# JVM调优

## JVM Specification

整体结构图:



主要包括俩个子系统和俩个组件:Class Loader(类装载器子系统),Execution engine(执行引擎)子系统;Runtime data area(运行时数据区)组件,native interface(本地接口)组件

Class loader 子系统的作用：根据给定的全限定名类名(如

java.lang.Object)来装载class 文件的内容到Runtime data area 中的method

area(方法区域)。

Execution engine 子系统的作用：执行classes 中的指令。任何JVM

specification 实现(JDK)的核心是Execution engine， 换句话说：Sun 的JDK

和IBM 的JDK 好坏主要取决于他们各自实现的Execution engine 的好坏。每个

运行中的线程都有一个Execution engine 的实例。

Native interface组件:与native libaries 交互,是其他编程语言交互的接口.

Runtime data area 组件:这个组件就是jvm中的内存.

### Runtime data area 的整体架构图



Runtime data area主要分为五个部分:

Heap , Method area ,java stack , native stack , programe counter

Heap和method area是所有线程共享的,其他的都是以线程为粒度的.

### Heap:

Java 程序在运行时创建的所有类实或数组都放在同一个堆中。而一个Java 虚拟

实例中只存在一个堆空间，因此所有线程都将共享这个堆。每一个java 程序独

占一个JVM 实例，因而每个java 程序都有它自己的堆空间，它们不会彼此干扰。

但是同一java 程序的多个线程都共享着同一个堆空间，就得考虑多线程访问对

象（堆数据）的同步问题。（这里可能出现的异常java.lang.OutOfMemoryError:

Java heap space）

### Method Area

在Java 虚拟机中，被装载的class 的信息存储在Method area 的内存中。当虚

拟机装载某个类型时，它使用类装载器定位相应的class 文件，然后读入这个

class 文件内容并把它传输到虚拟机中。紧接着虚拟机提取其中的类型信息，并

将这些信息存储到方法区。该类型中的类（静态）变量同样也存储在方法区中。

与Heap 一样，method area 是多线程共享的，因此要考虑多线程访问的同步问

题。比如，假设同时两个线程都企图访问一个名为Lava 的类，而这个类还没有

内装载入虚拟机，那么，这时应该只有一个线程去装载它，而另一个线程则只能

等待。（这里可能出现的异常java.lang.OutOfMemoryError: PermGen full）

### java Stack

Java stack 以帧为单位保存线程的运行状态。虚拟机只会直接对Java

stack 执行两种操作：以帧为单位的压栈或出栈。每当线程调用一个方法的时候，

就对当前状态作为一个帧保存到java stack 中(压栈)；当一个方法调用返回时，

从java stack 弹出一个帧(出栈)。栈的大小是有一定的限制，这个可能出现

StackOverFlow 问题。

### Program counter

每个运行中的Java 程序，每一个线程都有它自己的PC 寄存器，也是该线程启动

时创建的。PC 寄存器的内容总是指向下一条将被执行指令的饿&ldquo;地址

&rdquo;，这里的&ldquo;地址&rdquo;可以是一个本地指针，也可以是在方法区

中相对应于该方法起始指令的偏移量。

### Native method stack

对于一个运行中的Java 程序而言，它还能会用到一些跟本地方法相关的数据区。

当某个线程调用一个本地方法时，它就进入了一个全新的并且不再受虚拟机限制

的世界。本地方法可以通过本地方法接口来访问虚拟机的运行时数据区，不止与

此，它还可以做任何它想做的事情。比如，可以调用寄存器，或在操作系统中分

配内存等。总之，本地方法具有和JVM 相同的能力和权限。(这里出现JVM 无法

控制的内存溢出问题native heap OutOfMemory )

## 基本概念

### 数据类型

Java虚拟机中，数据类型可以分为两类：**基本类型**和**引用类型**。基本类型的变量保存原始值，即：他代表的

值就是数值本身；而引用类型的变量保存引用值。“引用值”代表了某个对象的引用，而不是对象本身，对象

本身存放在这个引用值所表示的地址的位置。

基本类型包括：byte,short,int,long,char,float,double,Boolean,returnAddress

引用类型包括：**类类型**，**接口类型**和**数组**。

### 堆与栈

**栈是运行时的单位，而堆是存储的单位。**

栈解决程序的运行问题，即程序如何执行，或者说如何处理数据；堆解决的是数据存储的问题，即数据怎么

放、放在哪儿。

在Java中一个线程就会相应有一个线程栈与之对应，这点很容易理解，因为不同的线程执行逻辑有所不同，

因此需要一个独立的线程栈。而堆则是所有线程共享的。栈因为是运行单位，因此里面存储的信息都是跟当前

线程（或程序）相关信息的。包括局部变量、程序运行状态、方法返回值等等；而堆只负责存储对象信息。

### 问题:为什么要把堆和栈分开?栈中不是也可以存储数据？

第一，从软件设计的角度看，栈代表了处理逻辑，而堆代表了数据。这样分开，使得处理逻辑更为清晰。分

而治之的思想。这种隔离、模块化的思想在软件设计的方方面面都有体现。

第二，堆与栈的分离，使得堆中的内容可以被多个栈共享（也可以理解为多个线程访问同一个对象）。这种

共享的收益是很多的。一方面这种共享提供了一种有效的数据交互方式(如：共享内存)，另一方面，堆中的共享

常量和缓存可以被所有栈访问，节省了空间。

第三，栈因为运行时的需要，比如保存系统运行的上下文，需要进行地址段的划分。由于栈只能向上增长，

因此就会限制住栈存储内容的能力。而堆不同，堆中的对象是可以根据需要动态增长的，因此栈和堆的拆分，

使得动态增长成为可能，相应栈中只需记录堆中的一个地址即可。

第四，面向对象就是堆和栈的完美结合。其实，面向对象方式的程序与以前结构化的程序在执行上没有任何

区别。但是，面向对象的引入，使得对待问题的思考方式发生了改变，而更接近于自然方式的思考。当我们把

对象拆开，你会发现，对象的属性其实就是数据，存放在堆中；而对象的行为（方法），就是运行逻辑，放在

栈中。我们在编写对象的时候，其实即编写了数据结构，也编写的处理数据的逻辑。不得不承认，面向对象的

设计，确实很美。

### 堆中存什么？栈中存什么?

堆中存的是对象。栈中存的是基本数据类型和堆中对象的引用。一个对象的大小是不可估计的，或者说是可

以动态变化的，但是在栈中，一个对象只对应了一个4btye的引用（堆栈分离的好处：））。

为什么不把基本类型放堆中呢？因为其占用的空间一般是1~8个字节——需要空间比较少，而且因为是基本

类型，所以不会出现动态增长的情况——长度固定，因此栈中存储就够了，如果把他存在堆中是没有什么意义

的（还会浪费空间，后面说明）。可以这么说，基本类型和对象的引用都是存放在栈中，而且都是几个字节的

一个数，因此在程序运行时，他们的处理方式是统一的。但是基本类型、对象引用和对象本身就有所区别了，

因为一个是栈中的数据一个是堆中的数据。最常见的一个问题就是，Java中参数传递时的问题。

# Jvm监控

## JDK内置工具使用

### **jps**(Java Virtual Machine Process Status Tool)

查看所有的jvm进程，包括进程ID，进程启动的路径等等。

具体应用:

#### Jps

[root@master bin]# jps

#####列出所有的java进程

7024 JobTracker

551 Bootstrap

7655 Bootstrap

6761 NameNode

32718 Bootstrap

8268 Jps

1554 Bootstrap

#### Jps –l

[root@master ~]# jps -l

7024 org.apache.hadoop.mapred.JobTracker

551 org.apache.catalina.startup.Bootstrap

8379 sun.tools.jps.Jps

7655 org.apache.catalina.startup.Bootstrap

6761 org.apache.hadoop.hdfs.server.namenode.NameNode

32718 org.apache.catalina.startup.Bootstrap

1554 org.apache.catalina.startup.Bootstrap

### **jstack**(Java Stack Trace)

1观察jvm中当前所有线程的运行情况和线程当前状态。  
2系统崩溃了？如果java程序崩溃生成core文件，jstack工具可以用来获得core文件的java stack和native stack的信息，从而可以轻松地知道java程序是如何崩溃和在程序何处发生问题。  
3系统hung住了？jstack工具还可以附属到正在运行的java程序中，看到当时运行的java程序的java stack和native stack的信息, 如果现在运行的java程序呈现hung的状态，jstack是非常有用的。

#### Jstack {pid}

显示jvm中当前所有线程的运行情况和线程当前状态

[root@master ~]# jstack 1554 | more

2014-01-17 17:10:26

Full thread dump Java HotSpot(TM) 64-Bit Server VM (11.3-b02 mixed mode):

"Attach Listener" daemon prio=10 tid=0x000000004a2fe400 nid=0x20e5 runnable [0x0000000000000000..0x0000000000000000]

java.lang.Thread.State: RUNNABLE

"pool-2-thread-2" prio=10 tid=0x000000004a2f1800 nid=0x94c waiting on condition [0x0000000041673000..0x0000000041673d90]

java.lang.Thread.State: WAITING (parking)

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

- parking to wait for <0x00002aaab49339f0> (a java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject)

at java.util.concurrent.locks.LockSupport.park(LockSupport.java:158)

at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(AbstractQueuedSynchronizer.java:1925)

at java.util.concurrent.LinkedBlockingQueue.take(LinkedBlockingQueue.java:358)

at java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:947)

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

at java.lang.Thread.run(Thread.java:619)

"pool-2-thread-1" prio=10 tid=0x000000004980fc00 nid=0x8cb waiting on condition [0x0000000041111000..0x0000000041111c10]

java.lang.Thread.State: WAITING (parking)

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

- parking to wait for <0x00002aaab49339f0> (a java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject)

at java.util.concurrent.locks.LockSupport.park(LockSupport.java:158)

at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(AbstractQueuedSynchronizer.java:1925)

at java.util.concurrent.LinkedBlockingQueue.take(LinkedBlockingQueue.java:358)

at java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:947)

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

at java.lang.Thread.run(Thread.java:619)

"Heal-Session-Thread" prio=10 tid=0x00000000499c7000 nid=0x7ca waiting on condition [0x0000000041010000..0x0000000041010c90]

java.lang.Thread.State: WAITING (parking)

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

- parking to wait for <0x00002aaab4933708> (a java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject)

at java.util.concurrent.locks.LockSupport.park(LockSupport.java:158)

at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(AbstractQueuedSynchronizer.java:1925)

at java.util.concurrent.DelayQueue.take(DelayQueue.java:160)

at net.rubyeye.xmemcached.impl.MemcachedConnector$SessionMonitor.run(MemcachedConnector.java:107)

"Xmemcached-Reactor-3" prio=10 tid=0x000000004961e800 nid=0x7c9 runnable [0x00000000438a9000..0x00000000438a9b10]

java.lang.Thread.State: RUNNABLE

at sun.nio.ch.EPollArrayWrapper.epollWait(Native Method)

at sun.nio.ch.EPollArrayWrapper.poll(EPollArrayWrapper.java:215)

at sun.nio.ch.EPollSelectorImpl.doSelect(EPollSelectorImpl.java:65)

at sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:69)

- locked <0x00002aaab4933000> (a sun.nio.ch.Util$1)

- locked <0x00002aaab4932fe8> (a java.util.Collections$UnmodifiableSet)

- locked <0x00002aaab4932c70> (a sun.nio.ch.EPollSelectorImpl)

at sun.nio.ch.SelectorImpl.select(SelectorImpl.java:80)

at com.google.code.yanf4j.nio.impl.Reactor.run(Reactor.java:114)

"Xmemcached-Reactor-2" prio=10 tid=0x0000000049fac400 nid=0x7c8 runnable [0x00000000437a8000..0x00000000437a8b90]

java.lang.Thread.State: RUNNABLE

at sun.nio.ch.EPollArrayWrapper.epollWait(Native Method)

at sun.nio.ch.EPollArrayWrapper.poll(EPollArrayWrapper.java:215)

at sun.nio.ch.EPollSelectorImpl.doSelect(EPollSelectorImpl.java:65)

at sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:69)

- locked <0x00002aaab4932620> (a sun.nio.ch.Util$1)

- locked <0x00002aaab4932608> (a java.util.Collections$UnmodifiableSet)

### **jstat**(Java Virtual Machine Statistics Monitoring Tool)

1. jstat利用JVM内建的指令对Java应用程序的资源和性能进行实时的命令行的监控，包括了对进程的classloader，compiler，gc情况；  
       ②监视VM内存内的各种堆和非堆的大小及其内存使用量，以及加载类的数量。

具体使用:

#### jstat -class 1554 1000 10

[root@master ~]# jstat -class 1554 1000 10

Loaded Bytes Unloaded Bytes Time

5610 11692.9 306 346.2 5.02

5610 11692.9 306 346.2 5.02

5610 11692.9 306 346.2 5.02

5610 11692.9 306 346.2 5.02

5610 11692.9 306 346.2 5.02

5610 11692.9 306 346.2 5.02

5610 11692.9 306 346.2 5.02

5610 11692.9 306 346.2 5.02

5610 11692.9 306 346.2 5.02

5610 11692.9 306 346.2 5.02

|  |  |  |
| --- | --- | --- |
| ######################## 术语分隔符 ######################## | | |
|  | #Loaded 类加载数量 |

|  |  |  |
| --- | --- | --- |
|  | #Bytes  加载的大小（k） | |
|  | #Unloaded 类卸载的数量 |

|  |  |
| --- | --- |
|  | #Bytes 卸载的大小（k） |
|  | #Time 时间花费在执行类加载和卸载操作 | |
| jstat -gc 1554 2000 20 [root@master ~]# jstat -gc 1554 2000 20  S0C S1C S0U S1U EC EU OC OU PC PU YGC YGCT FGC FGCT GCT  6720.0 2688.0 0.0 2688.0 5632.0 2729.3 38080.0 27898.9 40128.0 39820.5 425433 3129.509 17258 2858.400 5987.909 |  | |

|  |  |  |
| --- | --- | --- |
| ######################## 术语分隔符 ######################## | | |
| 10 | #S0C 生还者区0 容量(KB) |

|  |  |
| --- | --- |
| 11 | #S1C 生还者区1 容量(KB) |
| 12 | #S0U 生还者区0 使用量(KB) | |

|  |  |  |
| --- | --- | --- |
| 13 | #S1U 生还者区1 使用量(KB) | |
| 14 | #EC 伊甸园区容量(KB) |

|  |  |  |
| --- | --- | --- |
| 15 | #EU 伊甸园区使用量(KB) | |
| 16 | #OC 老年区容量(KB) |

|  |  |
| --- | --- |
| 17 | #OU 老年区使用量(KB) |
| 18 | #PC 永久区容量(KB) |

|  |  |  |
| --- | --- | --- |
| 19 | #PU 永久区使用量(KB) | |
| 20 | #YGC 新生代GC次数 |

|  |  |
| --- | --- |
| 21 | #YGCT 新生代GC时间 |
| 22 | #FGC full GC 事件的次数 | |

|  |  |  |
| --- | --- | --- |
| 23 | #FGCT full GC的时间 | |
| 24 | #GCT 总GC时间 |

### **jmap**(Java Memory Map)

监视进程运行中的jvm物理内存的占用情况，该进程内存内，所有对象的情况，例如产生了哪些对象，对象数量；

|  |  |
| --- | --- |
| #参数 | |
|  | -dump:[live,]format=b,file=<filename> 使用hprof二进制形式,输出jvm的heap内容到文件=. live子选项是可选的，假如指定live选项,那么只输出活的对象到文件. | |

|  |  |
| --- | --- |
|  | -finalizerinfo 打印正等候回收的对象的信息. |
| 04 | -heap 打印heap的概要信息，GC使用的算法，heap的配置及wise heap的使用情况. | |

|  |  |
| --- | --- |
| 05 | -histo[:live] 打印每个class的实例数目,内存占用,类全名信息. VM的内部类名字开头会加上前缀”\*”. 如果live子参数加上后,只统计活的对象数量. |
| 06 | -permstat 打印classload和jvm heap长久层的信息. 包含每个classloader的名字,活泼性,地址,父classloader和加载的class数量. 另外,内部String的数量和占用内存数也会打印出来. |

|  |  |  |
| --- | --- | --- |
| 07 | -F 强迫.在pid没有相应的时候使用-dump或者-histo参数. 在这个模式下,live子参数无效. | |
| 08 | -h | -help 打印辅助信息 |

|  |  |
| --- | --- |
| 09 | -J 传递参数给jmap启动的jvm. |
| 10 | pid 需要被打印配相信息的java进程id. | |
|  | jmap -histo 1554 | more | |

[root@master ~]# jmap -histo 1554 | more

num #instances #bytes class name

----------------------------------------------

1: 80323 12358400 [C

2: 61963 8568848 <constMethodKlass>

3: 61963 7446008 <methodKlass>

4: 5304 6095680 <constantPoolKlass>

5: 94047 4803512 <symbolKlass>

6: 15094 4591184 [B

7: 5304 4083928 <instanceKlassKlass>

8: 85068 3402720 java.lang.String

9: 4273 3388992 <constantPoolCacheKlass>

10: 10057 1738440 [Ljava.util.HashMap$Entry;

11: 11345 1724440 java.lang.reflect.Method

12: 25287 1618368 java.util.LinkedHashMap$Entry

13: 5796 1066464 java.lang.Class

14: 21973 1054704 java.util.HashMap$Entry

15: 1650 894888 <methodDataKlass>

16: 8389 776656 [[I

17: 8291 649032 [Ljava.lang.Object;

18: 6335 606944 [I

19: 7609 585336 [S

20: 13358 427456 java.io.ExpiringCache$Entry

21: 4802 384160 java.util.LinkedHashMap

22: 6651 372456 java.lang.ref.SoftReference

23: 7220 346560 java.lang.ref.WeakReference

24: 5235 335040 java.util.HashMap

#### jmap -heap 551

[root@master ~]# jmap -heap 551

Attaching to process ID 551, please wait...

Debugger attached successfully.

Server compiler detected.

JVM version is 23.5-b02

using thread-local object allocation.

Parallel GC with 2 thread(s)

Heap Configuration:

MinHeapFreeRatio = 40

MaxHeapFreeRatio = 70

MaxHeapSize = 2147483648 (2048.0MB)

NewSize = 1048576 (1.0MB)

MaxNewSize = 4294901760 (4095.9375MB)

OldSize = 4194304 (4.0MB)

NewRatio = 2

SurvivorRatio = 8

PermSize = 268435456 (256.0MB)

MaxPermSize = 268435456 (256.0MB)

G1HeapRegionSize = 0 (0.0MB)

Heap Usage:

PS Young Generation

Eden Space:

capacity = 715653120 (682.5MB)

used = 34293152 (32.704498291015625MB)

free = 681359968 (649.7955017089844MB)

4.791867881467491% used

From Space:

capacity = 65536 (0.0625MB)

used = 0 (0.0MB)

free = 65536 (0.0625MB)

0.0% used

To Space:

capacity = 65536 (0.0625MB)

used = 0 (0.0MB)

free = 65536 (0.0625MB)

0.0% used

PS Old Generation

capacity = 1431699456 (1365.375MB)

used = 77844768 (74.23855590820312MB)

free = 1353854688 (1291.1364440917969MB)

5.4372283005184014% used

PS Perm Generation

capacity = 268435456 (256.0MB)

used = 102262504 (97.5251235961914MB)

free = 166172952 (158.4748764038086MB)

38.09575140476227% used

31609 interned Strings occupying 3389952 bytes.

# JVM调优实战

## 堆大小设置

JVM中最大堆大小有三方面限制：相关操作系统的数据模型（32-bt还是64-bit）限制；系统的可用虚拟内存限

制；系统的可用物理内存限制。32位系统下，一般限制在1.5G~2G；64为操作系统对内存无限制。在

Windows Server 2003 系统，3.5G物理内存，JDK5.0下测试，最大可设置为1478m。

## 典型设置

java **-Xmx3550m -Xms3550m -Xmn2g** –Xss128k

### -Xmx3550m:

设置jvm最大可用内存为3550M.

### -Xms3550m:

设置jvm初始内存为3550M,此值可以设置与-Xmx相同,以避免每次垃圾回收完成后jvm重新分配内存.

### -Xmng

设置年轻代为2G,整个堆大小=年轻代大小+年老代大小+持久代大小.持久代一般固定为64m,所以增大年轻代后将会减少年老代大小,此值对系统性能影响较大,Sun官方推荐为整个堆的

3/8

### -Xss128k

设置每个线程的堆栈大小,JDK5.0以后没个线程堆栈大小为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。

java -Xmx3550m -Xms3550m -Xss128k **-XX:NewRatio=4 -XX:SurvivorRatio=4 -**

**XX:MaxPermSize=16m -XX:MaxTenuringThreshold=0**

### -XX:NewRatio=4

设置年轻代(包括Eden和俩个Survivor区)与年老代的比值(除去持久代).设置为4,则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

### -XX:SurvivorRatio=4:

设置年轻代中Eden区与Survivor区的大小比值,设置为4,则俩个Survivor区与一个Eden区比值为2:4,一个Survivor区占整个年轻代的1/6

### -XX:MaxPermSize=16m:

设置持久代大小为16m

### -XX:MaxTenuringThreshold=0:

设置垃圾最大年龄,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比较大的值,则年轻代对象会在Survivor区进行多次复制,这样可以增加对象在年轻代的存活时间,增加在年轻代被回收的概率