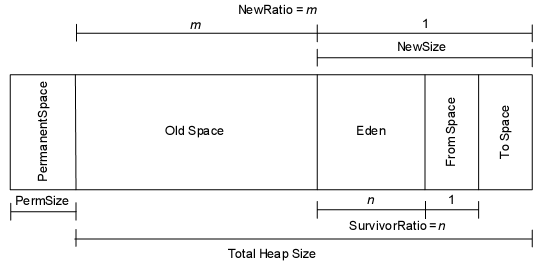
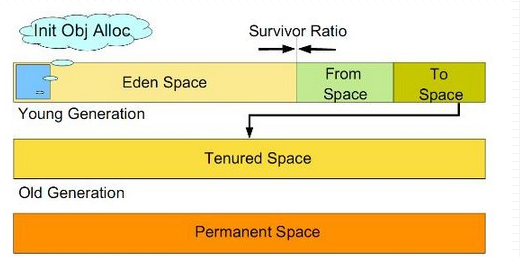
**内存区域**



1. Eden Space：对象被创建的时候首先放到这个区域。
2. Survivor Space：在eden space中经过GC后没有被回收的对象。
3. Minor GC(Young GC)：复制-清除算法。在GC开始的时候，对象只会存在于Eden区和名为“From”的Survivor区，Survivor区“To”是空的。紧接着进行GC，Eden区中所有存活的对象都会被复制到“To”，而在“From”区中，仍存活的对象会根据他们的年龄值来决定去向。年龄达到一定值(年龄阈值，可以通过-XX:MaxTenuringThreshold来设置)的对象会被移动到年老代中，没有达到阈值的对象会被复制到“To”区域。经过这次GC后，Eden区和From区已经被清空。这个时候，“From”和“To”会交换他们的角色，也就是新的“To”就是上次GC前的“From”，新的“From”就是上次GC前的“To”。不管怎样，都会保证名为To的Survivor区域是空的。Minor GC会一直重复这样的过程，直到“To”区被填满，“To”区被填满之后，会将所有对象移动到年老代中。
4. Old Space：在年轻代中经过多次GC后没有被回收的对象或大对象直接进入老年代。
5. Major GC(Full GC)：整个堆(young generation + old generation)扫描回收
6. Permanent Space：JDK8的metaspace in native memory
7. 对于操作系统，进程占用的内存的组成部分：

* heap = young + old
* metaspace
* thread stack
* direct memory

**Garbage Collection(heap)**

1. 引用类型

强引用StrongReference：常用的，只要引用，永不GC

软引用SoftReference：内存溢出异常时(OOM)，才纳入GC

弱引用WeakReference：只能生存至下次GC之前

虚引用PhantomReference：被GC时，收到一个系统通知

1. 回收标记策略

a. 引用计数

问题：循环引用，被引用次数一直非零，导致不标记回收

a.next = b;

b.next = a;

a = null;

b = null;

b. 可到达性分析(reachability analysis)

以GC root为起点，能遍历到的节点，不标记回收；不可到达的节点，标记回收

1. 内存清除策略

a. 标记-清除

问题：内存空间碎片化，难以为大对象分配连续的空间

b. 复制-清除

划分区域A和B，只使用A，若对A进行GC，则把A中存活的复制至B，清除整个A

问题：存活率高的场景，效率降低

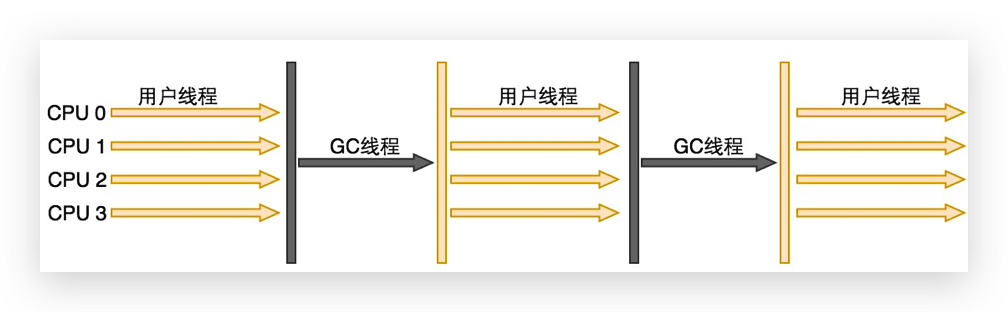
c. 标记-整理

d. 分代

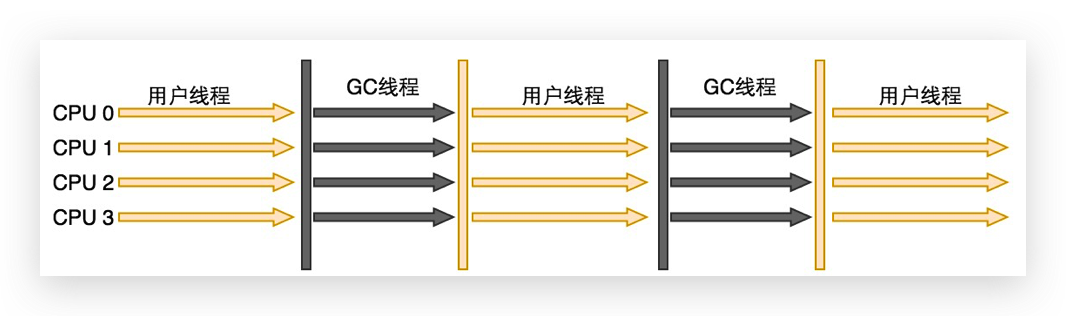
根据不同场景，对应不同策略。如：新生代、老年代

1. Collectors

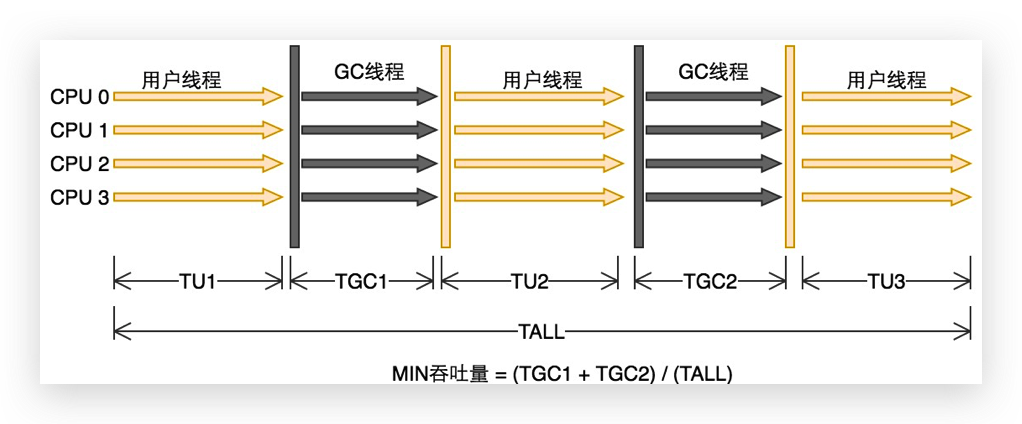
* Serial



* ParNew



* Parallel Scavenge

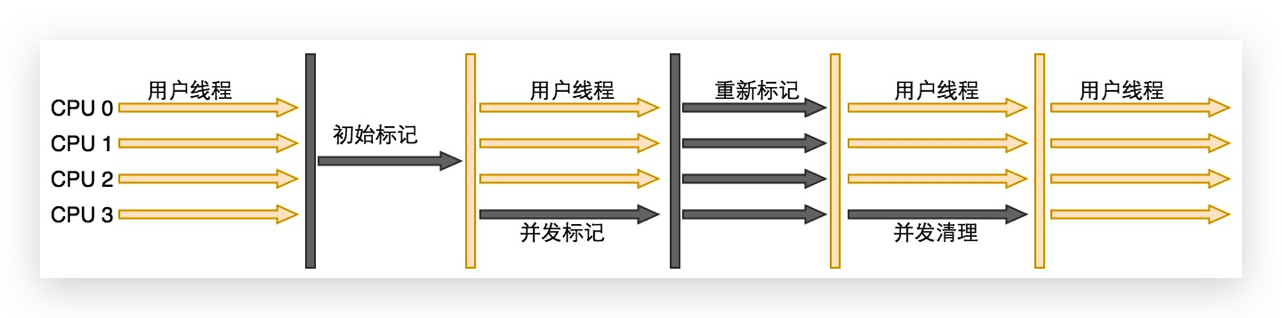


-XX:ParallelGCThreads 并行的GC线程数

-XX:MaxGCPauseMillis 只是提示，最大停顿时间

-XX:GCTimeRatio 指定GC时间比例，吞吐率=N/(1+N)

* Concurrent Mark Sweep



a. CPU占用率: 用户线程与GC线程并行，可能会占用较多CPU时间，影响应用系统的吞吐率和响应时间。

b. Concurrent Mode Failure: 在清理阶段中，清理线程和用户线程是并行的。如果用户线程产生了垃圾对象，但过了标记阶段，那么这些垃圾对象就成了浮动垃圾。CMS无法在当前GC过程中处理这些浮动垃圾。因此CMS收集器不能像其他收集器那样等到完全填满了老年代以后才进行垃圾收集，需要预留一部分空间来保证当出现浮动垃圾的时候可以有空间存放这些垃圾对象。如果在CMS运行期间没有足够的内存来存放浮动垃圾，那么就会导致"Concurrent Mode Failure"异常，这个时候，虚拟机将启动后备预案，临时启动Serial Old收集器来对老年代重新进行垃圾收集，这样会导致垃圾收集的时间边长，特别是当老年代内存很大的时候。

-XX:CMSInitiatingOccupancyFraction: 过高则降低GC次数但容易发生Concurrent Mode Failure，过低则浪费内存空间。

c. 内存碎片: 标记-清除算法的问题，必要时压缩整理，但会延长停顿时间。

* Garbage-First
* Z Garbage Collector (jdk11)

**ByteCode(.class)**

1. 数据：没有间隔符的无符号数表格(二进制字节流)

|  |  |
| --- | --- |
| Magic Number | 被JVM识别的标记 |
| minor version | jdk次版本号 |
| major version | jdk主版本号 |
| constant pool | 常量池：包括literal和symbolic references |
| access\_flag | 访问标志，区分ENUM/CLASS/INTERFACE/ABSTRACT等 |
| this\_class | 当前类索引 |
| super\_class | 父类索引 |
| interfaces | 接口索引列表 |
| field\_info | 变量字段表：access\_flags、name\_index、descriptor\_index、attributes\_count、attributes |
| method\_info | 方法表：同上 |
| attribute\_info | 属性表：  Code代码区  LineNumberTable源码行号与字节码行号的映射表  LocalVariableTable局部变量与源码的映射表  Signature类型擦除实现伪泛型的参数 |

2. 指令集：逻辑运算、类型转换、对象创建与访问、方法调用与返回、读写操作数栈、流程控制、异常处理、线程同步等

**Class Loader**

**基本流程**

加载、验证、准备、解析、初始化、使用、卸载

1. loading

a. 通过全限定名获取类的二进制字节流，如: class, jar, war等

b. 二进制字节流->运行时数据结构

c. 在内存中生成这个类的java.lang.class对象，作为该类的访问入口

2. verification：各种检查

3. preparation：类变量分配内存，设置默认值为0，如：static

4. resolution：symbolic reference(内存无关) -> direct reference(内存相关)

5. initialization

**类别**

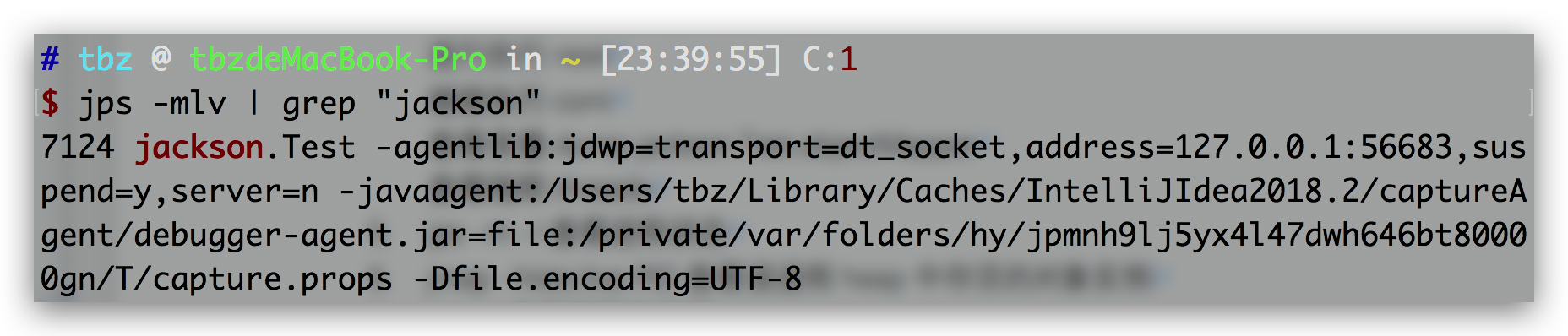
1. boostrap class loader：JVM必需的rt.jar的java.lang/java.io等
2. extension class loader：标准扩展目录(-Djava.ext.dir=)下的jar包等
3. system class loader：环境变量CLASSPATH下的jar包等

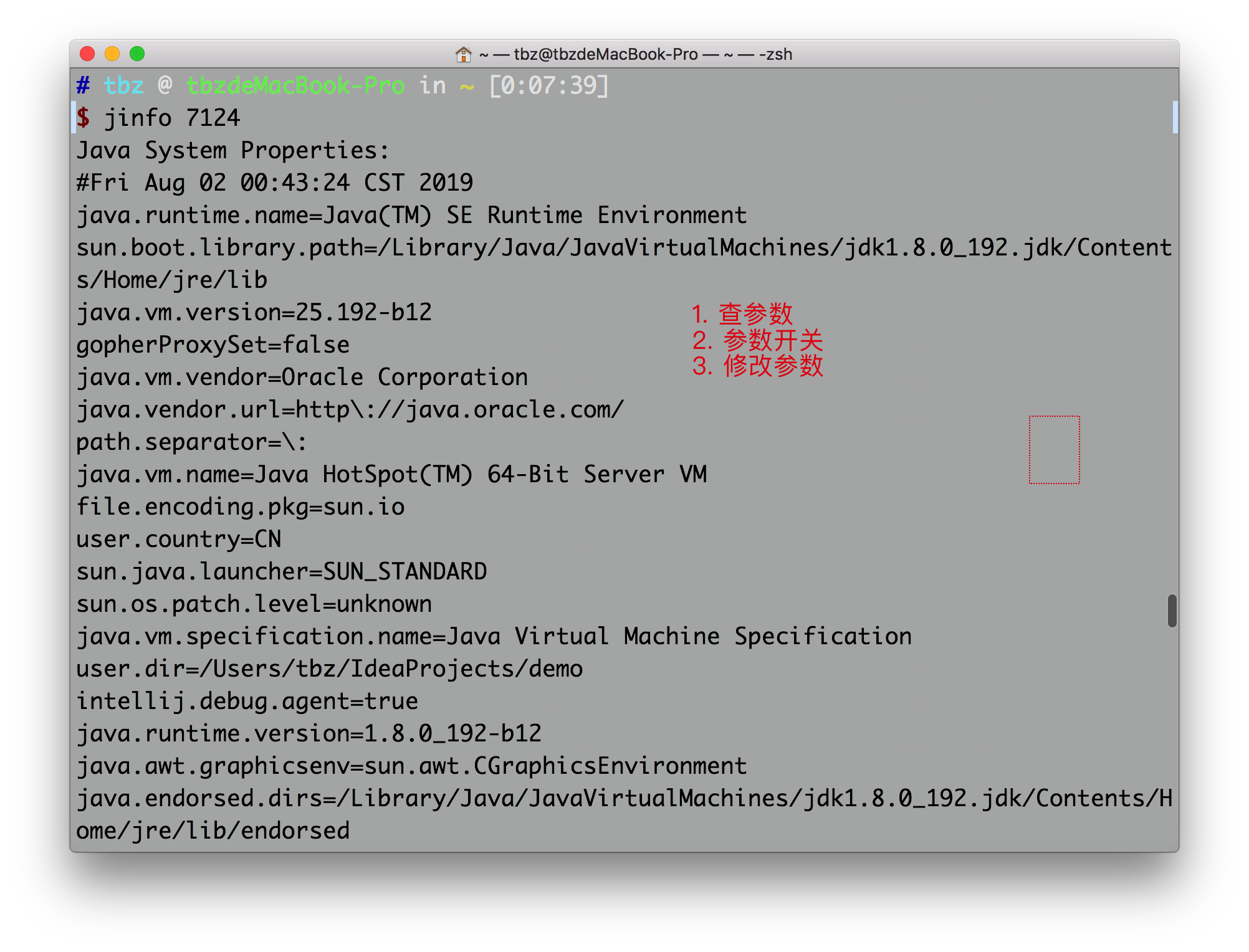
The Java platform uses a delegation model for loading classes. The basic idea is that every class loader has a "parent" class loader. When loading a class, a class loader first "delegates" the search for the class to its parent class loader before attempting to find the class itself.委托父加载器尝试加载，若失败，自己加载，再失败，ClassNotFoundException.

如：避免用户自定义的java.lang.Object覆盖rt.jar的java.lang.Object

**采集工具**

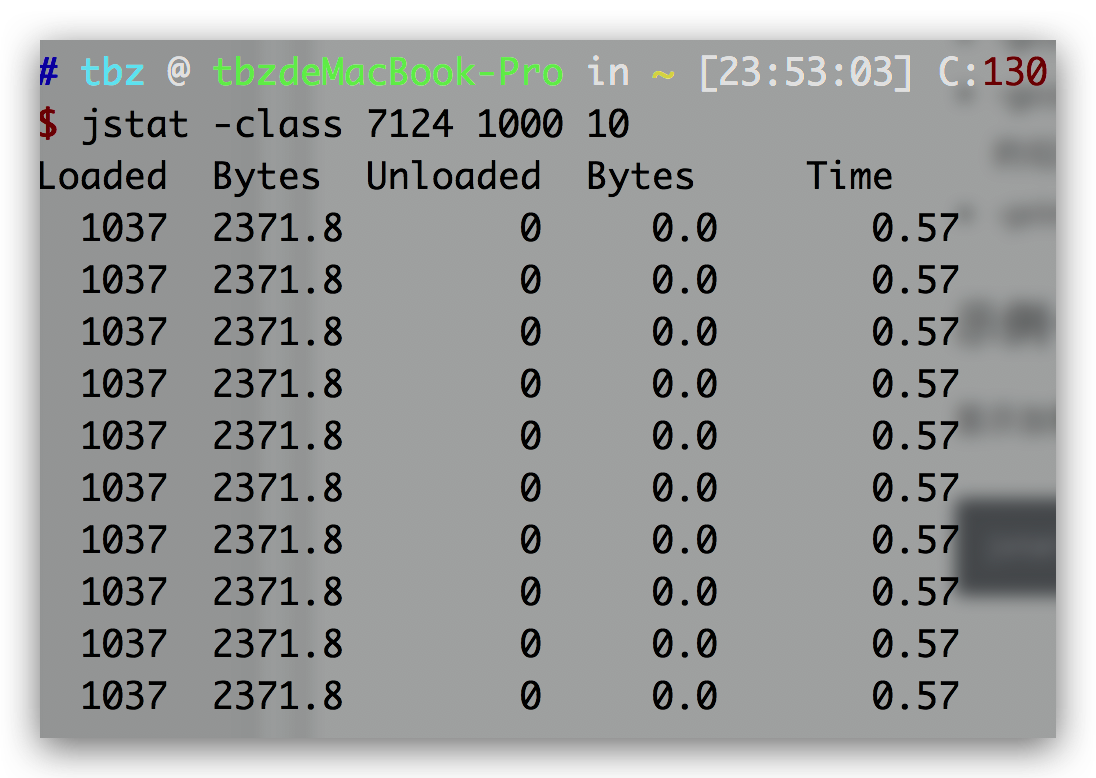
1. jps -mlv {pid}查看进程信息



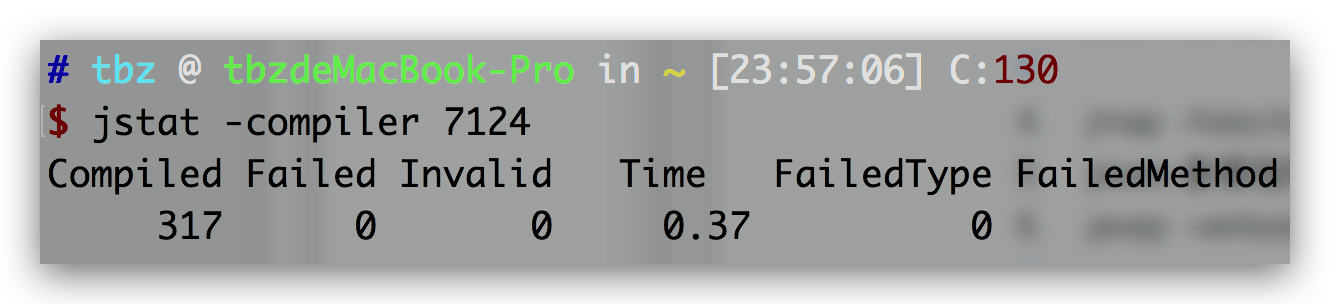


|  |  |
| --- | --- |
| **Jinfo** | |
| -Xms | min heap capacity |
| -Xmx | max heap capacity |
| -Xmn | young generation capacity |
| -XX:MinHeapFreeRatio | 最小空闲比例，当空闲比例<MinHeapFreeRatio，扩容至Xmx |
| -XX:MaxHeapFreeRatio | 最大空闲比例，当空闲比例>MaxHeapFreeRatio，收缩至Xms |
| -Xss | per thread stack capacity |
| -XX:NewRatio | old generation占比=a，则Young:Old=1:a |
| -XX:SurvivorRatio | eden space占比=b，则Survivor:Survivor:Eden=1:1:b |
| -XX:MaxTenuringThreshold | 设置新生代晋升到老年代的年龄阈值，并行收集器默认15，CMS收集器默认为6。由于该值用4个bit存储在JAVA对象头中，故该选项的最大值为15。 |
| -XX:+Scavenge BeforeFullGC | 执行Full GC前首先执行新生代的GC，因为在Full GC前对新生代执行Scavenge GC可以减少晋升到老年代的对象。 |

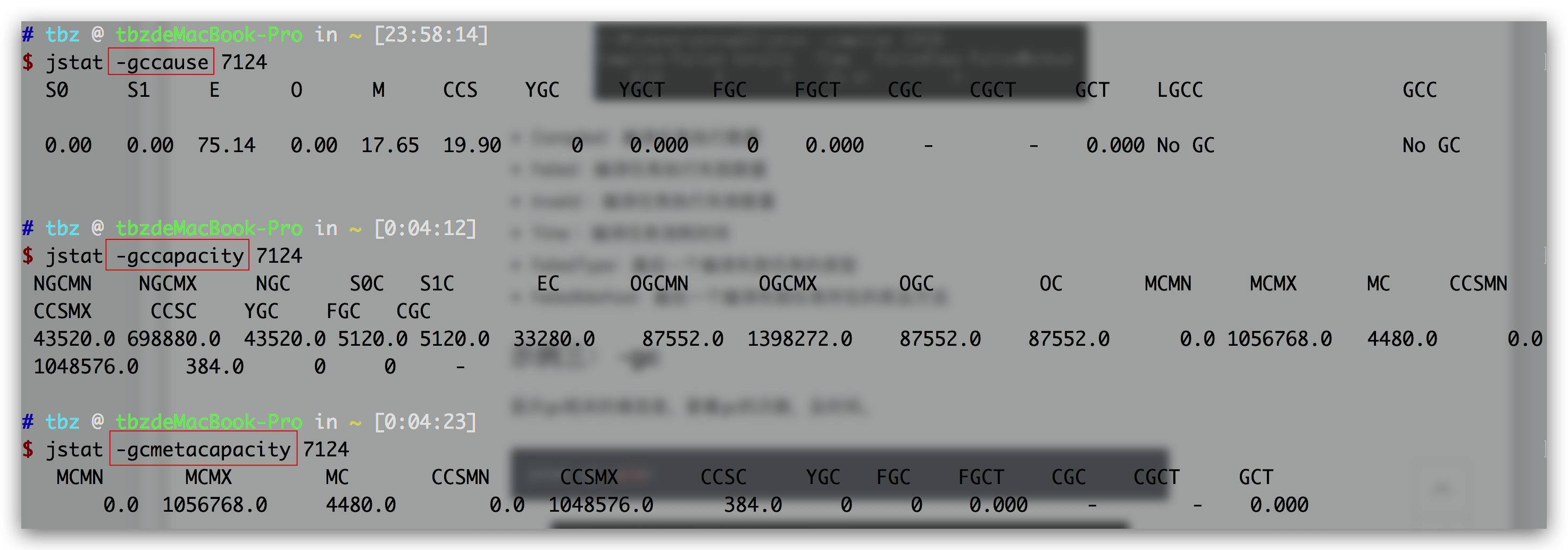
1. jstat -class {pid} {interval} {count} 监控ClassLoader情况



jstat -compiler {pid}监控JIT情况

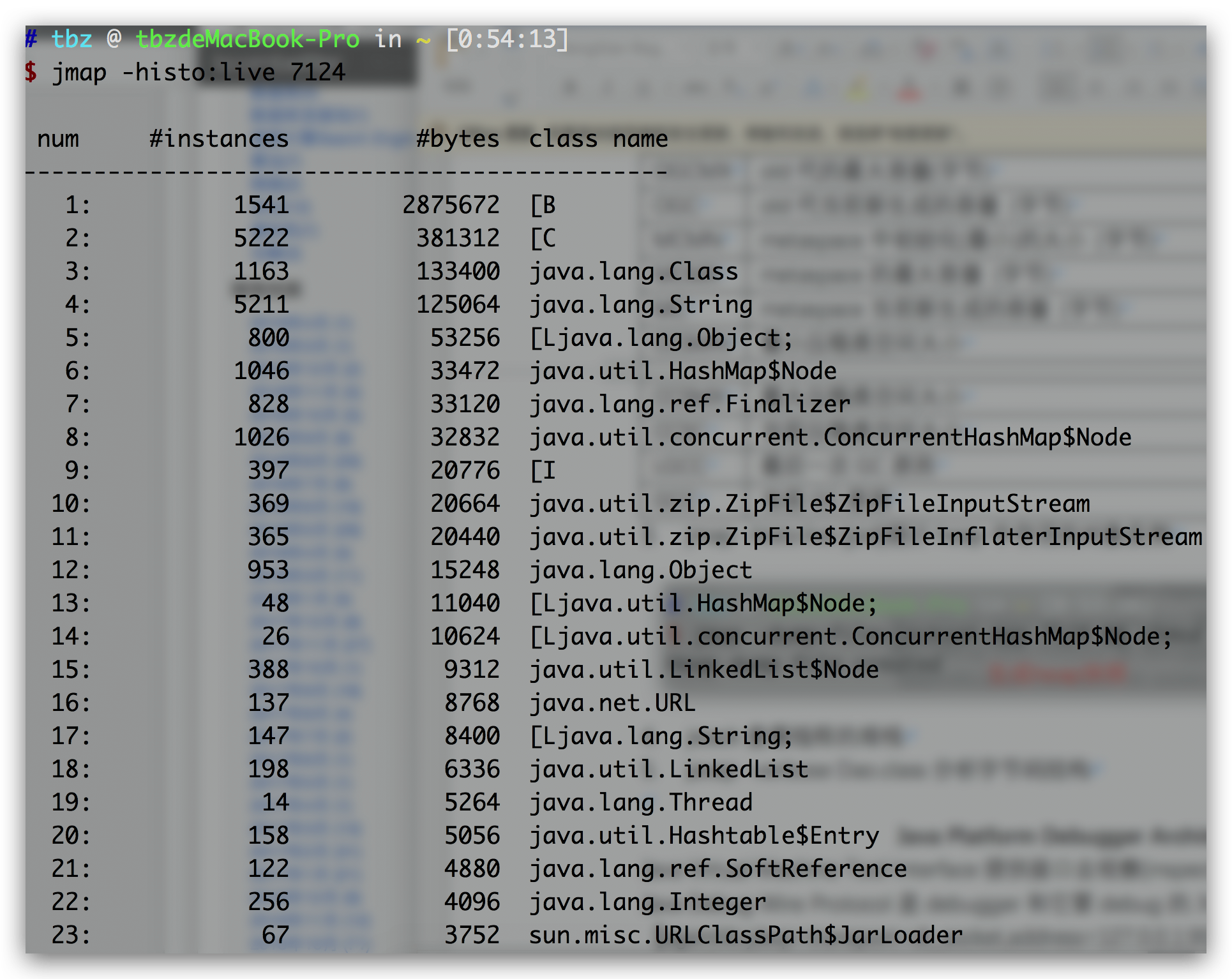


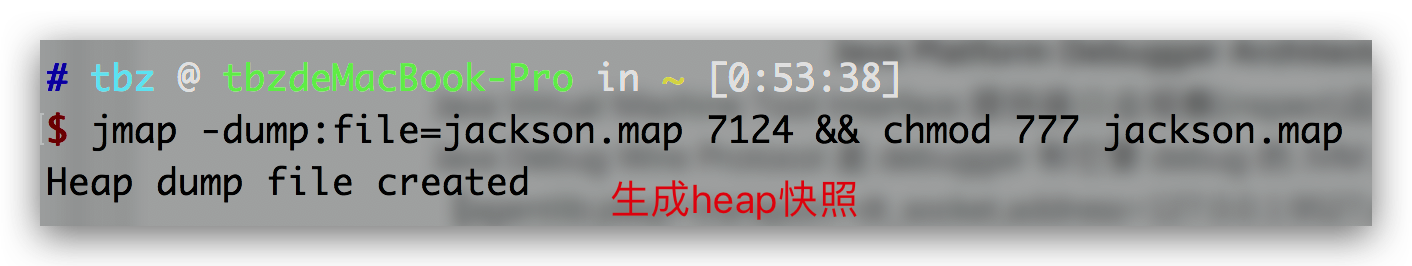
监控heap、metaspace、gc情况！



|  |  |
| --- | --- |
| S0C | 年轻代中第一个survivor（幸存区）的容量（字节） |
| S1C | 年轻代中第二个survivor（幸存区）的容量 (字节) |
| EC | 年轻代中Eden（伊甸园）的容量 (字节) |
| OC | Old代的容量 (字节) |
| MC | metaspace的容量 (字节) |
| 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 |
| 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) |
| CGC | Concurrent Garbage Collection次数 |
| CGCT | Concurrent Garbage Collection时间 |
| NGCMN | 年轻代中初始化(最小)的大小(字节) |
| NGCMX | 年轻代的最大容量 (字节) |
| NGC | 年轻代中当前的容量 (字节) |
| OGCMN | old代中初始化(最小)的大小 (字节) |
| OGCMX | old代的最大容量(字节) |
| OGC | old代当前新生成的容量 (字节) |
| MCMN | metaspace中初始化(最小)的大小 (字节) |
| MCMX | metaspace的最大容量 (字节) |
| MC | metaspace当前新生成的容量 (字节) |
| CCSMN | 最小压缩类空间大小 |
| CCSMX | 最大压缩类空间大小 |
| CCSC | 当前压缩类空间大小 |
| LGCC | 最后一次GC原因 |
| GCC | 当前GC原因 |

1. jmap -histo:live {pid} 统计heap中存活的对象实例





1. jstack查看线程的堆栈
2. javap -verbose Dao.class分析字节码结构

**Java Platform Debugger Architecture**

Java Virtual Machine Tool Interface提供接口去观察(inspect)应用状态和控制应用的执行

Java Debug Wire Protocol是debugger和它要debug的JVM之间进行通讯的协议

【agentlib:jdwp=transport=dt\_socket,address=127.0.0.1:9527,server=y,suspend=n】

Java Debug Interface高层开发接口，方便debugger编写符合JDWP格式的数据

我们每次在IDE里进行debug时，实质上是通过IDE里的debugger界面执行GUI操作，然后通过JDI发送数据到JDWP，再经过JVMTI最终实现的程序调试。

**jdb命令行调试**

开始监听java -agentlib:jdwp=transport=dt\_socket,server=y,suspend=n,address=9527

建立连接jdb -attach 9527

设置断点stop in jackson.Test.print

清除断点clear jackson.Test.print

监视变量watch jackson.Test.i

清除监视unwatch jackson.Test.i

逐行执行next

继续执行cont

查看变量dump jackson.Test.objectMapper

查看线程threads