Java常用命令工具

# 工具

## Gcview

通过GCView工具打开日志文件，查看整体的 GC 性能.

[gcviewer download | SourceForge.net](https://sourceforge.net/projects/gcviewer/)

## java8 元空间

元空间与永久代之间最大的区别在于：元空间并不在虚拟机中，而是使用本地内存。因此，默认情况下，元空间的大小仅受本地内存限制，但可以通过以下参数来指定元空间的大小。

JDK 8 中永久代向元空间的转换理由：

1、字符串存在永久代中，容易出现性能问题和内存溢出。

2、类及方法的信息等比较难确定其大小，因此对于永久代的大小指定比较困难，太小容易出现永久代溢出，太大则容易导致老年代溢出。

3、永久代会为 GC 带来不必要的复杂度，并且回收效率偏低。

**1. 老年代（Old Generation）：** - **老年代是 Java 堆的一部分，用于存储长生命周期的对象。**当对象在年轻代（Young Generation）中经过多次垃圾回收后，如果仍然存活，它们会被移动到老年代。 - 老年代的垃圾回收相对年轻代来说频率较低，通常采用标记-整理算法进行回收。

**2. 元空间（Metaspace）：** - **元空间是用于存储类的元数据的内存区域**。- 元空间的垃圾回收机制与老年代不同，它主要是通过监控类的加载和卸载来管理内存。

总结来说，老年代主要用于存储对象实例，而元空间则用于存储类的元数据。两者在内存管理和垃圾回收策略上有显著的区别。

## 基础知识

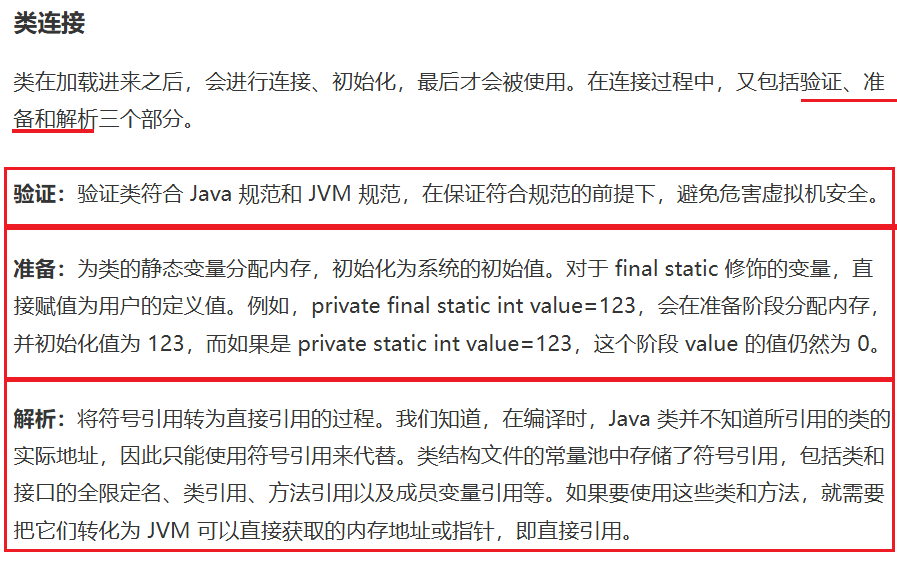
吞吐量 = CPU在用户应用程序运行的时间 / （CPU在用户应用程序运行的时间 + CPU垃圾回收的时间）

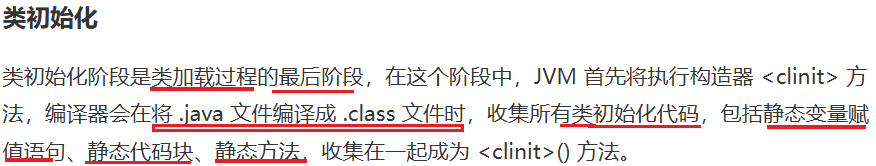
并发:一个处理器同时处理多个任务。

并行:多个处理器或者是多核的处理器同时处理多个不同的任务.

### 方法区

JVM 在执行某个类的时候，必须经过加载、连接、初始化，而连接又包括验证、准备、解析三个阶段。在加载类的时候，JVM 会先加载 class 文件，而在 class 文件中除了有类的版本、字段、方法和接口等描述信息外，还有一项信息是常量池 (Constant Pool Table)，用于存放编译期间生成的各种**字面量和符号引用**。





方法区用于存放已被虚拟机加载的类相关信息，包括类信息、运行时常量池、字符串常量池。

类信息又包括了类的版本、字段、方法、接口和父类等信息。包括类信息、常量、静态变量、即时编译器编译后的代码等。

### 运行时常量池

1. **定义**：运行时常量池是方法区的一部分，它包含了编译器生成的各种字面量（如文本字符串、数字、final变量等）和符号引用（如类、接口、字段、方法等）。
2. **作用**：它用于存储在编译阶段生成的各种符号引用和字面量，这些数据在运行时会被JVM使用。
3. **存储内容**：包括字符串字面量、数字字面量、true、false、null等。

### 字符串常量池

1. **定义**：字符串常量池是运行时常量池的一个子集，专门用于存储字符串字面量。
2. **作用**：它用于存储字符串字面量，以便于JVM进行字符串池化，提高字符串处理的效率。
3. **存储内容**：主要存储字符串字面量，如"Hello"、"World"等。

# Java8堆空间默认配置

**新生代 ( Young ) 与老年代 ( Old ) 的比例的值为 1:2**，即：新生代 ( Young ) = 1/3 的堆空间大小。

老年代 ( Old ) = 2/3 的堆空间大小。其中，新生代 ( Young ) 被细分为 Eden 和 两个 Survivor 区域，这两个 Survivor 区域分别被命名为 from 和 to，以示区分。

**默认的，Edem : from : to = 8 : 1 : 1** ，即： Eden = 8/10 的新生代空间大小，from = to = 1/10 的新生代空间大小。

‍JVM 每次只会使用 Eden 和其中的一块 Survivor 区域来为对象服务，所以无论什么时候，总是有一块 Survivor 区域是空闲着的

# GCEasy

https://gceasy.io/

GC日志分析是免费的，**Machine Learning**收费, 几秒内解决GC和内存问题

# jdk命令

## 1. javap-反编译工具

javap是jdk自带的反解析工具。它的作用就是根据class字节码文件，反解析出当前类对应的code区（汇编指令）、本地变量表、异常表和代码行偏移量映射表、常量池等等信息。

<https://www.cnblogs.com/frankcui/p/12134360.html>

常用命令选项

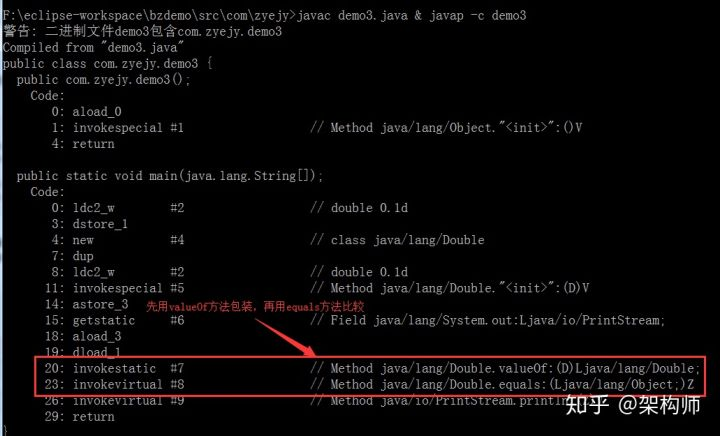
javap -v classxx，不仅会输出行号、本地变量表信息、反编译汇编代码，还会输出当前类用到的常量池等信息。

javap -l 会输出行号和本地变量表信息。

javap -c 会对当前class字节码进行反编译生成汇编代码。

案例

javac Test.java & javap -c Test



## 2. jstack-分析线程栈信息

**功能描述：**

Jstack【Java Stack Trace】是Java虚拟机自带的一种堆栈跟踪工具，用于生成java虚拟机当前时刻的线程快照。

线程快照是当前Java虚拟机内每一条线程正在执行的方法堆栈的集合，生成线程快照的主要目的是定位线程出现长时间停顿的原因，如线程间死锁、死循环、请求外部资源导致的长时间等待、等等。

**线程快照里留意下面几种状态**

**死锁， Deadlock （重点关注）**

**等待资源， Waiting on condition （重点关注）**

**等待获取管程，Waiting on monitor entry （重点关注）**

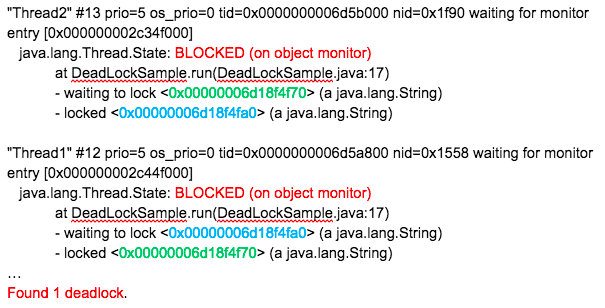
**阻塞， Blocked （重点关注）**

执行中， Runnable

暂停， Suspended

对象等待中， Object.wait() 或 TIMED\_WAITING

停止， Parked



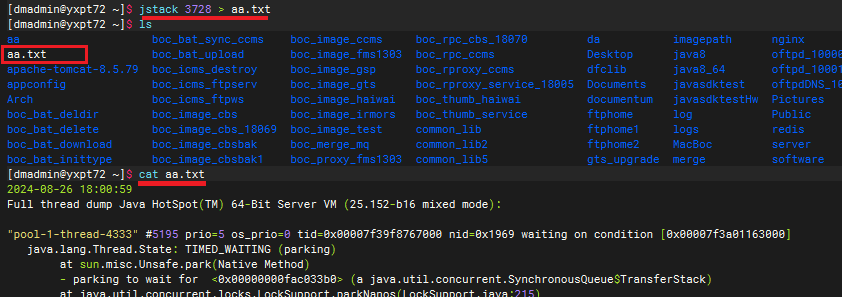
### 案例：导出栈文件

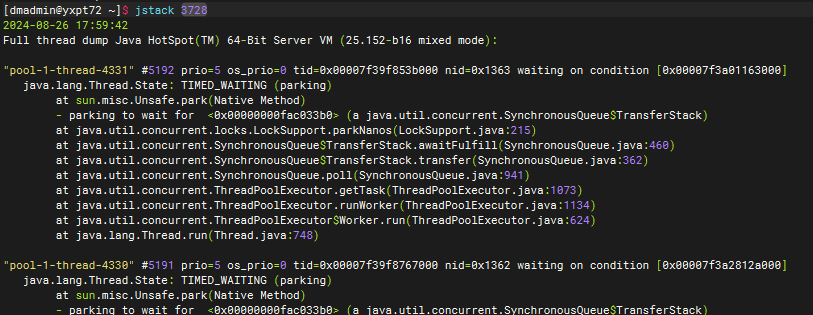
jstack [ option ] pid 查看当前时间点，指定进程的dump堆栈信息。

jstack [ option ] pid > 文件

将当前时间点的指定进程的dump堆栈信息，写入到指定文件中。

# 注:若该文件不存在，则会自动生成; 若该文件存在，则会覆盖源文件。





**可选参数说明-一般情况不需要使用**

**-F**  当进程挂起了，此时'jstack [-l] pid'是没有相应的，这时候可使用此参数来强制打印堆栈信息,强制jstack），一般情况不需要使用。

**-m** 打印java和native c/c++框架的所有栈信息。可以打印JVM的堆栈，以及Native的栈帧，一般应用排查不需要使用。

**-l** 长列表. 打印关于锁的附加信息。例如属于java.util.concurrent的ownable synchronizers列表，会使得JVM停顿得长久

得多（可能会差很多倍，比如普通的jstack可能几毫秒和一次GC没区别，加了-l 就是近一秒的时间），-l 建议不要用。**-h** 打印帮助信息

## 3.jmap堆信息查看

### 案例1.导出dump文件

命令： jmap -dump:format=b,file=heapdump.hprof pid

描述：生成堆转储快照dump文件：以二进制格式转储Java堆到指定文件中。如果指定了

live子选项，堆中只有活动的对象会被转储。浏览heap dump 可以使用jhat 读取

生成的文件，也可以使用MAT等堆内存分析工具。

注意：这个命令执行，JVM会将整个heap的信息dump写入到一个文件，heap如果比较

大的话，就会导致这个过程比较耗时，并且执 行的过程中为了保证dump的信息是

可靠的，所以会暂停应用， 线上系统慎用！

操作：

jmap -dump:file=javaDump.hprof,format=b 进程号



将dump中存活的对象导出，那么可以使用:live参数

jmap -dump:live,format=b,file=javaDump.hprof

hprof（Heap Profile 的缩写）堆快照

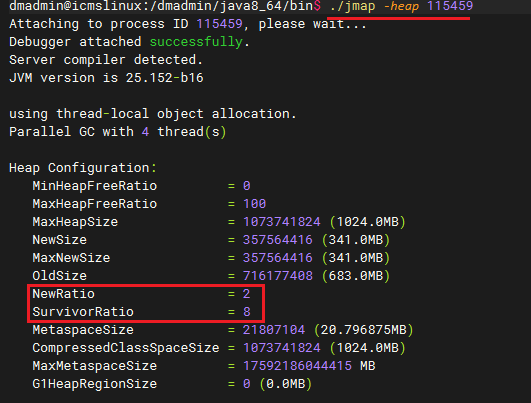
### 案例2.查看堆详细信息

命令：jmap -heap pid

描述：显示Java堆详细信息：打印堆的摘要信息，包括使用的GC算法、堆配置信息和各内

存区域内存使用信息

jmap -heap 115459



在 JDK1.8 中，默认情况下

年轻代和老年代的比例是 1:2。参数：–XX:NewRatio

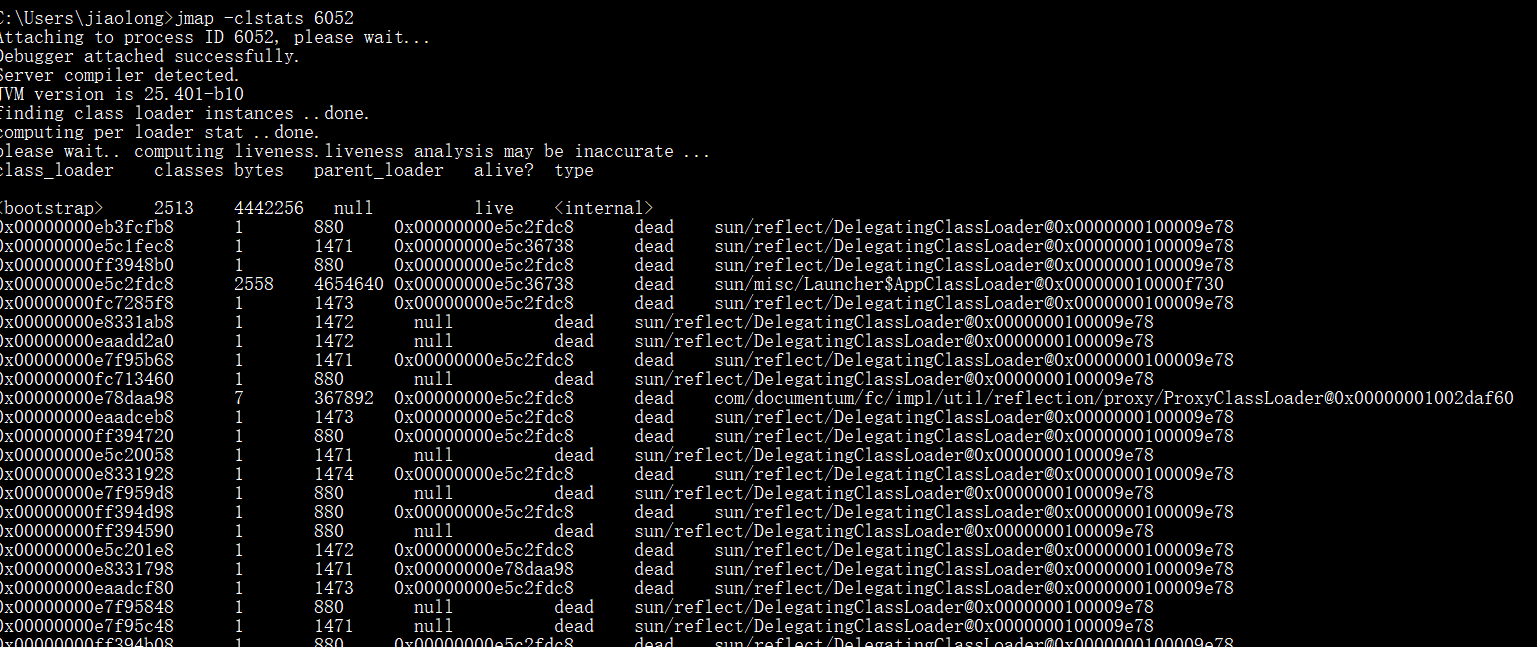
年轻代中的**Eden和Survivor的比例**是 8:2。 参数：-XX:SurvivorRatio

多版本java需要指定绝对路径



[sun.jvm.hotspot.runtime.VMVersionMismatchException](https://www.cnblogs.com/erlou96/p/14657174.html)

* **jmap -clstats**：查看类加载统计信息（JDK 8及以上支持）
* jmap -clstats <pid>
* 输出会包含类名、类加载器、实例数量、元数据大小等，重点关注space列（类元数据占用空间）。



## 4.jstat查看各空间使用情况

作用： jstat可以查看Java程序运行时相关信息，可以通过它查看运行时堆信息的相关情况。

语法：jstat -<options> [-t] [-h<lines>] <vmid> [<interval> [<count>]]

options：由以下值构成

-class：显示ClassLoader的相关信息

-compiler：显示JIT编译的相关信息

-gc：显示与GC相关信息

-gccapacity：显示各个代的容量和使用情况

-gccause：显示垃圾收集相关信息（同-gcutil），同时显示最后一次或当前正在发生

的垃圾收集的诱发原因

-gcnew：显示新生代信息

-gcnewcapacity：显示新生代大小和使用情况

-gcold：显示老年代信息

-gcoldcapacity：显示老年代大小

-gcpermcapacity：显示永久代大小

-gcutil：显示垃圾收集信息

### 案例1：gc-显示与GC相关信息

命令：jstat -gc 30108 250 4 【进程ID 30108，采样间隔250ms，采样数4】

说明：进程内存区域及GC详细信息。



S0C：年轻代中第一个survivor（幸存区）的容量 （单位kb）

S1C：年轻代中第二个survivor（幸存区）的容量 (单位kb)

S0U ：年轻代中第一个survivor（幸存区）目前已使用空间 (单位kb)

S1U ：年轻代中第二个survivor（幸存区）目前已使用空间 (单位kb)

EC ：年轻代中Eden的容量 (单位kb)

EU ：年轻代中Eden目前已使用空间 (单位kb)

OC ：Old代的容量 (单位kb)

OU ：Old代目前已使用空间 (单位kb)

MC：metaspace的容量 (单位kb)

MU：metaspace目前已使用空间 (单位kb)

CCSC：压缩类空间大小

CCSU：压缩类空间使用大小

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

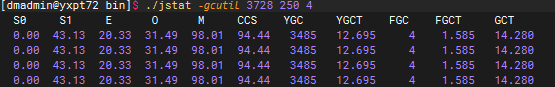
FGCT ：从应用程序启动到采样时old代(全gc)gc所用时间(s)

GCT：从应用程序启动到采样时gc用的总时间(s)

### 案例2：-gcutil：显示垃圾收集信息

命令：jstat -gcutil 30108 1s 5 【进程ID 30108，采样间隔1s，采样数5】

说明：进程内存区域百分比及GC详细信息



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

M metaspace已使用的占当前容量百分比

CCS 压缩使用比例

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

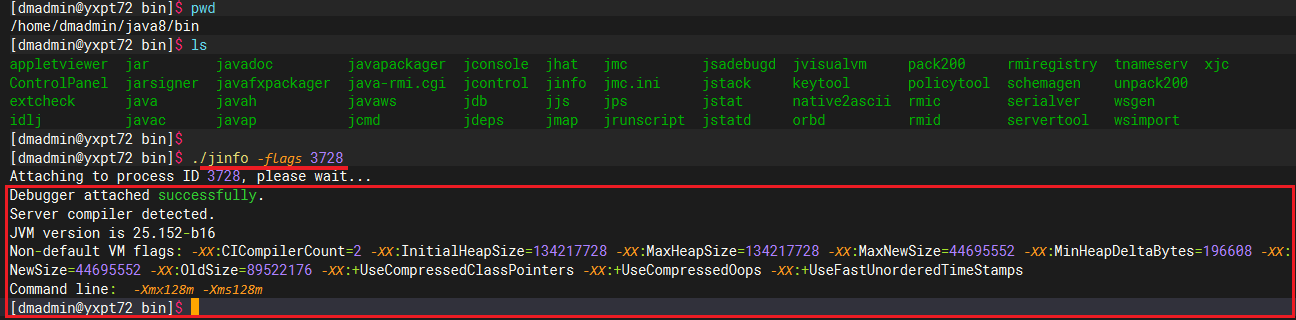
FGCT 从应用程序启动到采样时old代(全gc)gc所用时间(s)

GCT 从应用程序启动到采样时gc用的总时间(s)

## 5.jinfo-查看vm运行参数

查看正在运行的Java程序的虚拟机参数.

命令jinfo -flags pid



## 6.jps查看Java进程

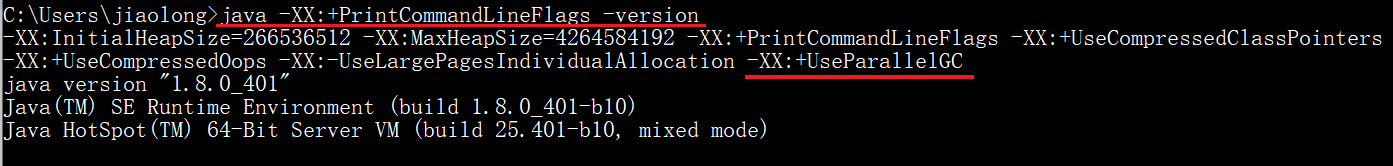
jps -m ：输出传递给Java进程（主函数）的参数

jps -l ：输出主函数的完整路径

jps -v ：显示传递给Java虚拟机的参数

## 7.查看默认GC

java -XX:+PrintCommandLineFlags -version



# JVM常用汇总

## GC日志作用

GC日志是Java虚拟机（JVM）在运行过程中输出的关于垃圾回收（GC）的详细信息。这些日志对于Java应用的性能调优和问题诊断至关重要，例如：

3.调整堆内存大小：如果GC日志显示频繁的Full GC，可能需要增加堆内存大小。

4.调整年轻代大小：合理调整年轻代大小可以减少Minor GC的频率和停顿时间。

## Dump分类

### heap dump

heap dump文件是一个二进制文件，它保存了某一时刻JVM堆中对象使用情况。HeapDump文件是指定时刻的Java堆栈的快照，是一种镜像文件。Heap Analyzer工具通过分析HeapDump文件，哪些对象占用了太多的堆栈空间，来发现导致内存泄露或者可能引起内存泄露的对象。

### thread dump

thread dump文件主要保存的是java应用中各线程在某一时刻的运行的位置，即执行到哪一个类的哪一个方法哪一个行上。thread dump是一个文本文件，打开后可以看到每一个线程的执行栈，以stacktrace的方式显示。通过对thread dump的分析可以得到应用是否“卡”在某一点上，即在某一点运行的时间太长，如数据库查询，长期得不到响应，最终导致系统崩溃。单个的thread dump文件一般来说是没有什么用处的，因为它只是记录了某一个绝对时间点的情况。比较有用的是，线程在一个时间段内的执行情况。

两个thread dump文件在分析时特别有效，困为它可以看出在先后两个时间点上，线程执行的位置，如果发现先后两组数据中同一线程都执行在同一位置，则说明此处可能有问题，因为程序运行是极快的，如果两次均在某一点上，说明这一点的耗时是很大的。通过对这两个文件进行分析，查出原因，进而解决问题。

原文链接：<https://blog.csdn.net/songjianlong/article/details/132179955>

### 创建dump文件

**获取heap dump文件**

jmap -dump:format=b,file=heap.hprof 2576

如果我们只需要将dump中存活的对象导出，那么可以使用:live参数

jmap -dump:live,format=b,file=heapLive.hprof 2576

**获取thread dump文件**

jstack 2576 > thread.txt

**自动生成heap dump文件**

-XX:+HeapDumpOnOutOfMemoryError [OutOfMemoryError时自动生成Heap Dump文件]

-XX:+HeapDumpBeforeFullGC [FullGC 前创建 dump]

-XX:+HeapDumpAfterFullGC [FullGC 后创建 dump]

-XX:HeapDumpPath=d:\dump.hprof [dump文件保存路径]

## Minor GC、Major GC与Full GC 区别？

**Minor GC（年轻代垃圾回收）**

主要作用：Minor GC负责清理年轻代（通常包括Eden空间和两个Survivor空间）中的垃圾对象。

内存区域：主要涉及年轻代的Eden空间和Survivor空间。

**Major GC（老年代垃圾回收）**

定义与特点：

主要作用：Major GC负责清理老年代中的垃圾对象。

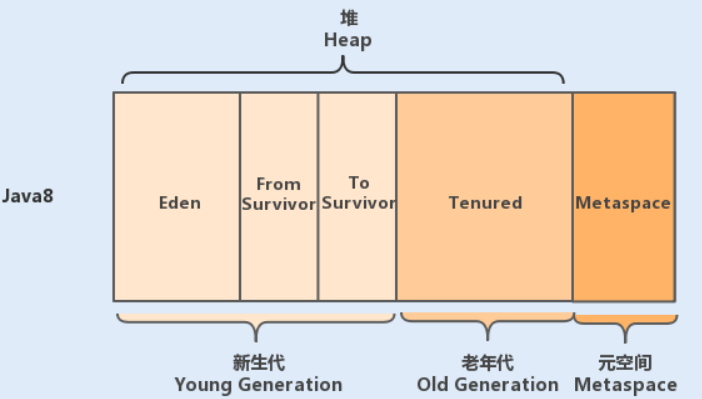
内存区域：主要涉及老年代。

**Full GC（完全垃圾回收）**

主要作用：Full GC负责清理**整个Java堆**（年轻代和老年代）以及到元空间（Metaspace）中的垃圾对象。

内存区域：涉及整个Java堆和元空间

## Heap内存区域



### JVM设置Eden、S0、S1的理由

#### 1 为什么要有Survivor区？

设置Survivor区的意义在哪里？如果没有Survivor，Eden区每进行一次Minor GC，存活的对象就会被送到老年代。老年代很快被填满，触发Major GC（因为Major GC一般伴随着Minor GC，也可以看做触发了Full GC）。老年代的[内存](https://so.csdn.net/so/search?q=%E5%86%85%E5%AD%98&spm=1001.2101.3001.7020)空间远大于新生代，进行一次Full GC消耗的时间比Minor GC长得多。频发的Full GC消耗的时间是非常可观的，这一点会影响程序的执行和响应速度。

Survivor的存在意义：**就是减少被送到老年代的对象，进而减少Full GC的发生**，Survivor的预筛选保证，只有经历16次Minor GC还能在新生代中存活的对象，才会被送到老年代。

#### 2 为什么要设置两个Survivor区？

设置两个Survivor区最大的好处就是解决了碎片化，为什么一个Survivor区不行？

假设现在只有一个survivor区：刚刚新建的对象在Eden中，一旦Eden满了，触发一次Minor GC，Eden中的存活对象就会被移动到Survivor区。这样继续循环下去，下一次Eden满了的时候，问题来了，此时进行Minor GC，Eden和Survivor各有一些存活对象，如果此时把Eden区的存活对象硬放到Survivor区，很明显这两部分对象所占有的内存是不连续的，也就导致了内存碎片化。碎片化带来的风险是极大的，堆中没有足够大的连续内存空间，无法为一个内存需求很大的对象分配内存。

所以应该建立两块Survivor区，刚刚新建的对象在Eden中，经历一次Minor GC，Eden中的存活对象就会被移动到S0，Eden被清空；等Eden区再满了，就再触发一次Minor GC，Eden和S0中的存活对象又会被复制**送入第二块S1**（这个过程非常重要，因为这种**复制算法保证了S1**中来自S0和Eden两部分的存活对象占用连续的内存空间，避免了碎片化的发生）。

### 降低 Minor GC 频率

由于新生代空间较小，Eden区很快被填满，就会导致频繁 Minor GC，因此我们可以通过增大新生代空间来降低Minor GC 的频率。

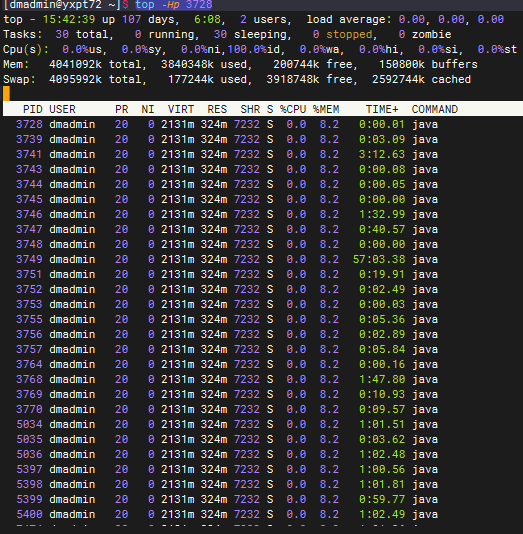
单次 Minor GC 时间是由两部分组成：T1（扫描新生代）和 T2（复制存活对象）。假设一个对象在 Eden 区的存活时间为 500ms，Minor GC 的时间间隔是 300ms，那么正常情况下，Minor GC 的时间为 ：T1+T2。在虚拟机中，复制对象的成本要远高于扫描成本。

如果在堆内存中存在较多的**长期存活**的对象，此时增加年轻代空间，反而会增加 Minor GC 的时间。如果堆中的短期对象很多，那么扩容新生代，单次 Minor GC 时间不会显著增加。因此，单次 Minor GC 时间更多取决于 GC 后存活对象的数量，而非 Eden 区的大小。

# 查看上下文切换

## top查看进程

top -Hp pid 查看具体线程使用系统资源情况.



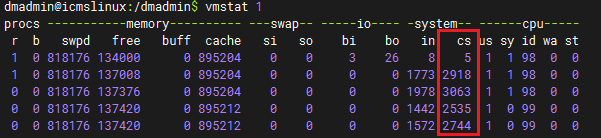
## vmstat查看资源使用率

vmstat是Virtual Meomory Statistics（虚拟内存统计）的缩写，可对操作系统的虚拟内存、进程、CPU活动进行监控。是对系统的整体情况进行统计。

命令：vmstat 延迟 统计次数

Eg: vmstat 1 3

这里的 1 表示每秒刷新一次统计信息，统计3次。查看cs这一列



**r**：等待运行的进程数； bo：从块设备接收到的块数；

b：处于非中断睡眠状态的进程数； in：每秒中断数；

swpd：虚拟内存使用情况； cs：每秒上下文切换次数；

free：空闲的内存； us：用户 CPU 使用时间；

buff：用来作为缓冲的内存数； sy：内核 CPU 系统使用时间?

si：从磁盘交换到内存的交换页数量； id：空闲时间；

so：从内存交换到磁盘的交换页数量； wa：等待 I/O 时间；

bi：发送到块设备的块数； st：运行虚拟机窃取的时间。

## pidstat查看进程上下文切换

pidstat是sysstat工具的一个命令，用于监控全部或指定进程的cpu、内存、线程、设备IO等系统资源的占用情况



常用的参数：

-u：默认的参数，显示各个进程的cpu使用统计

-r：显示各个进程的内存使用统计

-d：显示各个进程的IO使用情况

-p：指定进程号

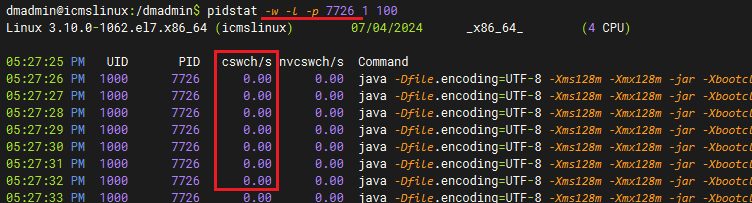
-w：显示每个进程的上下文切换情况

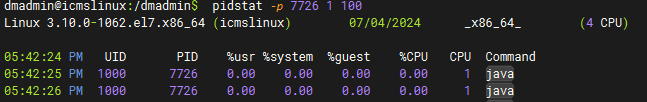
-t：显示选择任务的线程的统计信息外的额外信息

-h：在一行上显示了所有活动，这样其他程序可以容易解析。

-l：显示命令名和所有参数

**pidstat -w -l -p 7726 1 100**





# 参考：

1. [Java中JVM常用参数配置示](https://blog.csdn.net/weixin_44606481/article/details/136038078)
2. [Java 基础 - javap 反编译命令](https://www.cnblogs.com/frankcui/p/12134360.html)