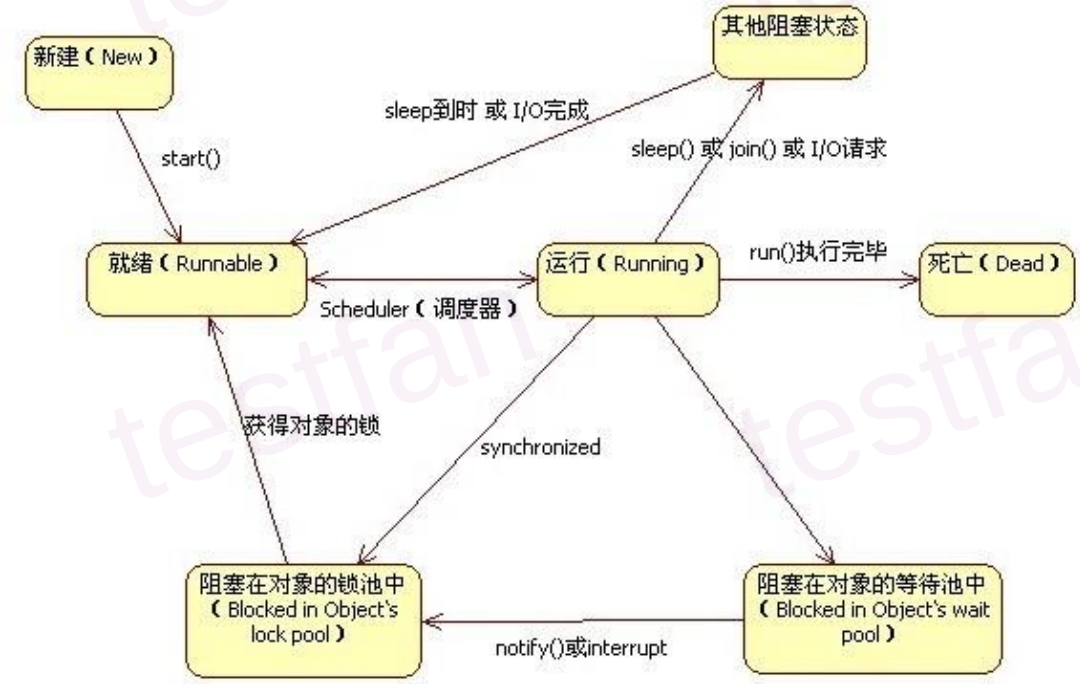
```
[root@LetvWebServer-8CFC66 ~]# more /tmp/of.txt
2016-07-12 21:11:09
Full thread dump Java HotSpot(TM) 64-Bit Server VM (24.76-b04 mixed mode):

"Attach Listener" daemon prio=10 tid=0x00007fe63c001000 nid=0x530b waiting on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"http-nio2-8304-exec-300" daemon prio=10 tid=0x00007fe4d0007800 nid=0x391f waiting on condition [0x00007fe640863000]
   java.lang.Thread.State: WAITING (parking)
        at sun.misc.Unsafe.park(Native Method)
        - parking to wait for  <0x00007feb480006e8> (a java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject
        at java.util.concurrent.locks.LockSupport.park(LockSupport.java:186)
        at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(AbstractQueuedSynchronizer.java:2043)
        at java.util.concurrent.LinkedBlockingQueue.take(LinkedBlockingQueue.java:442)
        at org.apache.tomcat.util.threads.TaskQueue.take(TaskQueue.java:103)
        at org.apache.tomcat.util.threads.TaskQueue.take(TaskQueue.java:31)
        at java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:1068)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1130)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
        at org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run(TaskThread.java:61)
        at java.lang.Thread.run(Thread.java:745)

"http-nio2-8304-exec-295" daemon prio=10 tid=0x00007fe500007800 nid=0x3908 waiting on condition [0x00007fe640df9000]
   java.lang.Thread.State: WAITING (parking)
        at sun.misc.Unsafe.park(Native Method)
        - parking to wait for  <0x00007feb480006e8> (a java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject
        at java.util.concurrent.locks.LockSupport.park(LockSupport.java:186)
        at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(AbstractQueuedSynchronizer.java:2043)
        at java.util.concurrent.LinkedBlockingQueue.take(LinkedBlockingQueue.java:442)
        at org.apache.tomcat.util.threads.TaskQueue.take(TaskQueue.java:103)
        at org.apache.tomcat.util.threads.TaskQueue.take(TaskQueue.java:31)
        at java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:1068)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1130)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
        at org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run(TaskThread.java:61)
```

每个线程执行都有相应的状态和整个线程的执行的堆栈调用。



线程状态转换情况

经验总结：

- 对线程状态的关注，一般来说值关注我们业务代码的线程执行情况，根据堆栈信息完成代码或框架层的优化。中间件的线程状态如 **tomcat** 如果有性能问题只能考虑升级版本或协议的方式处理。
- 对于线程状态为 **BLOCKED、WAITING、TIMED_WAITING** 的要重点关注，这些状态的线程会严重影响性能，对状态为 **RUNNABLE** 的也要关注，如果产生了 **wait** 也是需要优化。

附：线程堆栈分析例子

```
1. "Timer-0" daemon prio=10tid=0xac190c00 nid=0xaef in Object.wait() [0xae77d000]
2.  java.lang.Thread.State: TIMED_WAITING (on object monitor)
3.  atjava.lang.Object.wait(Native Method)
4.  -waiting on <0xb3885f60> (a java.util.TaskQueue)     ###继续 wait
5.  atjava.util.TimerThread.mainLoop(Timer.java:509)
6  atcom.jd.shop.UserService(UserService.java:507)
7.  -locked <0xb3885f60> (a java.util.TaskQueue)        ###已经 locked
8.  atjava.util.TimerThread.run(Timer.java:462)
* 线程名称：Timer-0
* 线程类型：daemon
```

*线程状态：TIMED_WAITING

* 优先级: 10（prio=10）

* jvm 线程 id：tid=0xac190c00，jvm 内部线程的唯一标识（通过 java.lang.Thread.getId()获取，通常用自增方式实现。）

* 对应系统线程 id（NativeThread ID）：nid=0xaef，和 top 命令查看的线程 pid 对应，不过一个是 10 进制，一个是 16 进制。

（通过命令：top -H -p pid，可以查看该进程的所有线程信息）

线程分析增强工具：

命令：sh show-busy-java-threads.sh -p 8321 -c 5（ 脚本见附件）

说明：-p 为进程 id，-c 为采样次数

注意：使用前需要在/etc/profile 设置 PATH 变量。

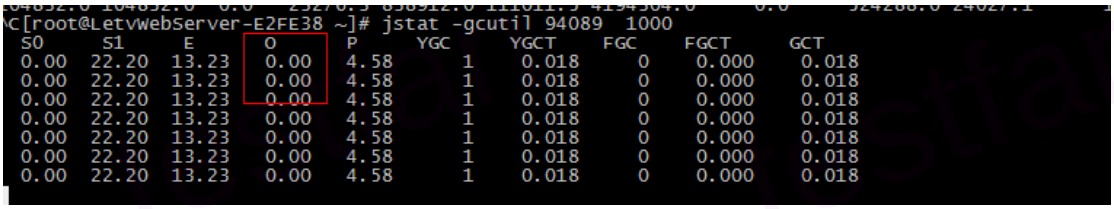
export JAVA_HOME=/usr/java/jdk1.7.0_76

export PATH=$JAVA_HOME/bin:$PATH

3.7.3 Jstat 监控 gc 情况

命令：jstat -gcutil【pid】 1000

说明：每秒采样对应进程 id 的 gc 情况



S0：年轻代中第一个 survivor（幸存区）已使用的占当前容量百分比

S1：年轻代中第二个 survivor（幸存区）已使用的占当前容量百分比

E：年轻代中 Eden（伊甸园）已使用的占当前容量百分比

O：old 代已使用的占当前容量百分比

P：perm 代已使用的占当前容量百分比

YGC：从应用程序启动到采样时年轻代中 gc 次数

YGCT：从应用程序启动到采样时年轻代中 gc 所用时间(s)

FGC：从应用程序启动到采样时 old 代(全 gc)gc 次数

FGCT：从应用程序启动到采样时 old 代(全 gc)gc 所用时间(s)

GCT：从应用程序启动到采样时 gc 用的总时间(s)

总结：

- **如果 FGC 很频繁，则应该检查年老代堆内存是否分配太小；**
- **如果 YGC 很频繁，则应该检查年轻代堆内存是否分配太小；**

- **YGCT 比较长，可以考虑是否程序中出现了大对象，增加 -XX:PretenureSizeThreshold 配置。**

3.7.4 Jvm 堆内存查看方式

3.7.4.1 jhat 内置分析工具

jhat 为 jvm 内置的对象分析工具，并提供网页形式进行浏览，此工具比较轻量级，只能简单查看对象。以下是在服务器 10.149.11.221 上分析的过程：

1. 使用 jmap 导出堆内存

jmap -dump:format=b,file=/home/server/dump.bin 110401[pid]

2. 使用 jhat 分析堆内存

jhat -J-Xmx10240M /home/server/dump.bin

3. 客户端浏览器，界面展示

http://10.149.11.221:7000/

3.7.4.2 MAT 堆内存分析工具

MAT（Memory Analysis Tools）是是一个分析 Java 堆数据的专业工具，用它可以准确定位内存泄漏的原因,推荐使用这种方式。

使用过程中需要结合 linux 和 windows 两个版本完成内存分析，具体步骤如下：

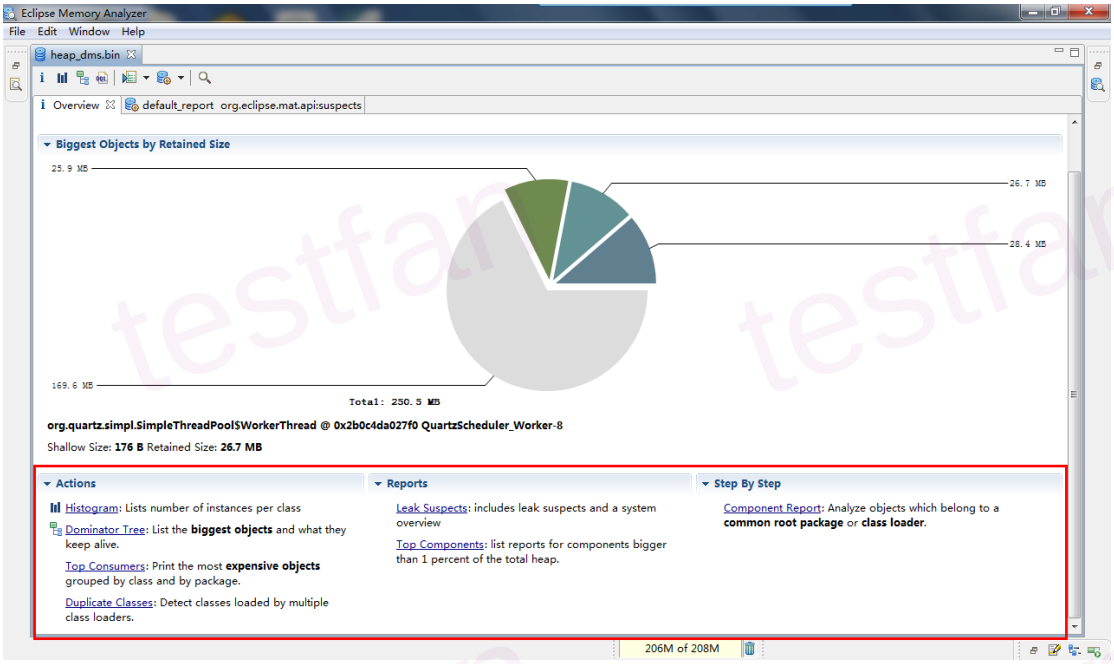
1） 使用 jmp 导出堆文件，如 jmap -dump:format=b,file=/home/server/dump.bin 29716【pid】

2） 执行./ParseHeapDump.sh /home/server/dump.bin 来分析 dump 文件，最终得到以下文件：

```
dump.a2s.index
dump.bin
dump.domIn.index
dump.domOut.index
dump.idx.index
dump.inbound.index
dump.index
dump.o2c.index
dump.o2hprof.index
dump.o2ret.index
dump.outbound.index
dump.threads
```

3） 将分析得到的文件包括原 dump 文件下载回 windows 平台，打开 MAT 工具使用菜单 File－>Open Heap Dump 打开 dump 文件即可查看到分析结果。

MAT 主要有以下功能：



1. Histogram 可以列出内存中的对象，对象的个数以及大小。

2. Dominator Tree 列出最大的对象和他们保持活跃情况。此功能比较常用，可以通过比较两次的 dump 文件，确定哪些对象导致的内存一直不能回收。

3.Top consumers 通过图形列出最大的 object。

4.Leak Suspects 通过 MA 自动分析泄漏的原因。

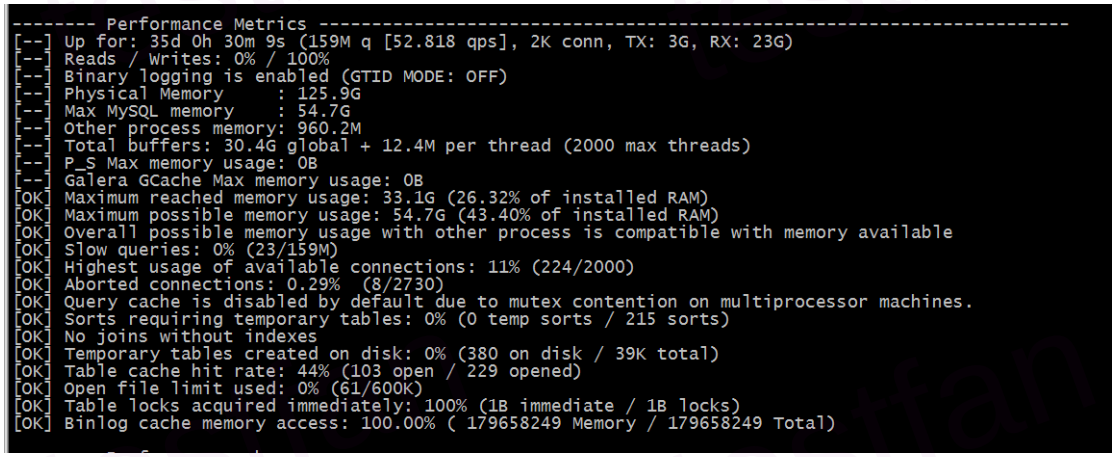
3.8 Mysql 监控

3.8.1 mysqltuner.pl 工具使用

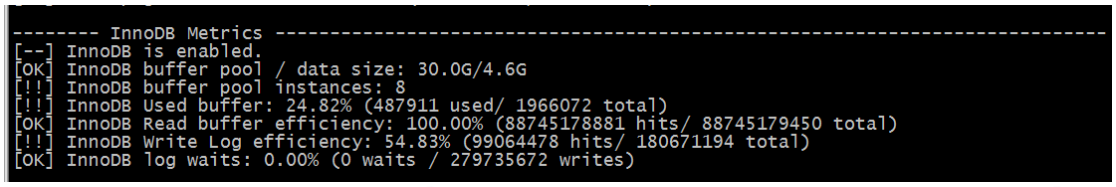
安装：wget https://raw.github.com/major/MySQLTuner-perl/master/mysqltuner.pl

设置 PATH：PATH=$PATH:/usr/local/mysql/bin

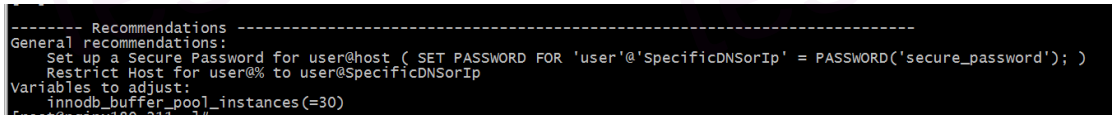
命令： ./mysqltuner.pl --socket /home/zx_test_mysql/tmp/mysql.sock



Performance Metrics：性能度量值.主要关注读写比、table cache 命中率、连接的最高使用率。



InnoDB Metrics: InnoDB 性能指标。主要关注读写 buffer 的效率。



Recommendations：此程序给的一些修改建议，大家可参考修改。

4 常见问题总结

4.1 Cpu 利用率和 load 值有无直接关系

在过去做压力测试的时候，我们经常会关注两个指标，CPU 利用率和 Load 值。一般认为 CPU 利用率和 Load 值是正比的关系，Load 值高，CPU 利用率就高。但是有时候 Load 很高，CPU 利用率却可能比较低（多核更是可能出现分配不均的情况）。其原因为 Load 是等待处理的任务队列，当你的应用在等待同步消息返回处理的同时，CPU 还是会将时间切片分配给这些线程，而真正需要 CPU 的线程，却不得不在到了时间片以后暂时放弃工作被挂起。

因此在程序设计的时候就要考虑如何利用好 CPU 的这个资源，如何均匀的将压力分摊到各个 CPU 上（有时候就一个线程在不断循环，导致单个 CPU 负荷很高）。

附：sar-q 1 获取 load 情况和运行队列情况

4.2 随机 I/O 与顺序 I/O 区分标准与优化

一般来说，如果平均 io 数据尺寸小于 32k，可认为磁盘使用模式以随机存取为主，如果平均 i/o 数据大于 32k，则以顺序存取为主。

顺序 i/o 由吞吐量决定；

随机 i/o 由 IOPS 决定；

附：

平均 io 数据尺寸=吞吐量/io 数目；

吞吐量：硬盘传输数据流的速度：传出数据=读出数据+写入数据

每秒 i/o 数（IOPS,tps）：一次磁盘的连续读或连续写称为一次磁盘 i/o。iops 每秒磁盘连续读和连续写次数之和。

IO 优化：

1）增加内存，增加磁盘硬件缓存（SSD 固态硬盘、硬盘缓存）

2）调整 raid 级别：raid（独立冗余磁盘阵列）

基于 raid 卡硬 raid（硬盘实现、速度高、适用范围于大型应用），基于系统的软 raid（可优化，适合小型应用）raid 级别：raid0：无校验，数据分段写入磁盘，吞吐量大，不容错，至少两块，安全性不高 raid1：镜像，容错，读性能好，至少两块，一般是 2 的倍数 raid5：分布式奇偶校验，至少三块，支持热备盘，数据经常更新时开销较大 raid6：两份校验可以同时坏两个盘，至少是四块盘 raid10：先做 raid1，再作 raid0 至少四块

3）选择 IO 调度器

主要的功能将随机 IO 尽可能合并为顺序 IO 调度算法：CFQ（完全公平队列算法，比较适合于交互式场景）Deadline（最后期限，比较适合于数据库服务上）Anticipatory（预期的，比较适合于行为预估的场景下）Noop（先到先得）

4）根据场景选择合适的文件系统 ext3，ext4（最小分配 8KB->128KB 预读取、延迟分配）

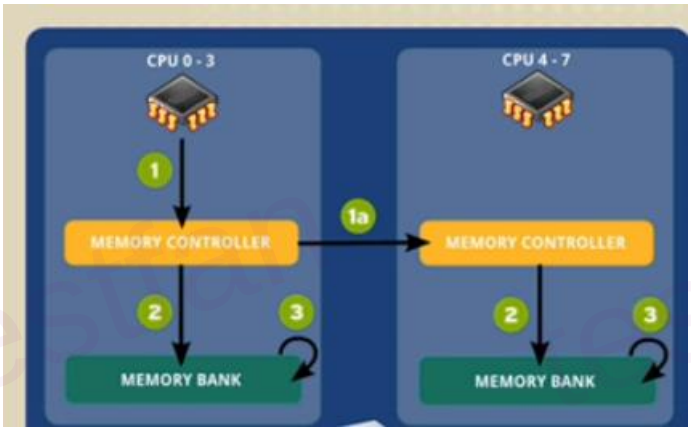
4.3 Tcp 连接及内存资源使用情况

TCP 连接需要占用操作系统以下资源，分别为：socket 文件描述符、IP 地址、端口、内存。源 IP、源端口、目标 IP、目标端口确定了唯一的一个 tcp 连接。

服务端的一个 TCP 连接确定后，IP 地址和端口随即确定，每个 TCP 连接占用 1 个 socket 文件描述符（fd），约占用 1KB 左右的内存，每个 TCP 连接都需要双方接收和发送数据，那么就需要一个读缓冲区和写缓冲区，这两个 buffer 在 linux 下最小为 4k。

综上，一个 TCp 连接占用的内存(系统内核参数默认的情况下）大概为 9k，为方便计算取 10k。1 万个连接需要的内存=10000*10k=100M.

4.4 为何对于 numa 架构的 CPU 需要进行绑定



通过图示可以看出，NUMA 架构的 cpu 内存是独享的。此架构下的 cpu 会产生一个问题如果访问的内存地址是夸节点的访问，效率会很低（通过 watch -n1 --differences numastat 可以观察 miss 值变化），此时我们通常需要将进程的 pid 绑定到单独的 node 或者 cpu，绑定时，对 numa 架构的建议使用 numa 自身的绑定命令处理，否则使用 taskset 进行绑定。

4.5 Numa 架构单实例和多实例优化策略

首先 NUMA 的内存分配策略有四种：

1.缺省(default)：总是在本地节点分配（分配在当前进程运行的节点上）；

2.绑定(bind)：强制分配到指定节点上；

3.交叉(interleave)：在所有节点或者指定的节点上交织分配；

4.优先(preferred)：在指定节点上分配，失败则在其他节点上分配。

因为 NUMA 默认的内存分配策略是优先在进程所在 CPU 的本地内存中分配，会导致 CPU 节点之间内存分配不均衡，当某个 CPU 节点的内存不足时，会导致 swap 产生，而不是从远程节点分配内存。这就是所谓的 swap insanity 现象。

- 如果一台服务器需要值部署一个单例（内存消耗比较大的应用），为避免上述情况的发生，此时我们通常需要关闭 numa。方法有：

1.硬件层，在 BIOS 中设置关闭；

2.OS 内核，启动时设置 numa=off；

3.可以用 numactl 命令将内存分配策略修改为 interleave（交叉）。即：

numactl --interleave=all

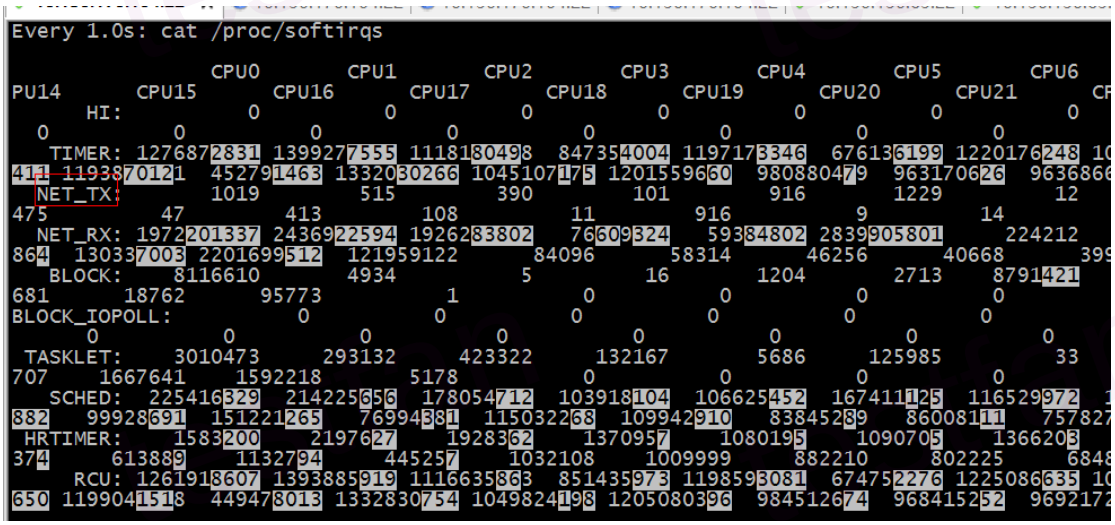
- 如果单机部署多个实例（内存消耗比较小，要求更快的程序运行时间），有利于提升服务器使用效率。通常需要使用绑定 node 或者绑定 cpu，此时我们需要特别注意应用的占用内存不能超过 node 的空闲内存（通过 numactl–H 查看），否则会出现 swap 的现象。

正确绑定：numactl --cpunodebind 1 --membind 1 myapplication

错误绑定：numactl –cpubind0 –membind0,1myapplication

4.6 中断种类及分析方法

使用命令 watch -d -n 1 'cat /proc/softirqs'



右侧为中断种类，NET_TX 或 NET _RX 如果变化很快的话就是网络中断引起的，否则是由系统其他原因造成的中断。

4.7 GC 回收器选择

比较成熟的 GC 基本上有三种选择，serial、Parallel 和 CMS。serial 和 ParallelGC 都是完

全 stop the world 的 GC，会暂停时间会很长，而 CMS 执行过程是非完全暂停。我们一般会选择：新生代使用 UseParNewGC 设置年轻代为并行收集，年老代使用 CMS 收集器

4.8 年轻代与年老代的参数调优建议

主要从耗时、频率两个方面考虑，尽可能降低这两个值。

young 区：尽量大，太小可能导致对象直接进入 old 区，如果 old 满了，触发 full gc，但不能过大，过大会引起回收耗时过长，导致应用停顿，gui 程序不要太大的 young 区。

old 区：尽量大，过小会导致产生 old 区小碎片，放不下大对象，引起频繁 full gc。如果用了缓存，old 区要适当大些，同时缓存不应该无限增长。

1、年轻代小对象尽量多，大对象尽可能直接进入老年代。年轻代由于使用标记复制算法进行回收内存，速度很快，当 Eden 区没有足够的空间时会引发一次 yungGC，通过-XX:SurvivorRatio 进行调整 Eden 和 Survivor 比例大小，当 yongGC 的时候，会将 Eden 区的对象放到 Survivor 区，如果 Survivor 空间不足，将通过分配担保机制(老年代负责分配担保让 Survivor 无法容纳的对象直接进入老年代，如果剩余空间小于转移对象大小，将进行 FullGc)提前转移到老年代；

2、少量对象存活，适合复制算法（年轻代），大量对象存活，适合标记清理或者标记压缩（年老代）。

4.9 GC 回收时常见的异常

一、 [ParNew (promotion failed): 320138K->320138K(353920K), 0.2365970 secs]42576.951: [CMS: 1139969K->1120688K(2166784K), 9.2214860 secs] 1458785K->1120688K(2520704K), 9.4584090 secs]

原因是是由于救助空间不够，从而向年老代转移对象，年老代没有足够的空间来容纳这些对象，导致一次 full gc 的产生。

解决办法（选择其中的一个）：

1、**增大救助空间**

增大救助空间就是调整-XX:SurvivorRatio 参数，这个参数是 Eden 区和 Survivor 区的大小比值，默认是 8，也就是说 Eden 区是 Survivor 区的 8 倍大小，要注意 Survivo 是有两个区的，因此 Surivivor 其实占整个 young genertation 的 1/10。调小这个参数将增大 survivor 区，让对象尽量在 survitor 区呆长一点，减少进入年老代的对象。

2、增大年老代或者去掉救助空间

去掉救助空间的想法是让大部分不能马上回收的数据尽快进入年老代，加快年老代的回收频率，减少年老代暴涨的可能性，这个是通过将-XX:SurvivorRatio 设置成比较大的值（比如 65536)来做到。

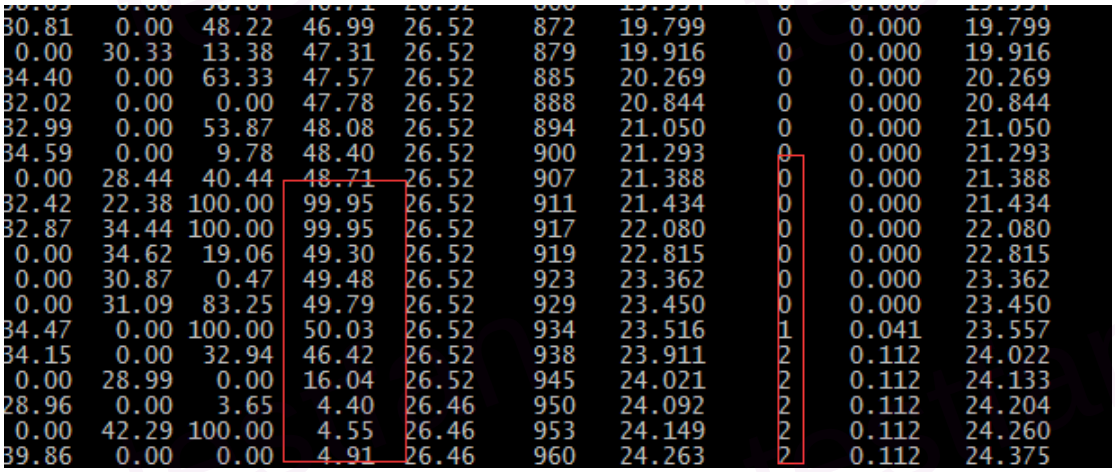
二、 [GC 90010.628: [ParNew: 261760K->261760K(261952K), 0.0000350secs]90010.628: [CMS (concurrent mode failure)]

原因是新空间分配请求在年老代的剩余空间中无法得到满足。

解决办法是需要减少 young、增加 old 的大小，或者使用 -XX:CMSFullGCsBeforeCompaction 并设置较小的值，提高 full gc 后压缩 old 的频次，避免 young 大对象无法晋升到 old.

4.10CMS 回收是否等于 FULL GC?

不等于。



30.81	0.00	48.22	46.99	26.52	872	19.799	0	0.000	19.799
0.00	30.33	13.38	47.31	26.52	879	19.916	0	0.000	19.916
34.40	0.00	63.33	47.57	26.52	885	20.269	0	0.000	20.269
32.02	0.00	0.00	47.78	26.52	888	20.844	0	0.000	20.844
32.99	0.00	53.87	48.08	26.52	894	21.050	0	0.000	21.050
34.59	0.00	9.78	48.40	26.52	900	21.293	0	0.000	21.293
0.00	28.44	40.44	48.71	26.52	907	21.388	0	0.000	21.388
32.42	22.38	100.00	99.95	26.52	911	21.434	0	0.000	21.434
32.87	34.44	100.00	99.95	26.52	917	22.080	0	0.000	22.080
0.00	34.62	19.06	49.30	26.52	919	22.815	0	0.000	22.815
0.00	30.87	0.47	49.48	26.52	923	23.362	0	0.000	23.362
0.00	31.09	83.25	49.79	26.52	929	23.450	0	0.000	23.450
34.47	0.00	100.00	50.03	26.52	934	23.516	1	0.041	23.557
34.15	0.00	32.94	46.42	26.52	938	23.911	2	0.112	24.022
0.00	28.99	0.00	16.04	26.52	945	24.021	2	0.112	24.133
28.96	0.00	3.65	4.40	26.46	950	24.092	2	0.112	24.204
0.00	42.29	100.00	4.55	26.46	953	24.149	2	0.112	24.260
39.86	0.00	0.00	4.91	26.46	960	24.263	2	0.112	24.375

从图示上看出当 old 到了 99.95%时并没有 FGC 还是 0.

首先必须明确 CMS 对年老代对象回收的过程，CMS 过程：

initial-mark> concurrent-mark> concurrent-preclean> remark> concurrent-sweep> concurrent-reset

其中 initial mark 和 remark 会产生 stop the world。只有 stop the world 的阶段被计算到了 Full GC 的次数和时间。一般的一次 CMS 至少会给 Full GC 的次数 + 2。

以下日志上看总共发生了 2 次 CMS，所以 **Full GC 的次数应该是 4**

2014-12-08T17:24:18.514+0800: 77443.326: [GC [1 CMS-initial-mark: 1382782K(1843200K)] 1978934K(4710400K), 0.0702700 secs] [Times: user=0.07 sys=0.00, real=0.07 secs]

2014-12-08T17:24:18.586+0800: 77443.398: [CMS-concurrent-mark-start]

2014-12-08T17:24:19.890+0800: 77444.702: [CMS-concurrent-mark: 1.206/1.303 secs] [Times: user=2.80 sys=0.07, real=1.30 secs]

2014-12-08T17:24:19.890+0800: 77444.702: [CMS-concurrent-preclean-start]

2014-12-08T17:24:19.906+0800: 77444.718: [CMS-concurrent-preclean: 0.015/0.015 secs] [Times: user=0.02 sys=0.00, real=0.02 secs]

2014-12-08T17:24:19.906+0800: 77444.718: [CMS-concurrent-abortable-preclean-start]

CMS: abort preclean due to time 2014-12-08T17:24:25.181+0800: 77449.993: [CMS-concurrent-abortable-preclean: 5.241/5.275 secs] [Times: user=6.03 sys=0.09,

real=5.27 secs]

2014-12-08T17:24:25.187+0800: 77449.999: [GC[YG occupancy: 749244 K (2867200 K)]77450.000: [Rescan (parallel) , 0.0276780 secs]77450.028: [weak refs processing, 0.2029030 secs]

[1 CMS-remark: 1382782K(1843200K)] 2132027K(4710400K), 0.2340660 secs] [Times: user=0.43 sys=0.00, real=0.23 secs]

2014-12-08T17:24:25.424+0800: 77450.236: [CMS-concurrent-sweep-start]

2014-12-08T17:24:27.420+0800: 77452.232: [CMS-concurrent-sweep: 1.918/1.996 secs] [Times: user=2.61 sys=0.05, real=2.00 secs]

2014-12-08T17:24:27.421+0800: 77452.233: [CMS-concurrent-reset-start]

2014-12-08T17:24:27.430+0800: 77452.242: [CMS-concurrent-reset: 0.010/0.010 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]

2014-12-09T12:45:05.545+0800: 147090.358: [GC [1 CMS-initial-mark: 1384080K(1843200K)] 2013429K(4710400K), 0.0656230 secs] [Times: user=0.06 sys=0.00, real=0.07 secs]

2014-12-09T12:45:05.613+0800: 147090.425: [CMS-concurrent-mark-start]

2014-12-09T12:45:06.848+0800: 147091.660: [CMS-concurrent-mark: 1.196/1.235 secs] [Times: user=2.77 sys=0.03, real=1.23 secs]

2014-12-09T12:45:06.849+0800: 147091.661: [CMS-concurrent-preclean-start]

2014-12-09T12:45:06.862+0800: 147091.674: [CMS-concurrent-preclean: 0.013/0.013 secs] [Times: user=0.02 sys=0.00, real=0.02 secs]

2014-12-09T12:45:06.862+0800: 147091.674: [CMS-concurrent-abortable-preclean-start]

CMS: abort preclean due to time 2014-12-09T12:45:11.874+0800: 147096.686: [CMS-concurrent-abortable-preclean: 4.948/5.012 secs] [Times: user=6.04 sys=0.10, real=5.01 secs]

2014-12-09T12:45:11.882+0800: 147096.694: [GC[YG occupancy: 815312 K (2867200 K)]147096.695: [Rescan (parallel) , 0.0476710 secs]147096.743: [weak refs

processing, 0.1565260 secs]

[1 CMS-remark: 1384080K(1843200K)] 2199393K(4710400K), 0.2064660 secs] [Times: user=0.48 sys=0.00, real=0.20 secs]

2014-12-09T12:45:12.091+0800: 147096.903: [CMS-concurrent-sweep-start]

2014-12-09T12:45:14.078+0800: 147098.890: [CMS-concurrent-sweep: 1.934/1.986 secs] [Times: user=2.43 sys=0.04, real=1.99 secs]

2014-12-09T12:45:14.078+0800: 147098.890: [CMS-concurrent-reset-start]

2014-12-09T12:45:14.084+0800: 147098.896: [CMS-concurrent-reset: 0.006/0.006 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]

Full GC 的时间 = 0.07 secs(第一次 initial mark)+ 0.23 secs(第一次 remark) + 0.07 secs(第二次 initial mark) + 0.20 secs(第二次 remark) = 0.57s，与观察时间是相同的

```
PU     YGC     YGCT    FGC    FGCT     GCT
262144.0 169568.9   2338   481.745   4       0.576   482.321
262144.0 169568.9   2338   481.745   4       0.576   482.321
262144.0 169568.9   2338   481.745   4       0.576   482.321
```

4.11判断 FULL GC 是否正常的标准

FGCT/FGC<=200ms.举例

```
PU     YGC     YGCT    FGC    FGCT     GCT
 262144.0 149951.4   8805 2623.948  223   1188.750 3812.699
 262144.0 149951.4   8805 2623.948  223   1188.750 3812.699
 262144.0 149951.4   8805 2623.948  223   1188.750 3812.699
 262144.0 149951.4   8805 2623.948  223   1188.750 3812.699
 262144.0 149951.4   8805 2623.948  223   1188.750 3812.699
```

Full GC 的平均时间 = 1188 / 223 = 5 秒，也就是说单次 Full GC 的 stop the world 的时间达到了 5s。查看单条日志：

```
72117.421: [Full GC (System) 72117.423: [CMS: 905025K->1529502K(4096000K), 5.3562640
secs] 3556285K->1529502K(7168000K), [CMS Perm : 148049K->147945K(262144K)], 5.358779
0secs] [Times: user=5.36 sys=0.00, real=5.36 secs]
```

我们看到是 System.gc 引起的 Full GC，而老年代的 GC 时间到达了 5 秒多，它显示的是 CMS，但是实际上不是 CMS 并发的收集器，而是 CMS 发生了 concurrent mode fail 之后退化成了 Serial Old 收集器，它是单线程的标记-压缩收集器，所以耗时非常的长。

4.12FULL GC 出现的几种情况

1. System.gc()方法的调用
2. 老年代代空间不足
3. 永久区空间不足
4. CMS GC 时出现 promotion failed 和 concurrent mode failure。

   说明：对于采用 CMS 进行老年代 GC 的程序而言，尤其要注意 GC 日志中是否有 promotion failed 和 concurrent mode failure 两种状况。
5. 统计得到的 Minor GC 晋升到老年代的平均大小大于老年代的剩余空间
6. 堆中分配很大的对象

   说明：大对象，是指需要大量连续内存空间的 java 对象，例如很长的数组。老年代虽然有很大的剩余空间，但是无法找到足够大的连续空间来分配给当前对象，此种情况就会触发 JVM 进行 Full GC。一般使用 -XX:+UseCMSCompactAtFullCollection 和 -XX:CMSFullGCsBeforeCompaction 参数结合来解决问题。
7. NIO 使用 DirectBuffer 分配物理内存，空间不足。

   说明：当 DirectBuffer 占用内存空间不足时，会显示调用 system.gc()，主动触发 fullgc。所以对于 NIO 的应用建议不要在 JVM 启动参数中追加-XX:+ DisableExplicitGC，否则会出现 DirectBuffer oom。

4.13CMS 常用参数

-XX:+UseConcMarkSweepGC 启用 CMS。

-XX:+UseCMSCompactAtFullCollection 在 full gc 的时候，对 old 区压缩。

-XX:+CMSScavengeBeforeRemark 这个参数还重要的，它的意思是在执行 CMS remark 之前进行一次 youngGC，这样能有效降低 remark 的时间，之前没有加这个参数，remark 时间最大能达到 3s，加上这个参数之后 remark 时间减少到 1s

之内。但是 remark 之后也将立即开始又一次 minor gc。

-XX:CMSFullGCsBeforeCompaction=1 多少次 full gc 后进行 old 区压缩，cms 会产生 old 区"碎片"，要进行整理，避免没有连续空间放大对象而引发 cms failure 出现。

-XX:CMSInitiatingOccupancyFraction=70 old 区使用 70%后开始 CMS 收集，jdk5 默认 68%，jdk6 默认 92%

-XX:CMSInitiatingPermOccupancyFraction=70 perm 区使用 70%后开始 CMS 收集，jdk5 默认 68%，jdk6 默认 92%

4.14堆内存溢出后处理方案。

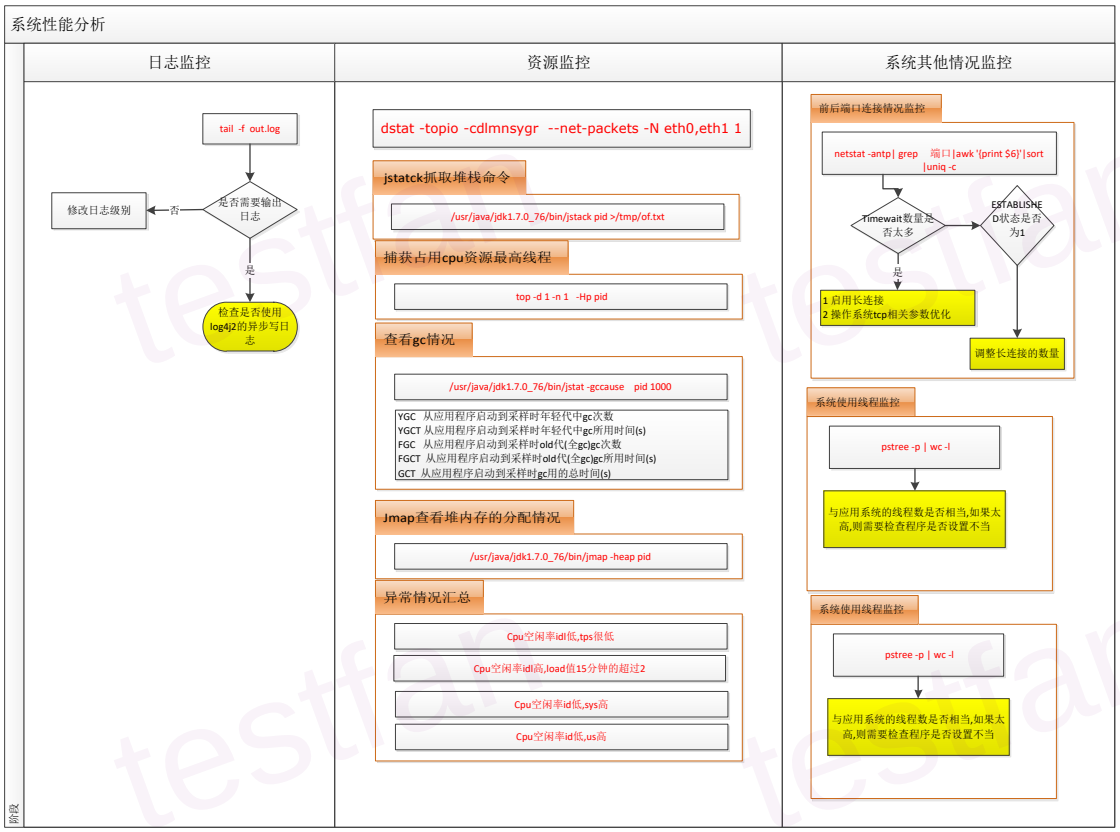
1、 增加-XX:OnOutOfMemoryError。在 OOM 时，执行一个脚本。

例如-XX:OnOutOfMemoryError=D:/tools/jdk1.7_40/bin/printstack.bat %p

2、 使用-XX:+HeapDumpOnOutOfMemoryError、-XX:+HeapDumpPath 当发生 OOM 时，导出堆到指定文件

例 如 ： -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/home/a.dump

5 瓶颈分析总结及案例



监控命令整体图

性能分析总结：

- **在压力足够大的情况下，除了资源监控外，还要看日志，观连接，抓堆栈。**
- **不要过度优化，当性能满足的情况下，以稳定性为主。如连接数 1 能满足不要设置为 2.**
- **单点性能调优有限时，充分利用资源，部署多点服务。**

5.1 Cpu 使用率很高，tps 很低

现象：在虚拟机下 tps：4000，cpu 使用率在 90%以上

排查问题步骤：

1. 使用命令： top -H -p pid（pid 为被测系统的进程号），找到导致 cpu 高的线

程 id。

2. 将线程 Id 由十进制转换为十六进制

3. 在 dump thread 信息中线程的 nid

问题确定：发现线程在执行 json 序列化。

解决方案：1、将序列化对象本地化缓存

2、不使用序列化，绕过 repsonseBody 直接将字符串写入响应流。

5.2 Cpu 使用率很低，tps 很低，IO 无瓶颈

现象：dubbo 消费端调用服务端服务，消费端没有压力，服务端没有压力

排查问题步骤：

1. 使用命令：jstack【pid】

2. 查找线程状态为 WAITING、TIMED_WAITING 的线程的堆栈情况

问题确定：发现线程都在等待获取连接。

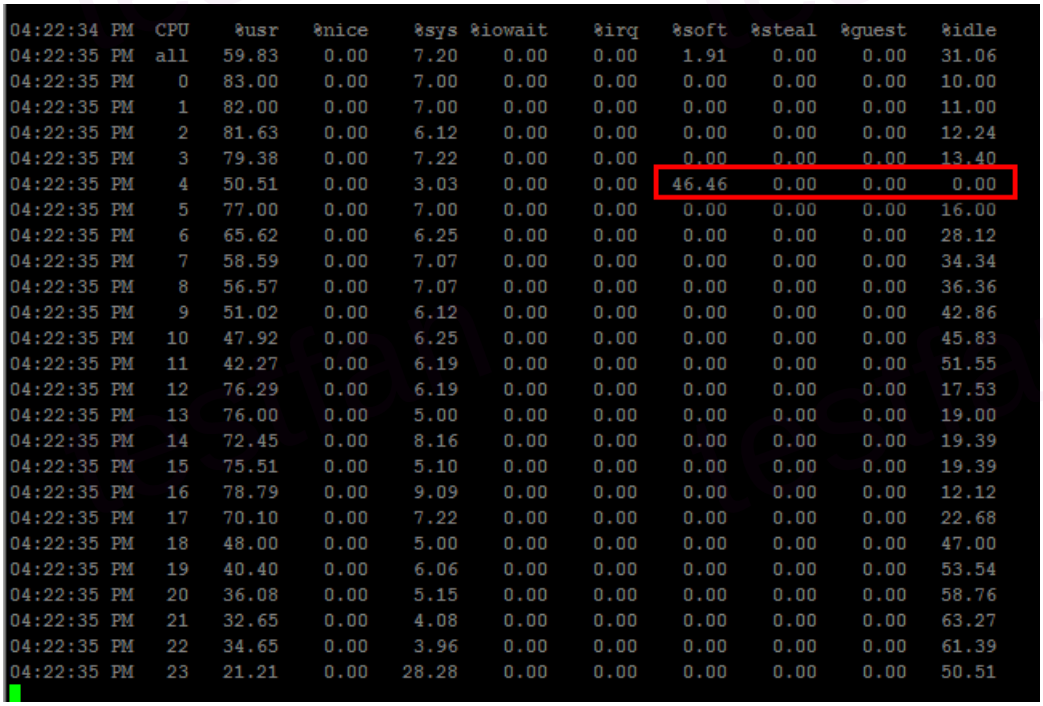
解决方案：1、增加连接数

5.3 Cpu 使用率不高，增大压力，tps 无变化

现象：tomcat 写 mongodb 数据，无论加压多大，增加连接数，增加线程数，tps 都无变化。

排查问题步骤：

1、 mpstat -P ALL 1 观察 tomcat 的 cpu 资源使用情况，



	CPU	%usr	%nice	%sys	%iowait	%irq	%soft	%steal	%guest	%idle
04:22:35 PM	all	59.83	0.00	7.20	0.00	0.00	1.91	0.00	0.00	31.06
04:22:35 PM	0	83.00	0.00	7.00	0.00	0.00	0.00	0.00	0.00	10.00
04:22:35 PM	1	82.00	0.00	7.00	0.00	0.00	0.00	0.00	0.00	11.00
04:22:35 PM	2	81.63	0.00	6.12	0.00	0.00	0.00	0.00	0.00	12.24
04:22:35 PM	3	79.38	0.00	7.22	0.00	0.00	0.00	0.00	0.00	13.40
04:22:35 PM	4	50.51	0.00	3.03	0.00	0.00	46.46	0.00	0.00	0.00
04:22:35 PM	5	77.00	0.00	7.00	0.00	0.00	0.00	0.00	0.00	16.00
04:22:35 PM	6	65.62	0.00	6.25	0.00	0.00	0.00	0.00	0.00	28.12
04:22:35 PM	7	58.59	0.00	7.07	0.00	0.00	0.00	0.00	0.00	34.34
04:22:35 PM	8	56.57	0.00	7.07	0.00	0.00	0.00	0.00	0.00	36.36
04:22:35 PM	9	51.02	0.00	6.12	0.00	0.00	0.00	0.00	0.00	42.86
04:22:35 PM	10	47.92	0.00	6.25	0.00	0.00	0.00	0.00	0.00	45.83
04:22:35 PM	11	42.27	0.00	6.19	0.00	0.00	0.00	0.00	0.00	51.55
04:22:35 PM	12	76.29	0.00	6.19	0.00	0.00	0.00	0.00	0.00	17.53
04:22:35 PM	13	76.00	0.00	5.00	0.00	0.00	0.00	0.00	0.00	19.00
04:22:35 PM	14	72.45	0.00	8.16	0.00	0.00	0.00	0.00	0.00	19.39
04:22:35 PM	15	75.51	0.00	5.10	0.00	0.00	0.00	0.00	0.00	19.39
04:22:35 PM	16	78.79	0.00	9.09	0.00	0.00	0.00	0.00	0.00	12.12
04:22:35 PM	17	70.10	0.00	7.22	0.00	0.00	0.00	0.00	0.00	22.68
04:22:35 PM	18	48.00	0.00	5.00	0.00	0.00	0.00	0.00	0.00	47.00
04:22:35 PM	19	40.40	0.00	6.06	0.00	0.00	0.00	0.00	0.00	53.54
04:22:35 PM	20	36.08	0.00	5.15	0.00	0.00	0.00	0.00	0.00	58.76
04:22:35 PM	21	32.65	0.00	4.08	0.00	0.00	0.00	0.00	0.00	63.27
04:22:35 PM	22	34.65	0.00	3.96	0.00	0.00	0.00	0.00	0.00	61.39
04:22:35 PM	23	21.21	0.00	28.28	0.00	0.00	0.00	0.00	0.00	50.51

问题确定：软中断全在一个 cpu 核心上，且 idl 都为 0，而其他核心的 cpu 相对压力很小

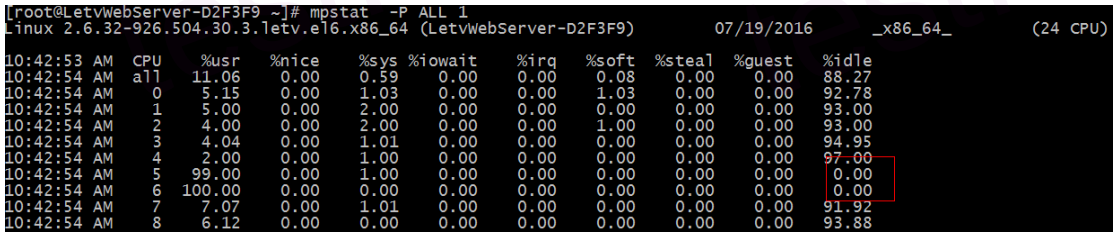
解决方案：根据观察此物理机为单队列网卡，开启 RPS，故直接运行脚本 set_irq.sh(附件)，

5.4 无软中断，个别 cpu 使用率在 100%

现象：logstach 在工作的时候无法使用更多的 cpu

排查步骤：

1、mpstat -P ALL 1



```
[root@LetvWebServer-D2F3F9 ~]# mpstat -P ALL 1
Linux 2.6.32-926.504.30.3.letv.el6.x86_64 (LetvWebServer-D2F3F9)        07/19/2016      _x86_64_        (24 CPU)

10:42:53 AM  CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest   %idle
10:42:54 AM  all   11.06    0.00    0.59    0.00    0.00    0.08    0.00    0.00   88.27
10:42:54 AM    0    5.15    0.00    1.03    0.00    0.00    1.03    0.00    0.00   92.78
10:42:54 AM    1    5.00    0.00    2.00    0.00    0.00    0.00    0.00    0.00   93.00
10:42:54 AM    2    4.00    0.00    2.00    0.00    0.00    1.00    0.00    0.00   93.00
10:42:54 AM    3    4.04    0.00    1.01    0.00    0.00    0.00    0.00    0.00   94.95
10:42:54 AM    4    2.00    0.00    1.00    0.00    0.00    0.00    0.00    0.00   97.00
10:42:54 AM    5   99.00    0.00    1.00    0.00    0.00    0.00    0.00    0.00    0.00
10:42:54 AM    6  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
10:42:54 AM    7    7.07    0.00    1.01    0.00    0.00    0.00    0.00    0.00   91.92
10:42:54 AM    8    6.12    0.00    0.00    0.00    0.00    0.00    0.00    0.00   93.88
```

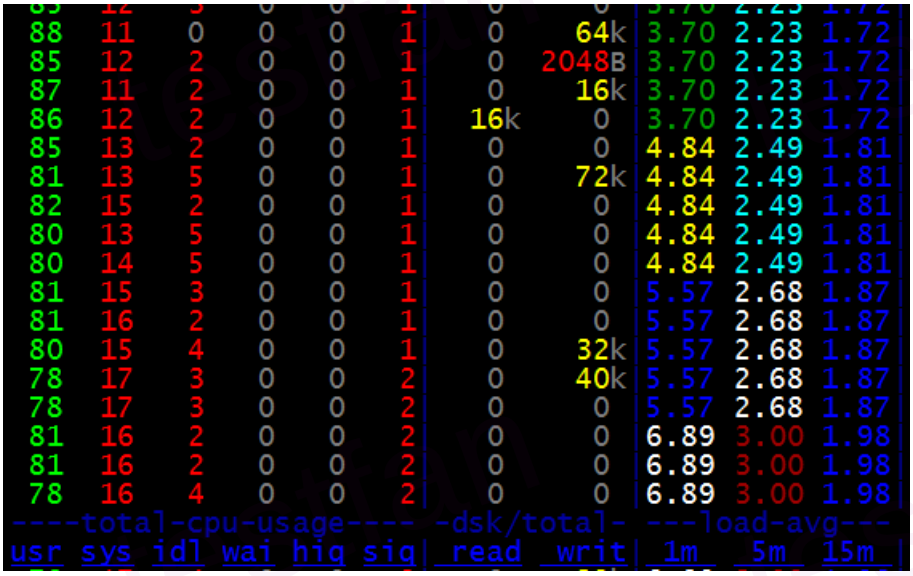
从图示上观察%irq 和%soft 没有比例值，说明没有发生任何的中断，使用 taskset 绑定进程 pid，无任何效果

问题原因：对于单线程的应用，只会使用 1 颗 cpu，绑定多颗无效

解决方案：可在服务器上部署多个应用实例，每个实例绑定 1 颗 cpu

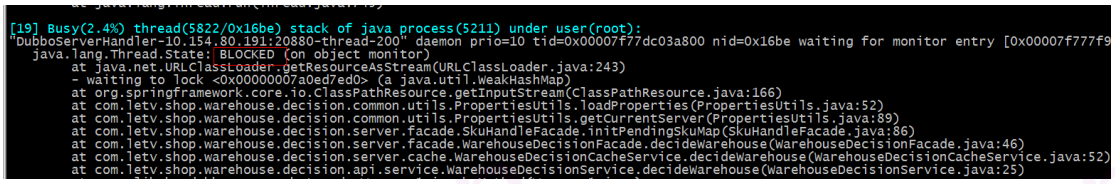
5.5 Cpu 压力很大，load 值很低，tps 很低

现象：dubbo 服务压测。消费端到服务端连接为 10，服务端为虚拟机，服务端存在问题，资源为 cpu 使用率在 95%，load 为小于 2，tps 只有 6000 左右



排查过程：

使用./show-busy-java-threads.sh -p 8321 -c 20 抓取堆栈分析



通过抓取堆栈观察，对应的代码

```
public static String getCurrentServer(String key) {
  try {
    currentProperties = loadProperties(new String[] { "/config/server.properties" });
    String value = new String(currentProperties.getProperty(key).getBytes("ISO8859_1"), "utf-8").trim();
    log.debug(key + ":" + value);
    return value;
  } catch (Exception e) {
    log.error("读取 static.properties 配置文件失败.", e);
```

问题原因：每次请求的处理都会 load 文件，产生 io 问题

解决方案：资源文件只加载一次，代码修改后，tps 提升到 2.1w。修改后代码

```
if (currentProperties == null) {
    currentProperties = loadProperties(new String[] { "/conf/server.properties" });
}
```

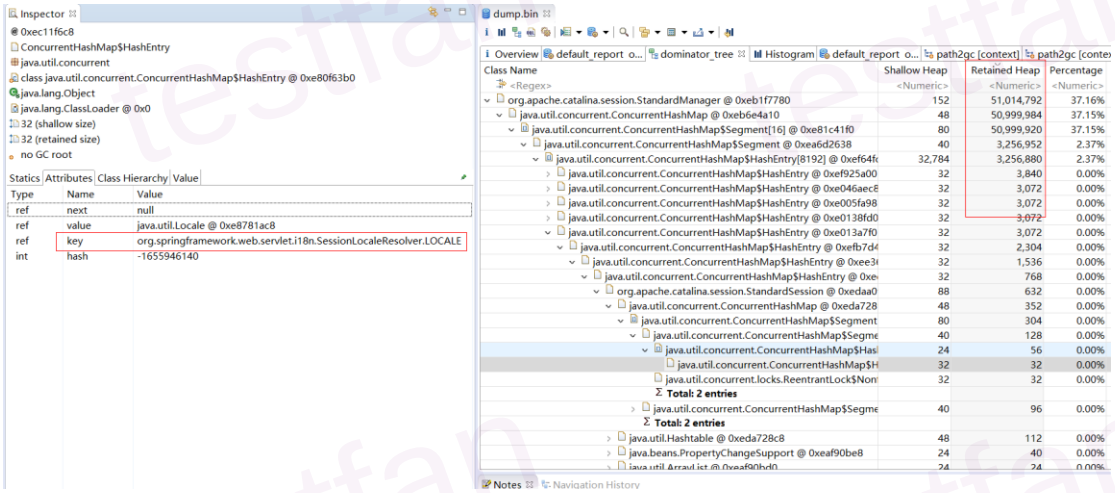
5.6 堆内存一直不能回收，tps 直接变为 0

现象：tps 前期稳定，后期变为 0

排查过程：观察 gc 情况，发现年老代的内存一直增加，gc 失效

```
c[root@tomcat150_66 ~]# /usr/java/jdk1.7.0_76/bin/jstat -gcutil  31427 1000
  S0     S1     E      O      P     YGC     YGCT    FGC    FGCT     GCT
  0.00  65.93 100.00 100.00  33.72   2448   22.444  3927 1282.144 1304.588
  0.00  69.26 100.00 100.00  33.72   2449   22.444  3930 1282.852 1305.296
  0.00  66.08 100.00 100.00  33.72   2450   22.444  3933 1283.558 1306.002
  0.00  65.18 100.00 100.00  33.72   2451   22.444  3936 1284.172 1306.616
  0.00  72.06 100.00 100.00  33.72   2452   22.444  3939 1284.967 1307.411
  0.00  88.74 100.00 100.00  33.72   2453   22.444  3942 1285.601 1308.045
  0.00 100.00 100.00 100.00  33.72   2455   22.444  3946 1286.202 1308.646
  0.00 100.00 100.00 100.00  33.72   2456   22.444  3949 1286.804 1309.248
  0.00 100.00 100.00 100.00  33.72   2457   22.444  3952 1287.490 1309.934
  0.00 100.00 100.00  99.99  33.72   2458   22.444  3955 1288.145 1310.589
  0.00 100.00 100.00  99.99  33.72   2459   22.444  3958 1288.771 1311.215
```

使用 MAT 工具分析对内存，在 Dominator Tree 观察，发现 org.apache.catalina.session.StandardManager 类，可回收内存最多。一直按照 Retained size 按照有大到小展开，发现 ConcurrentHashMap 中的 key 为 org.springframework.web.servlet.i18n.SessionLocaleResolver.LOCALE。



注：Retained Size 就是当前对象被 GC 后，从 Heap 上总共能释放掉的内存；Shallow Size 对象自身占用的内存大小，不包括它引用的对象

原因：Session 对象创建过多导致 jvm 堆内存溢出。

解决方案： 屏蔽 session 拦截器。

```
<!-- 国际化操作拦截器 如果采用基于（请求/session/cookie）则必需配置 -->
<!--    <bean class="com.letv.shop.demoWeb.web.interceptor.LocaleHandleInterceptor">
        <property name="paramName" value="locale" />
    </bean>
     -->
```

5.7 服务假死，telnet 可以通，但不能访问。

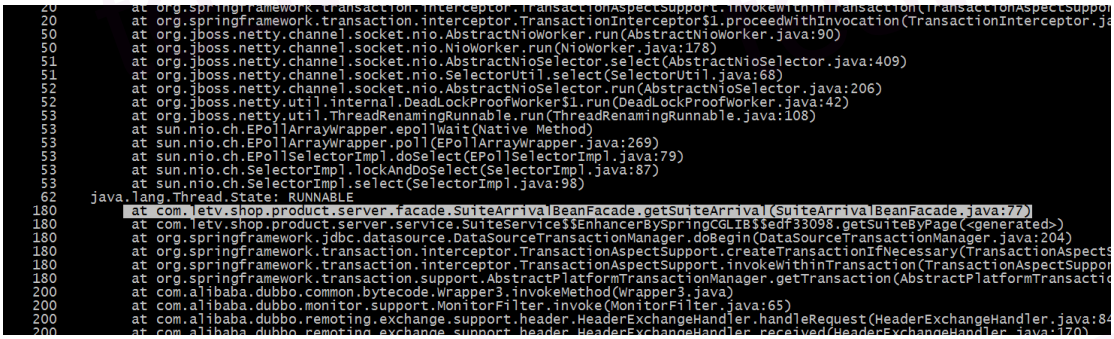
现象：dubbo 端口能够访问，但是 invoke 失败

排查过程：通过现象分析，初步定为有可能是线程池有耗尽产生的问题。

首先，使用 jstack 抓取堆栈信息

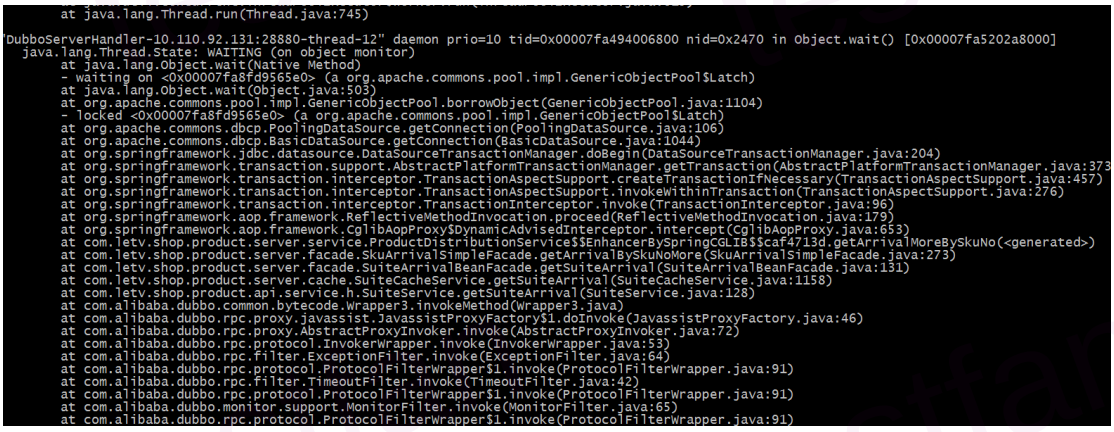
/usr/java/jdk1.7.0_76/bin/jstack 21792 >/tmp/of.txt

然后，对堆栈的信息进行排序统计（sort /tmp/of.txt |uniq -c| sort -nk 1）。



我们发现系统配置线程池的大小是 200，加亮后的方法却占用了 180 个线程。

使用 more 命令查看此方法的完整线程栈信息



原因：此方法的调用有复杂的 Sql，执行效率比较耗时，数据连接资源不足，执行线程全部挂起，线程池中无可用的线程资源。

解决方案：

1、 对 dubbo 服务方法使用 dubbo:method 增加访问限制并发控制。

2、 将重服务独立成单独 dubbo 应用，并增加本地缓存

5.8 集群性能差异分析

现象:购物车添加、删除购物车，pc 端性能 20Wtps，手机端只有 10wtps

排查过程：

1、 首先排除前端 LVS，nginx 的压力不足问题，因为 pc 端调用同样的过程。

2、 单压一台 tomcat，发现没有性能问题，同时分别压多台发现 tps 是倍增的关系

3、 单压一台 tomcat 直连后端一台 tomcat 服务，发现没有太大问题。

4、 单压域名，发现 tps10w 左右

原因：后端的服务能力限制了前端的性能。进一步比较发现 pc 端和手机端调用的域名并不一样。

解决方案：在压测集群的过程中，尽可能使用排除法，分步骤排除，用不同的组合进行压测。

5.9 集群压测服务响应超时

现象: 集群混合压测的时候，总是出现 morder、mcart 响应超时，应用在调用 dubbo 或者 http 发现有超时错误。

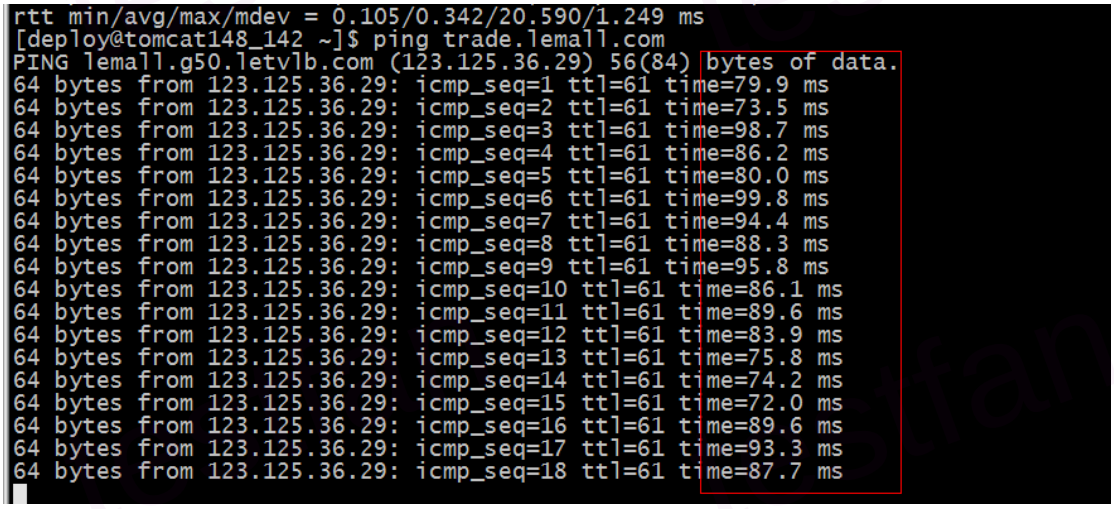
排查过程：

1、 单压一台 http 响应时间很短不在问题。

2、 单压一台 tomcat，响应时间也很短。

3、 单压应用集群响应时间很短不存在问题。

4、 再进行混合压，通过随机 ping 后端服务的域名



原因： 机房网络出现问题，经排查是机房网络带宽用尽，导致该机房内所有的服务响应超时。

解决方案： 增加机房网络带宽。

5.10一次 gc 调优案例全过程

调优的基本原则：gc 次数越少越好，时间越短越好；对象尽可能在年轻代完成收集，尽可能减少 fullgc；在对象数量很多的情况下，尽可能让年轻代完成小对象收集，大对象如果在年轻代回收时间比较长，则应在对象生成后直接扔到年老代。

调优案例说明：本例为观察方便，使用串行收集器。

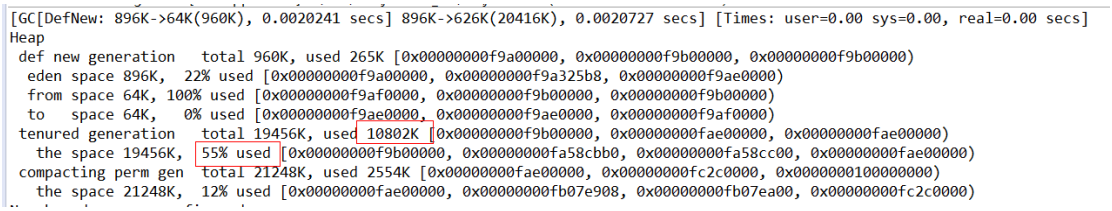
运行代码：

```
public static void testSurvivorRatio(){
    byte[] b=null;
   for(int i=0;i<10;i++){
     b=new byte[1*1024*1024];
   }
}
```

调优过程：

第 1 次：新生代设置很小的值

启动参数：-Xmx20m -Xms20m -Xmn1m -XX:+UseSerialGC -XX:+PrintGCDetails

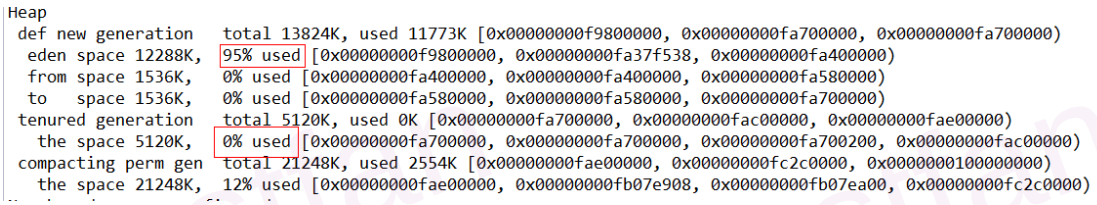


结果：发生 1 次 yungGc，对象直接进入老年代。新生代大小设置的 1M（**图片显示 960k+1 个幸存带的大小 64k=1024k**），对象大小为 1m，幸存区大小无法满足，直接进入年老代

存在问题：由于对象直接进入年老代，将会产生频繁产生 fullGc。

第 2 次：新生代设置很大的值

启动参数：-Xmx20m -Xms20m –Xmn15m -XX:+UseSerialGC -XX:+PrintGCDetails

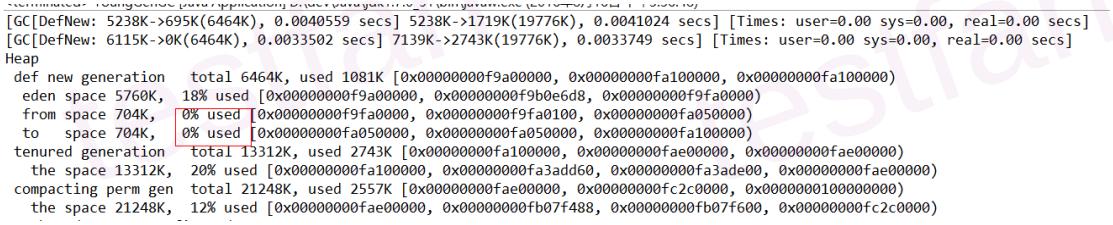


结果：未发生一次 gc，对象都存在于新生代的 eden 区，没有对象进入年老代

存在问题：年轻代空间过大，年老代太小。过小会导致产生 old 区小碎片，放不下大对象，引起频繁 full gc。

第 3 次：新生代设置设置折中值

启动参数：-Xmx20m -Xms20m –Xmn7m -XX:+UseSerialGC -XX:+PrintGCDetails



结果：进行了 2 次新生代 GC，s0、s1 太小，对象直接进入老年代。

存在问题： 幸存带没有起到垃圾回收的作用，进入老年代的几率变大，频繁 fullgc 的可能性变大。

第 4 次：增加幸存区域大小

启动参数：-Xmx20m -Xms20m –Xmn7m -XX:SurvivorRatio=2 -XX:+UseSerialGC

-XX:+PrintGCDetails

```
[GC[DefNew: 3170K->1719K(5376K), 0.0032853 secs] 3170K->1719K(18688K), 0.0033429 secs] [Times: user=0.00 sys=0.00, real=0.00 se
[GC[DefNew: 4982K->1024K(5376K), 0.0027490 secs] 4982K->1719K(18688K), 0.0027776 secs] [Times: user=0.00 sys=0.02, real=0.00 se
[GC[DefNew: 4127K->1024K(5376K), 0.0014673 secs] 4822K->1719K(18688K), 0.0015036 secs] [Times: user=0.00 sys=0.00, real=0.00 se
Heap
 def new generation   total 5376K, used 3164K [0x00000000f9a00000, 0x00000000fa100000, 0x00000000fa100000)
  eden space 3584K,  59% used [0x00000000f9a00000, 0x00000000f9c170f0, 0x00000000f9d80000)
  from space 1792K,  57% used [0x00000000f9f40000, 0x00000000fa040010, 0x00000000fa100000)
  to   space 1792K,   0% used [0x00000000f9d80000, 0x00000000f9d80000, 0x00000000f9f40000)
 tenured generation   total 13312K, used 695K [0x00000000fa100000, 0x00000000fae00000, 0x00000000fae00000)
   the space 13312K,   5% used [0x00000000fa100000, 0x00000000fa1ade40, 0x00000000fa1ae000, 0x00000000fae00000)
 compacting perm gen  total 21248K, used 2557K [0x00000000fae00000, 0x00000000fc2c0000, 0x0000000100000000)
   the space 21248K,  12% used [0x00000000fae00000, 0x00000000fb07f488, 0x00000000fb07f600, 0x00000000fc2c0000)
```

结果：进行了 3 次新生代 GC，相对上次年老代占用率明显减小

存在问题：新生代 gc 次数比较多，同时幸存区的内存有浪费。

第 4 次：增加幸存区域大小，增大新生代大小

启动参数：-Xmx20m -Xms20m –Xmn10m -XX:SurvivorRatio=4 -XX:+UseSerialGC -XX:+PrintGCDetails

```
[GC[DefNew: 6259K->1664K(8576K), 0.0035068 secs] 6259K->1719K(18816K), 0.0035580 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
Heap
 def new generation   total 8576K, used 7280K [0x00000000f9a00000, 0x00000000fa400000, 0x00000000fa400000)
  eden space 6912K,  81% used [0x00000000f9a00000, 0x00000000f9f7c1a8, 0x00000000fa0c0000)
  from space 1664K, 100% used [0x00000000fa260000, 0x00000000fa400000, 0x00000000fa400000)
  to   space 1664K,   0% used [0x00000000fa0c0000, 0x00000000fa0c0000, 0x00000000fa260000)
 tenured generation   total 10240K, used 55K [0x00000000fa400000, 0x00000000fae00000, 0x00000000fae00000)
   the space 10240K,   0% used [0x00000000fa400000, 0x00000000fa40dfa0, 0x00000000fa40e000, 0x00000000fae00000)
 compacting perm gen  total 21248K, used 2557K [0x00000000fae00000, 0x00000000fc2c0000, 0x0000000100000000)
   the space 21248K,  12% used [0x00000000fae00000, 0x00000000fb07f488, 0x00000000fb07f600, 0x00000000fc2c0000)
```

结果：进行了 1 次新生代 GC，老年代几乎未占用 。

存在问题：本次达到最优的效果，gc 次数很少，对象在年轻代完成回收，效率比较高效。但是实际线上需要结合业务考虑大对象是否放在年老代，如果小对象比较多，则应将大对象放到年老代（例 -XX:PretenureSizeThreshold=3145728 大于 3m 的对象直接进入老年代），防止小对象大量进入年老代。

6 附件


unset_irq.sh


set_irq.sh


cpu_info.sh


show-busy-java-threads.sh