**[JVM监控命令详解(转)](http://www.cnblogs.com/rainy-shurun/p/5732341.html)**

<https://www.cnblogs.com/rainy-shurun/p/5732341.html>

### JVM监控命令基本就是 [jps、jstack、jmap、jhat、jstat](http://blog.csdn.net/wisgood/article/details/25343845) 几个命令的使用就可以了

### 

### JDK本身提供了很多方便的JVM性能调优监控工具，除了集成式的VisualVM和jConsole外，还有jps、jstack、jmap、jhat、jstat等小巧的工具，本博客希望能起抛砖引玉之用，让大家能开始对JVM性能调优的常用工具有所了解。

    现实企业级[Java](http://lib.csdn.net/base/17" \o "Java EE知识库" \t "_blank)开发中，有时候我们会碰到下面这些问题：

* OutOfMemoryError，内存不足
* 内存泄露
* 线程死锁
* 锁争用（Lock Contention）
* Java进程消耗CPU过高
* ......

    这些问题在日常开发中可能被很多人忽视（比如有的人遇到上面的问题只是重启服务器或者调大内存，而不会深究问题根源），但能够理解并解决这些问题是Java程序员进阶的必备要求。本文将对一些常用的JVM性能调优监控工具进行介绍，希望能起抛砖引玉之用。本文参考了网上很多资料，难以一一列举，在此对这些资料的作者表示感谢！关于JVM性能调优相关的资料，请参考文末。

### A、jps(Java Virtual Machine Process Status Tool)

**jps主要用来输出JVM中运行的进程状态信息。语法格式如下：**

1 jps [options] [hostid]

如果不指定hostid就默认为当前主机或服务器。

命令行参数选项说明如下：

1 -q 不输出类名、Jar名和传入main方法的参数

2 -m 输出传入main方法的参数

3 -l 输出main类或Jar的全限名

4 -v 输出传入JVM的参数

比如下面：

1 root@ubuntu:/# jps -m -l

2 2458 org.artifactory.standalone.main.Main /usr/local/artifactory-2.2.5/etc/jetty.xml

3 29920 com.sun.tools.hat.Main -port 9998 /tmp/dump.dat

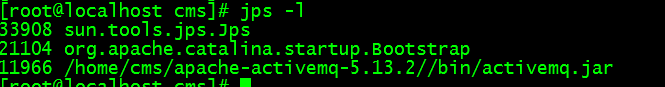
4 3149 org.apache.catalina.startup.Bootstrap start

5 30972 sun.tools.jps.Jps -m -l

6 8247 org.apache.catalina.startup.Bootstrap start

7 25687 com.sun.tools.hat.Main -port 9999 dump.dat

8 21711 mrf-center.jar



### B、 jstack

**jstack主要用来查看某个Java进程内的线程堆栈信息。语法格式如下：**

1 jstack [option] pid

2 jstack [option] executable core

3 jstack [option] [server-id@]remote-hostname-or-ip

命令行参数选项说明如下：

1 -l long listings，会打印出额外的锁信息，在发生死锁时可以用jstack -l pid来观察锁持有情况

2 -m mixed mode，不仅会输出Java堆栈信息，还会输出C/C++堆栈信息（比如Native方法）

**jstack可以定位到线程堆栈，根据堆栈信息我们可以定位到具体代码**，所以它在JVM性能调优中使用得非常多。下面我们来一个实例找出某个Java进程中最耗费CPU的Java线程并定位堆栈信息，用到的命令有ps、top、printf、jstack、grep。

第一步先找出Java进程ID，我部署在服务器上的Java应用名称为mrf-center：

1 root@ubuntu:/# ps -ef | grep mrf-center | grep -v grep

2 root 21711 1 1 14:47 pts/3 00:02:10 java -jar mrf-center.jar

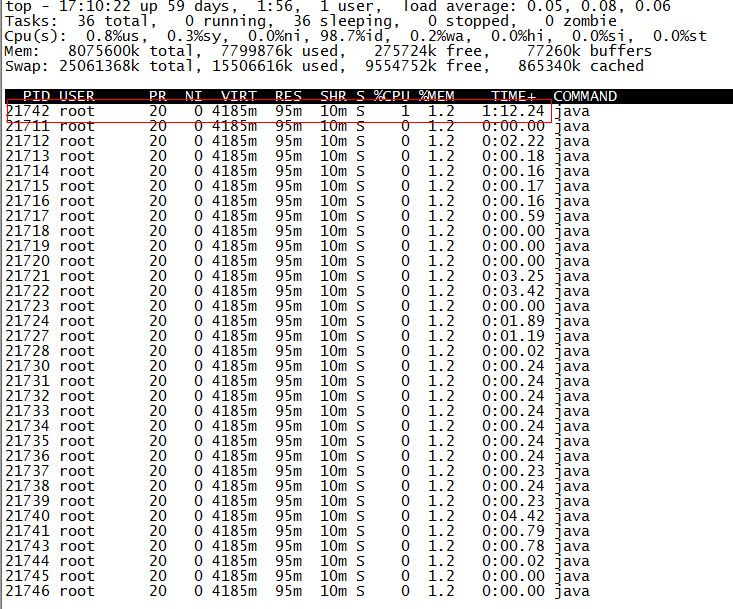
jstack可以定位到线程堆栈，根据堆栈信息我们可以定位到具体代码，所以它在JVM性能调优中使用得非常多。下面我们来一个实例找出某个Java进程中最耗费CPU的Java线程并定位堆栈信息，用到的命令有ps、top、printf、jstack、grep。

第一步先找出Java进程ID，我部署在服务器上的Java应用名称为mrf-center：

1 root@ubuntu:/# ps -ef | grep mrf-center | grep -v grep

2 root 21711 1 1 14:47 pts/3 00:02:10 java -jar mrf-center.jar

得到进程ID为21711，第二步找出该进程内最耗费CPU的线程，可以使用ps -Lfp pid或者ps -mp pid -o THREAD, tid, time或者**top -Hp pid，**我这里用第三个，输出如下：



TIME列就是各个Java线程耗费的CPU时间，CPU时间最长的是线程ID为21742的线程，用

1. printf "%x\n" 21742

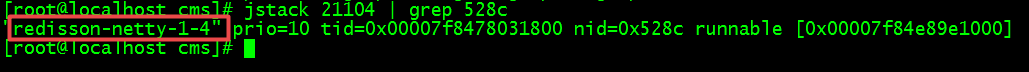


得到21742的十六进制值为**54ee**，下面会用到。

OK，下一步终于轮到jstack上场了，它用来输出进程21711的堆栈信息，然后根据线程ID的十六进制值grep，如下：

1 root@ubuntu:/# **jstack 21711 | grep 54ee**

2 "PollIntervalRetrySchedulerThread" prio=10 tid=0x00007f950043e000 nid=0x54ee in Object.wait() [0x00007f94c6eda000]



可以看到CPU消耗在PollIntervalRetrySchedulerThread这个类的Object.wait()，我找了下我的代码，定位到下面的代码：

01 // Idle wait

02 getLog().info("Thread [" + getName() + "] is idle waiting...");

03 schedulerThreadState = PollTaskSchedulerThreadState.IdleWaiting;

04 long now = System.currentTimeMillis();

05 long waitTime = now + getIdleWaitTime();

06 long timeUntilContinue = waitTime - now;

07 synchronized(sigLock) {

08 try {

09 if(!halted.get()) {

10 sigLock.wait(timeUntilContinue);

11 }

12 }

13 catch (InterruptedException ignore) {

14 }

15 }

它是轮询任务的空闲等待代码，上面的sigLock.wait(timeUntilContinue)就对应了前面的Object.wait()。

### C、 jmap（Memory Map）和jhat（Java Heap Analysis Tool）

jmap用来查看堆内存使用状况，一般结合jhat使用。

jmap语法格式如下：

1 jmap [option] pid

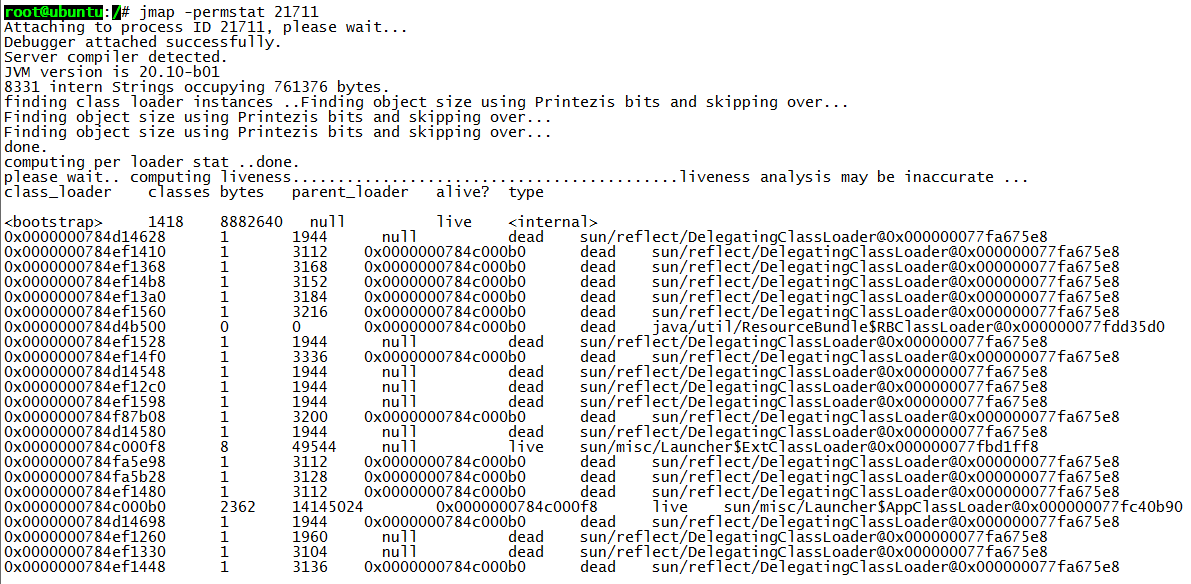
2 jmap [option] executable core

3 jmap [option] [server-id@]remote-hostname-or-ip

如果运行在64位JVM上，可能需要指定-J-d64命令选项参数。

1. **jmap -permstat pid**

 打印进程的类加载器和类加载器加载的持久代对象信息，输出：类加载器名称、对象是否存活（不可靠）、对象地址、父类加载器、已加载的类大小等信息，如下图：



使用**2、jmap -heap pid**查看进程堆内存使用情况，包括使用的GC算法、堆配置参数和各代中堆内存使用情况。比如下面的例子：

01 root@ubuntu:/# **jmap -heap 21711**

02 Attaching to process ID 21711, please wait...

03 Debugger attached successfully.

04 Server compiler detected.

05 JVM version is 20.10-b01

06

07 using thread-local object allocation.

08 Parallel GC with 4 thread(s)

09

10 Heap Configuration:

11 MinHeapFreeRatio = 40

12 MaxHeapFreeRatio = 70

13 MaxHeapSize = 2067791872 (1972.0MB)

14 NewSize = 1310720 (1.25MB)

15 MaxNewSize = 17592186044415 MB

16 OldSize = 5439488 (5.1875MB)

17 NewRatio = 2

18 SurvivorRatio = 8

19 PermSize = 21757952 (20.75MB)

20 MaxPermSize = 85983232 (82.0MB)

21

22 Heap Usage:

23 PS Young Generation

24 Eden Space:

25 capacity = 6422528 (6.125MB)

26 used = 5445552 (5.1932830810546875MB)

27 free = 976976 (0.9317169189453125MB)

28 84.78829520089286% used

29 From Space:

30 capacity = 131072 (0.125MB)

31 used = 98304 (0.09375MB)

32 free = 32768 (0.03125MB)

33 75.0% used

34 To Space:

35 capacity = 131072 (0.125MB)

36 used = 0 (0.0MB)

37 free = 131072 (0.125MB)

38 0.0% used

39 PS Old Generation

40 capacity = 35258368 (33.625MB)

41 used = 4119544 (3.9287033081054688MB)

42 free = 31138824 (29.69629669189453MB)

43 11.683876009235595% used

44 PS Perm Generation

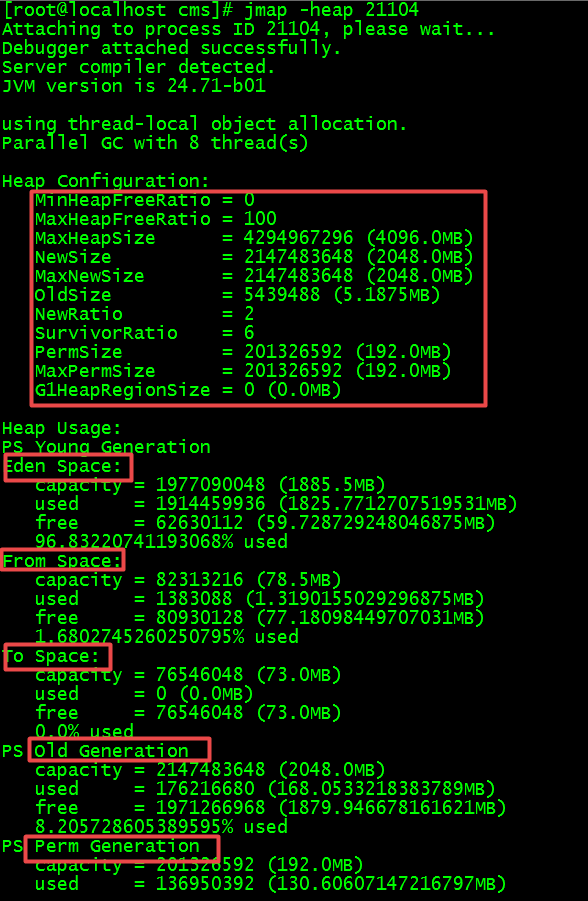
45 capacity = 52428800 (50.0MB)

46 used = 26075168 (24.867218017578125MB)

47 free = 26353632 (25.132781982421875MB)

48 49.73443603515625% used

49 ....



1: heap size   
a: -Xmx<n>   
指定 jvm 的最大 heap 大小 , 如 :-Xmx=2g   
b: -Xms<n>   
指定 jvm 的最小 heap 大小 , 如 :-Xms=2g ， 高并发应用， 建议和-Xmx一样， 防止因为内存收缩／突然增大带来的性能影响。   
c: -Xmn<n>   
指定 jvm 中 New Generation 的大小 , 如 :-Xmn256m。 这个参数很影响性能， 如果你的程序需要比较多的临时内存，建议设置到512M， 如果用的少， 尽量降低这个数值， 一般来说128／256足以使用了。   
d: -XX:PermSize=<n>   
指定 jvm 中 Perm Generation 的最小值 , 如 :-XX:PermSize=32m。 这个参数需要看你的实际情况，。可以通过jmap 命令看看到底需要多少。   
e: -XX:MaxPermSize=<n>   
指定 Perm Generation 的最大值 , 如 :-XX:MaxPermSize=64m   
f: -Xss<n>   
指定线程桟大小 , 如 :-Xss128k， 一般来说，webx框架下的应用需要256K。 如果你的程序有大规模的递归行为，请考虑设置到512K／1M。 这个需要全面的测试才能知道。 不过，256K已经很大了。 这个参数对性能的影响比较大的。   
g: -XX:NewRatio=<n>   
指定 jvm 中 Old Generation heap size 与 New Generation 的比例 , 在使用 CMS GC 的情况下此参数失效 , 如 :-XX:NewRatio=2   
h: -XX:SurvivorRatio=<n>   
指 定 New Generation 中 Eden Space 与一个 Survivor Space 的 heap size 比例 ,-XX:SurvivorRatio=8, 那么在总共 New Generation 为 10m 的情况下 ,Eden Space 为 8m   
i: -XX:MinHeapFreeRatio=<n>   
指定 jvm heap 在使用率小于 n 的情况下 ,heap 进行收缩 ,Xmx==Xms 的情况下无效 , 如 :-XX:MinHeapFreeRatio=30   
j: -XX:MaxHeapFreeRatio=<n>   
指定 jvm heap 在使用率大于 n 的情况下 ,heap 进行扩张 ,Xmx==Xms 的情况下无效 , 如 :-XX:MaxHeapFreeRatio=70   
k: -XX:LargePageSizeInBytes=<n>   
指定 Java heap 的分页页面大小 , 如 :-XX:LargePageSizeInBytes=128m

使用**jmap -histo[:live] pid**查看堆内存中的对象数目、大小统计直方图，如果带上live则只统计活对象，如下：

01 root@ubuntu:/# jmap -histo:live 21711 | more

02

03 num #instances #bytes class name

04 ----------------------------------------------

05 1: 38445 5597736 <constMethodKlass>

06 2: 38445 5237288 <methodKlass>

07 3: 3500 3749504 <constantPoolKlass>

08 4: 60858 3242600 <symbolKlass>

09 5: 3500 2715264 <instanceKlassKlass>

10 6: 2796 2131424 <constantPoolCacheKlass>

11 7: 5543 1317400 [I

12 8: 13714 1010768 [C

13 9: 4752 1003344 [B

14 10: 1225 639656 <methodDataKlass>

15 11: 14194 454208 java.lang.String

16 12: 3809 396136 java.lang.Class

17 13: 4979 311952 [S

18 14: 5598 287064 [[I

19 15: 3028 266464 java.lang.reflect.Method

20 16: 280 163520 <objArrayKlassKlass>

21 17: 4355 139360 java.util.HashMap$Entry

22 18: 1869 138568 [Ljava.util.HashMap$Entry;

23 19: 2443 97720 java.util.LinkedHashMap$Entry

24 20: 2072 82880 java.lang.ref.SoftReference

25 21: 1807 71528 [Ljava.lang.Object;

26 22: 2206 70592 java.lang.ref.WeakReference

27 23: 934 52304 java.util.LinkedHashMap

28 24: 871 48776 java.beans.MethodDescriptor

29 25: 1442 46144 java.util.concurrent.ConcurrentHashMap$HashEntry

30 26: 804 38592 java.util.HashMap

31 27: 948 37920 java.util.concurrent.ConcurrentHashMap$Segment

32 28: 1621 35696 [Ljava.lang.Class;

33 29: 1313 34880 [Ljava.lang.String;

34 30: 1396 33504 java.util.LinkedList$Entry

35 31: 462 33264 java.lang.reflect.Field

36 32: 1024 32768 java.util.Hashtable$Entry

37 33: 948 31440 [Ljava.util.concurrent.ConcurrentHashMap$HashEntry;

使用**jmap -histo[:live] pid**查看堆内存中的对象数目、大小统计直方图，如果带上live则只统计活对象，如下：

01 root@ubuntu:/# **jmap -histo:live 21711 | more**

02

03 num #instances #bytes class name

04 ----------------------------------------------

05 1: 38445 5597736 <constMethodKlass>

06 2: 38445 5237288 <methodKlass>

07 3: 3500 3749504 <constantPoolKlass>

08 4: 60858 3242600 <symbolKlass>

09 5: 3500 2715264 <instanceKlassKlass>

10 6: 2796 2131424 <constantPoolCacheKlass>

11 7: 5543 1317400 [I

12 8: 13714 1010768 [C

13 9: 4752 1003344 [B

14 10: 1225 639656 <methodDataKlass>

15 11: 14194 454208 java.lang.String

16 12: 3809 396136 java.lang.Class

17 13: 4979 311952 [S

18 14: 5598 287064 [[I

19 15: 3028 266464 java.lang.reflect.Method

20 16: 280 163520 <objArrayKlassKlass>

21 17: 4355 139360 java.util.HashMap$Entry

22 18: 1869 138568 [Ljava.util.HashMap$Entry;

23 19: 2443 97720 java.util.LinkedHashMap$Entry

24 20: 2072 82880 java.lang.ref.SoftReference

25 21: 1807 71528 [Ljava.lang.Object;

26 22: 2206 70592 java.lang.ref.WeakReference

27 23: 934 52304 java.util.LinkedHashMap

28 24: 871 48776 java.beans.MethodDescriptor

29 25: 1442 46144 java.util.concurrent.ConcurrentHashMap$HashEntry

30 26: 804 38592 java.util.HashMap

31 27: 948 37920 java.util.concurrent.ConcurrentHashMap$Segment

32 28: 1621 35696 [Ljava.lang.Class;

33 29: 1313 34880 [Ljava.lang.String;

34 30: 1396 33504 java.util.LinkedList$Entry

35 31: 462 33264 java.lang.reflect.Field

36 32: 1024 32768 java.util.Hashtable$Entry

37 33: 948 31440 [Ljava.util.concurrent.ConcurrentHashMap$HashEntry;

class name是对象类型，说明如下：

1 B byte

2 C char

3 D double

4 F float

5 I int

6 J long

7 Z boolean

8 [ 数组，如[I表示int[]

9 [L+类名 其他对象

还有一个很常用的情况是：用jmap把进程内存使用情况dump到文件中，再用jhat分析查看。jmap进行dump命令格式如下：

1 **jmap -dump:format=b,file=dumpFileName**

我一样地对上面进程ID为21711进行Dump：

1 root@ubuntu:/# j**map -dump:format=b,file=/tmp/dump.dat 21711**

2 Dumping heap to /tmp/dump.dat ...

3 Heap dump file created

dump出来的文件可以用MAT、VisualVM等工具查看，这里用jhat查看：

01 root@ubuntu:/# jhat -port 9998 /tmp/dump.dat

02 Reading from /tmp/dump.dat...

03 Dump file created Tue Jan 28 17:46:14 CST 2014

04 Snapshot read, resolving...

05 Resolving 132207 objects...

06 Chasing references, expect 26 dots..........................

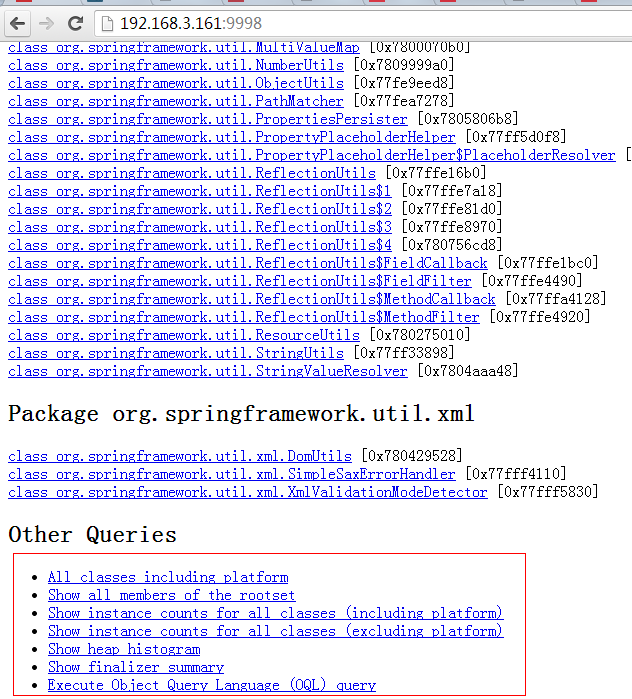
07 Eliminating duplicate references..........................

08 Snapshot resolved.

09 Started HTTP server on port 9998

10 Server is ready.

然后就可以在浏览器中输入主机地址:9998查看了：



    上面红线框出来的部分大家可以自己去摸索下，最后一项支持OQL（对象查询语言）。

### D、jstat（JVM统计监测工具）

语法格式如下：

1 **jstat [ generalOption | outputOptions vmid [interval[s|ms] [count]] ]**

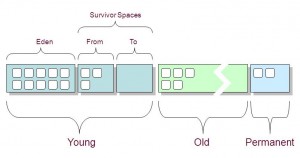
vmid是虚拟机ID，在Linux/Unix系统上一般就是进程ID。interval是采样时间间隔。count是采样数目。比如下面输出的是GC信息，采样时间间隔为250ms，采样数为4：

|  |  |
| --- | --- |
| 1 | root@ubuntu:/# jstat -gc 21711 250 4 |
| 2 | S0C    S1C    S0U    S1U      EC       EU        OC         OU       PC     PU    YGC     YGCT    FGC    FGCT     GCT |

|  |  |
| --- | --- |
| 3 | 192.0  192.0   64.0   0.0    6144.0   1854.9   32000.0     4111.6   55296.0 25472.7    702    0.431   3      0.218    0.649 |
| 4 | 192.0  192.0   64.0   0.0    6144.0   1972.2   32000.0     4111.6   55296.0 25472.7    702    0.431   3      0.218    0.649 |

|  |  |
| --- | --- |
| 5 | 192.0  192.0   64.0   0.0    6144.0   1972.2   32000.0     4111.6   55296.0 25472.7    702    0.431   3      0.218    0.649 |
| 6 | 192.0  192.0   64.0   0.0    6144.0   2109.7   32000.0     4111.6   55296.0 25472.7    702    0.431   3      0.218    0.649 |

要明白上面各列的意义，先看JVM堆内存布局：



    可以看出：

|  |  |
| --- | --- |
| 1 | 堆内存 = 年轻代 + 年老代 + 永久代 |
| 2 | 年轻代 = Eden区 + 两个Survivor区（From和To） |

    现在来解释各列含义：

|  |  |
| --- | --- |
| 1 | S0C、S1C、S0U、S1U：Survivor 0/1区容量（Capacity）和使用量（Used） |
| 2 | EC、EU：Eden区容量和使用量 |

|  |  |
| --- | --- |
| 3 | OC、OU：年老代容量和使用量 |
| 4 | PC、PU：永久代容量和使用量 |

|  |  |
| --- | --- |
| 5 | YGC、YGT：年轻代GC次数和GC耗时 |
| 6 | FGC、FGCT：Full GC次数和Full GC耗时 |

|  |  |
| --- | --- |
| 7 | GCT：GC总耗时 |

其他JVM性能调优参考资料：

《Java虚拟机规范》

《Java Performance》

《Trouble Shooting Guide for JavaSE 6 with HotSpot VM》: <http://www.oracle.com/technetwork/java/javase/tsg-vm-149989.pdf>

《Effective Java》

VisualVM: <http://docs.oracle.com/javase/7/docs/technotes/guides/visualvm/>

jConsole: <http://docs.oracle.com/javase/1.5.0/docs/guide/management/jconsole.html>

Monitoring and Managing JavaSE 6 Applications: <http://www.oracle.com/technetwork/articles/javase/monitoring-141801.html>

原文路径：[http://blog.csdn.net/wisgood/article/details/25343845](http://blog.csdn.net/wisgood/article/details/25343845" \t "_blank)

对应参数解释：

[各种 Java Thread State 第一分析法则](http://www.cnblogs.com/zhengyun_ustc/archive/2013/03/18/tda.html)

使用 [TDA 工具](http://java.net/projects/tda/)，看到大量 Java Thread State 的第一反应是：

**1，线程状态为“waiting for monitor entry”：**

意味着它 **在等待进入一个临界区**，所以它在”Entry Set“队列中等待。

此时线程状态一般都是 Blocked：

* java.lang.Thread.State: **BLOCKED** (on object monitor)

**2，线程状态为“waiting on condition”：**

说明它**在等待另一个条件的发生，来把自己唤醒**，或者干脆它是调用了 sleep(N)。

此时线程状态大致为以下几种：

* java.lang.Thread.State: **WAITING** (parking)：一直等那个条件发生；
* java.lang.Thread.State: **TIMED\_WAITING** (parking或sleeping)：定时的，那个条件不到来，也将定时唤醒自己。

**3，如果大量线程在“waiting for monitor entry”：**

可能是一个全局锁阻塞住了大量线程。

如果短时间内打印的 thread dump 文件反映，随着时间流逝，waiting for monitor entry 的线程越来越多，没有减少的趋势，可能意味着**某些线程在临界区里呆的时间太长了，以至于越来越多新线程迟迟无法进入临界区**。

**4，如果大量线程在“waiting on condition”：**

可能是它们又跑去获取第三方资源，**尤其是第三方网络资源，迟迟获取不到Response**，导致大量线程进入等待状态。

所以如果你发现有大量的线程都处在 Wait on condition，从线程堆栈看，正等待网络读写**，这可能是一个网络瓶颈的征兆**，因为网络阻塞导致线程无法执行。

**线程状态为“in Object.wait()”：**

说明它**获得了监视器之后，又调用了 java.lang.Object.wait() 方法**。

每个 Monitor在某个时刻，只能被一个线程拥有，该线程就是 “Active Thread”，而其它线程都是 “Waiting Thread”，分别在两个队列 “ Entry Set”和 “Wait Set”里面等候。在 “Entry Set”中等待的线程状态是 “Waiting for monitor entry”，而在 “Wait Set”中等待的线程状态是 “in Object.wait()”。

当线程获得了 Monitor，如果发现线程继续运行的条件没有满足，它则调用对象（一般就是被 synchronized 的对象）的 wait() 方法，放弃了 Monitor，进入 “Wait Set”队列。

此时线程状态大致为以下几种：

* java.lang.Thread.State: **TIMED\_WAITING** (on object monitor)；
* java.lang.Thread.State: **WAITING** (on object monitor)；

一般都是RMI相关线程（RMI RenewClean、 GC Daemon、RMI Reaper），GC线程（Finalizer），引用对象垃圾回收线程（Reference Handler）等系统线程处于这种状态。

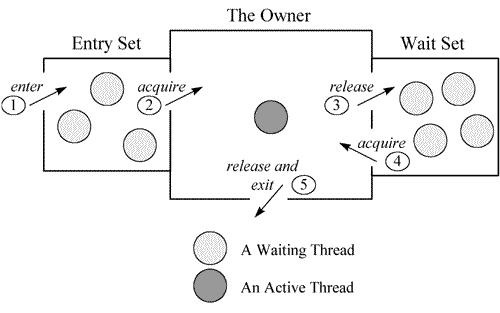


图1 A Java Monitor

示范一：

下面这个线程在等待这个锁 **0x00000000fe7e3b50**，等待进入临界区：

"RMI TCP Connection(64896)-172.16.52.118" daemon prio=10 tid=0x00000000405a6000 nid=0x68fe waiting for monitor entry [0x00007f2be65a3000]

   java.lang.Thread.State: BLOCKED (on object monitor)

at com.xyz.goods.service.impl.GoodsServiceImpl.**findChanellGoodsCountWithCache**(GoodsServiceImpl.java:1734)

- **waiting to lock <0x00000000fe7e3b50> (a java.lang.String)**

那么谁持有这个锁呢？

是另一个先调用了 findChanellGoodsCountWithCache 函数的线程：

"RMI TCP Connection(64878)-172.16.52.117" daemon prio=10 tid=0x0000000040822000 nid=0x6841 runnable [0x00007f2be76b3000]

   java.lang.Thread.State: RUNNABLE

at java.net.SocketInputStream.socketRead0(Native Method)

at java.net.SocketInputStream.read(SocketInputStream.java:129)

at java.io.BufferedInputStream.fill(BufferedInputStream.java:218)

at java.io.BufferedInputStream.read1(BufferedInputStream.java:258)

at java.io.BufferedInputStream.read(BufferedInputStream.java:317)

- locked <0x00000000af4ed638> (a java.io.BufferedInputStream)

at org.bson.io.Bits.readFully(Bits.java:35)

at org.bson.io.Bits.readFully(Bits.java:28)

at com.mongodb.Response.<init>(Response.java:35)

at com.mongodb.DBPort.go(DBPort.java:110)

- locked <0x00000000af442d48> (a com.mongodb.DBPort)

at com.mongodb.DBPort.go(DBPort.java:75)

- locked <0x00000000af442d48> (a com.mongodb.DBPort)

at com.mongodb.DBPort.call(DBPort.java:65)

at com.mongodb.DBTCPConnector.call(DBTCPConnector.java:202)

at com.mongodb.DBApiLayer$MyCollection.\_\_find(DBApiLayer.java:296)

at com.mongodb.DB.command(DB.java:152)

at com.mongodb.DBCollection.getCount(DBCollection.java:760)

at com.mongodb.DBCollection.getCount(DBCollection.java:731)

at com.mongodb.DBCollection.count(DBCollection.java:697)

at com.xyz.goods.manager.MongodbManager.count(MongodbManager.java:202)

at com.xyz.goods.service.impl.GoodsServiceImpl.findChanellGoodsCount(GoodsServiceImpl.java:1787)

at com.xyz.goods.service.impl.GoodsServiceImpl.**findChanellGoodsCountWithCache**(GoodsServiceImpl.java:1739)

- **locked <0x00000000fe7e3b50> (a java.lang.String)**

示范二：

等待另一个条件发生来将自己唤醒：

"RMI TCP Connection(idle)" daemon prio=10 tid=0x00007fd50834e800 nid=0x56b2 **waiting on condition** [0x00007fd4f1a59000]

   java.lang.Thread.State: **TIMED\_WAITING (parking)**

at sun.misc.Unsafe.park(Native Method)

- **parking to wait for  <0x00000000acd84de8>** (a java.util.concurrent.SynchronousQueue$TransferStack)

at java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:198)

at java.util.concurrent.SynchronousQueue$TransferStack.awaitFulfill(SynchronousQueue.java:424)

at java.util.concurrent.SynchronousQueue$TransferStack.transfer(SynchronousQueue.java:323)

at java.util.concurrent.SynchronousQueue.poll(SynchronousQueue.java:874)

at java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:945)

at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:907)

at java.lang.Thread.run(Thread.java:662)

1）“**TIMED\_WAITING (parking)**”中的 timed\_waiting 指等待状态，但这里指定了时间，到达指定的时间后自动退出等待状态；parking指线程处于挂起中。

2）“**waiting on condition**”需要与堆栈中的“**parking to wait for  <0x00000000acd84de8>** (a java.util.concurrent.SynchronousQueue$TransferStack)” 结合来看。首先，本线程肯定是在等待某个条件的发生，来把自己唤醒。其次，SynchronousQueue 并不是一个队列，只是线程之间移交信息的机制，当我们把一个元素放入到 SynchronousQueue 中时必须有另一个线程正在等待接受移交的任务，因此这就是本线程在等待的条件。

示范三：

"RMI RenewClean-[172.16.50.182:4888]" daemon prio=10 tid=0x0000000040d2c800 nid=0x97e **in Object.wait()** [0x00007f9ccafd0000]

   java.lang.Thread.State: **TIMED\_WAITING (on object monitor)**

at java.lang.Object.wait(Native Method)

- waiting on <0x0000000799b032d8> (a java.lang.ref.ReferenceQueue$Lock)

at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:118)

- locked <0x0000000799b032d8> (a java.lang.ref.ReferenceQueue$Lock)

at sun.rmi.transport.DGCClient$EndpointEntry$RenewCleanThread.run(DGCClient.java:516)

at java.lang.Thread.run(Thread.java:662)

**参考资源：**

1）CUBRID，2012，[How to Analyze Java Thread Dumps](http://www.cubrid.org/blog/dev-platform/how-to-analyze-java-thread-dumps/)；

2）郑昀，2013，[三个实例演示Java THread Dump日志分析](http://www.cnblogs.com/zhengyun_ustc/archive/2013/01/06/dumpanalysis.html)；

原文地址：<http://www.cnblogs.com/zhengyun_ustc/archive/2013/03/18/tda.html>