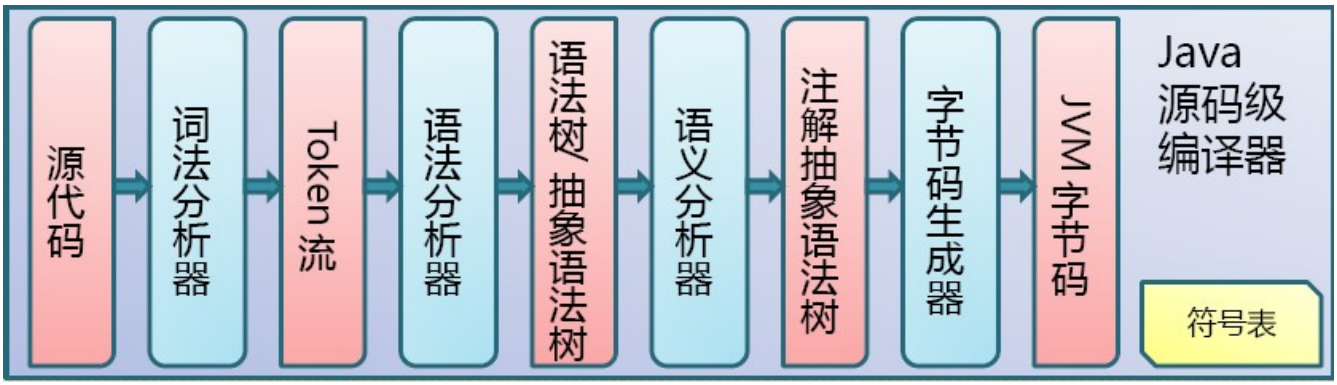
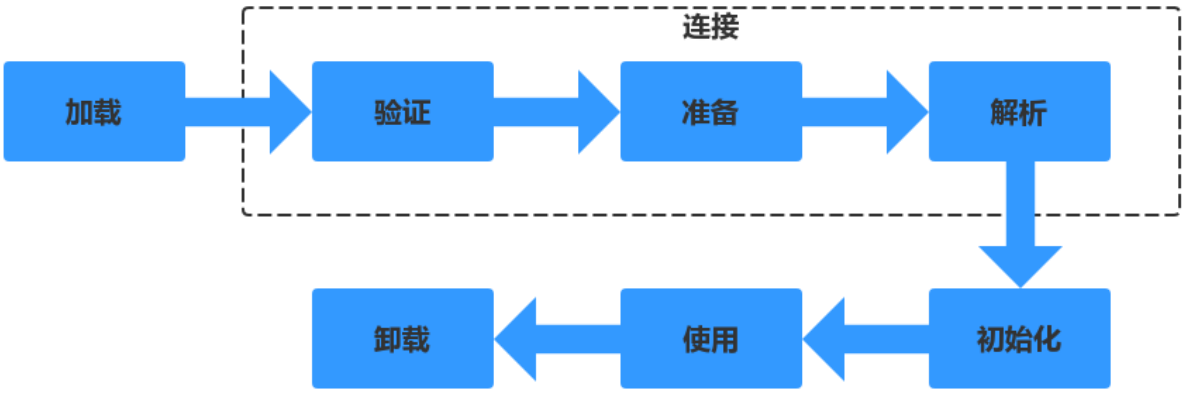
类加载机制与JDK调优监控工具

Java运行时编译源码(.java)成字节码，由jre运行。jre由java虚拟机实现。JVM分析字节码，后解释并执行



类的生命周期



1.加载

将.class文件从磁盘读到内存

2.连接

2.1 验证

验证字节码文件的正确性

2.2 准备

给类的静态变量分配内存，并赋予默认值

2.3 解析

类装载器装入类所引用的其它所有类

3.初始化

为类的静态变量赋予正确的初始值，上述的准备阶段为静态变量赋予的是虚拟机默认的初始值，此处赋予的才是程序编写者为变量分配的真正的初始值，执行静态代码块

4.使用

5.卸载

类加载器的种类

启动类加载器(Bootstrap ClassLoader)

负责加载JRE的核心类库，如JRE目标下的rt.jar，charsets.jar等

扩展类加载器(Extension ClassLoader)

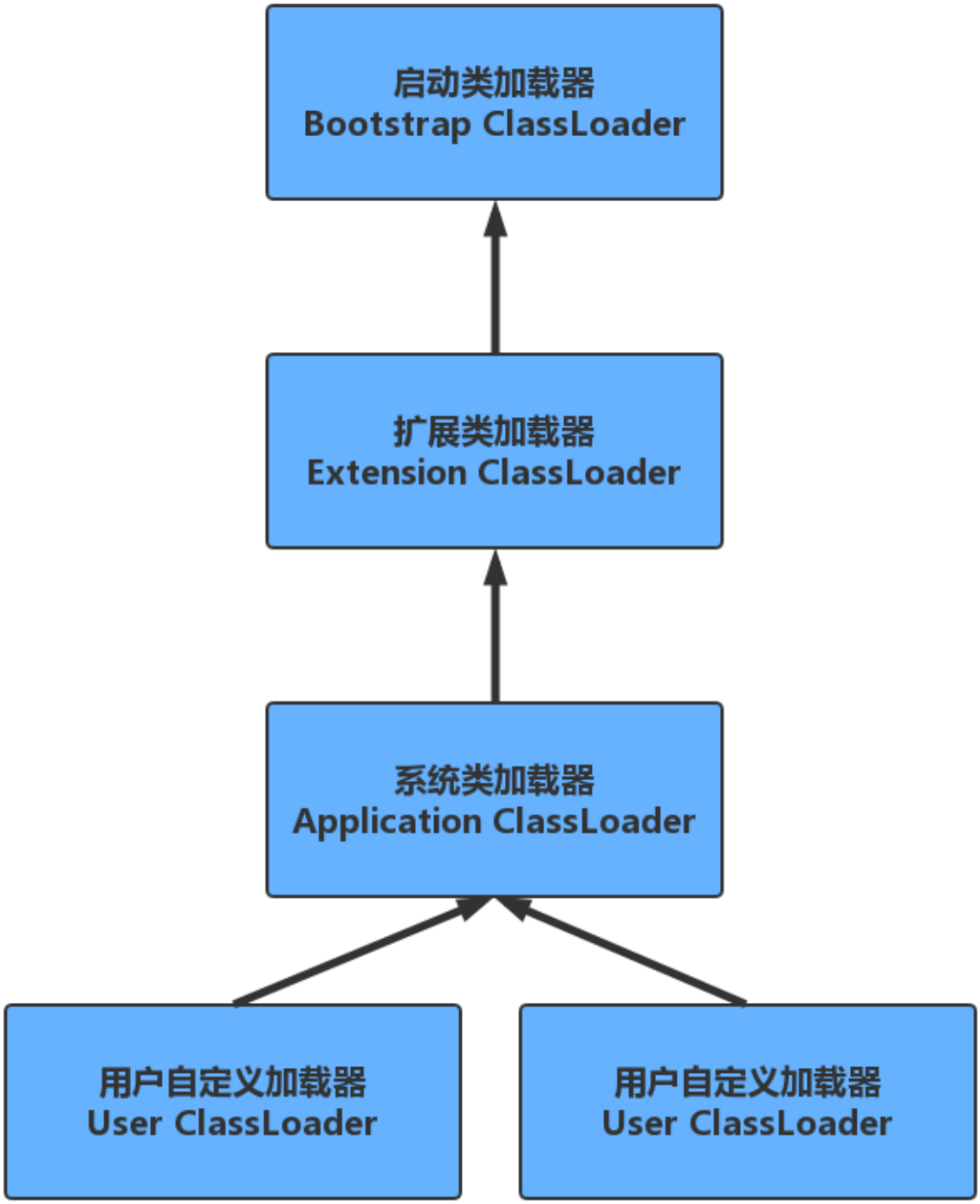
负责加载JRE扩展目录ext中jar类包

系统类加载器(Application ClassLoader)

负责加载ClassPath路径下的类包

用户自定义加载器(User ClassLoader)

负责加载用户自定义路径下的类包



类加载机制

全盘负责委托机制

当一个ClassLoader加载一个类的时候，除非显示的使用另一个ClassLoader，该类所依赖和引用的类也由这个ClassLoader载入

双亲委派机制

指先委托父类加载器寻找目标类，在找不到的情况下载自己的路径中查找并载入目标类

双亲委派模式的优势

- 沙箱安全机制：比如自己写的String.class类不会被加载，这样可以防止核心库被随意篡改
- 避免类的重复加载：当父ClassLoader已经加载了该类的时候，就不需要子CJlassLoader再加载一次

JVM性能调优监控工具

Jinfo

查看正在运行的Java程序的扩展参数

查看JVM的参数

```
D:\>jinfo -flags 7824
Attaching to process ID 7824, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.73-b02
Non-default VM flags: -XX:CICompilerCount=4 -XX:InitialHeapSize=134217728 -XX:MaxHeapSize=2124414976 -XX:MaxNewSize=707788800
-XX:MinHeapDeltaBytes=524288 -XX:NewSize=44564480 -XX:OldSize=89653248 -XX:+UseCompressedClassPointers -XX:+UseCompressedOops
-XX:+UseFastUnorderedTimeStamps -XX:-UseLargePagesIndividualAllocation -XX:+UseParallelGC
Command line:
```

查看java系统属性

等同于System.getProperties()

```
D:\>jinfo -sysprops 7824
Attaching to process ID 7824, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.73-b02
java.runtime.name = Java(TM) SE Runtime Environment
java.vm.version = 25.73-b02
sun.boot.library.path = C:\Program Files\Java\jdk1.8.0_73\jre\bin
java.protocol.handler.pkgs = org.springframework.boot.loader
java.vendor.url = http://java.oracle.com/
java.vm.vendor = Oracle Corporation
path.separator = ;
file.encoding.pkg = sun.io
java.vm.name = Java HotSpot(TM) 64-Bit Server VM
sun.os.patch.level =
sun.java.launcher = SUN_STANDARD
user.script =
user.country = CN
user.dir = D:\
java.vm.specification.name = Java Virtual Machine Specification
PID = 7824
java.runtime.version = 1.8.0_73-b02
java.awt.graphicsenv = sun.awt.Win32GraphicsEnvironment
os.arch = amd64
java.endorsed.dirs = C:\Program Files\Java\jdk1.8.0_73\jre\lib\endorsed
line.separator =

java.io.tmpdir = C:\Users\colde\AppData\Local\Temp\
java.vm.specification.vendor = Oracle Corporation
user.variant =
os.name = Windows 10
sun.jnu.encoding = GBK
java.library.path = C:\Program Files\Java\jdk1.8.0_73\bin;C:\WINDOWS\Sun\Java\bin;C:\WINDOWS\system32;C:\WINDOWS;D:\work\WorkS
oft\;C:\Program Files\Java\jdk1.8.0_73\bin;C:\Program Files\Java\jdk1.8.0_73\jre\bin;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS
\System32\Wbem;C:\WINDOWS\System32\WindowsPowerShell\v1.0\;C:\WINDOWS\System32\OpenSSH\;D:\work\WorkSoft\apache-maven-3.6.0\bi
n\;D:\Soft\pycSafefile\x64;C:\Program Files (x86)\Pandoc\;D:\work\WorkSoft\gradle-4.9\bin;C:\Users\colde\AppData\Local\Microso
ft\WindowsApps;;.
spring.beaninfo.ignore = true
```

Jstat

jstat命令可以查看堆内存各部分的使用量，以及加载类的数量。命令格式：

jstat [-命令选项] [vmid] [间隔时间/毫秒] [查询次数]

类加载统计

```
D:\>jstat -class 7824
Loaded  Bytes  Unloaded  Bytes     Time
 11623 20116.7        1     1.0       8.69
```

- Loaded：加载class的数量
- Bytes：所占用空间大小
- Unloaded：未加载数量
- Bytes：未加载占用空间
- Time：时间

垃圾回收统计

```
D:\>jstat -gc 7824
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC     MU    CCSC   CCSU   YGC     YGCT    FGC    FGCT     GCT
12288.0 12800.0 8535.7  0.0   617984.0 283891.9  93696.0    43929.5   54488.0 51921.7 7424.0 6946.6     14    0.123   2      0.225    0.348
```

- S0C：第一个Survivor区的空间
- S1C：第二个Survivor区的空间
- S0U：第一个Survivor区的使用空间
- S1U：第二个Survivor区的使用空间
- EC：Eden区的总空间
- EU：Eden区的使用空间
- OC：Old区的总空间
- OU：Old区的已使用空间
- MC：元空间的总空间
- MU：元空间的使用空间
- CCSC：压缩类的总空间
- CCSU：压缩类的使用空间
- YGC：年轻代垃圾回收次数
- YGCT：年轻代垃圾回收消耗时间
- FGC：老年代垃圾回收次数
- FGCT：老年代垃圾回收消耗时间
- GCT：垃圾回收消耗总时间

堆内存统计

```
D:\>jstat -gccapacity 7824
 NGCMN    NGCMX     NGC     S0C   S1C       EC      OGCMN      OGCMX       OGC         OC       MCMN     MCMX      MC     CCSMN    CCSMX     CCSC    YGC    FGC
 43520.0 691200.0 655360.0 12288.0 12800.0 617984.0    87552.0  1383424.0    93696.0    93696.0      0.0 1095680.0  54488.0      0.0 1048576.0   7424.0     14     2
```

- NGCMN：新生代最小空间
- NGCMX：新生代最大空间
- NGC：当前新生代空间
- S0C：第一个Survivor区空间
- S1C：第二个Survivor区空间
- EC：Eden区的总空间
- OGCMN：老年代最小空间
- OGCMX：老年代最大空间
- OGC：当前老年代空间
- OC：当前老年代空间
- MCMN：最小元空间大小
- MCMX：最大元空间大小

- MC：当前元空间大小
- 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
- YGC：年轻代GC次数
- FGC：老年代GC次数

新生代垃圾回收统计

```
D:\>jstat -gcnew 7824
 S0C    S1C    S0U    S1U   TT MTT  DSS      EC       EU     YGC     YGCT
12288.0 12800.0 8535.7    0.0   2  15 12800.0 617984.0 309847.4     14    0.123
```

- S0C：第一个Survivor区空间
- S1C：第二个Survivor区空间
- S0U：第一个Survivor区的使用空间
- S1U：第二个Survivor区的使用空间
- TT：对象在新生代存活的次数
- MTT：对象在新生代存活的最大次数
- DSS：期望Survivor区大小
- EC：Eden区的空间
- EU：Eden区的使用空间
- YGC：年轻代垃圾回收次数
- YGCT：年轻代垃圾回收消耗时间

新生代内存统计

```
D:\>jstat -gcnewcapacity 7824
  NGCMN      NGCMX       NGC      S0CMX     S0C     S1CMX     S1C       ECMX        EC      YGC   FGC
   43520.0   691200.0   655360.0 230400.0  12288.0 230400.0  12800.0   690176.0   617984.0    14     2
```

- NGCMN：新生代最小空间
- NGCMX：新生代最大空间
- NGC：当前新生代空间
- S0CMX：最大第一个Survivor区空间
- S0C：当前第一个Survivor区空间
- S1CMX：最大第二个Survivor区空间
- S1C：当前第二个Survivor区空间
- ECMX：最大Eden区空间
- EC：当前Eden区空间
- YGC：年轻代垃圾回收次数
- FGC：老年代垃圾回收次数

老年代垃圾回收统计

```
D:\>jstat -gcold 7824
   MC       MU      CCSC     CCSU       OC          OU       YGC    FGC    FGCT     GCT
 54488.0  51921.7   7424.0   6946.6     93696.0     43929.5     14     2    0.225    0.348
```

- MC：元空间的总空间
- MU：元空间的使用空间
- CCSC：压缩类的总空间

- CCSU：压缩类的使用空间
- OC：Old区的总空间
- OU：Old区的已使用空间
- YGC：年轻代GC次数
- FGC：老年代GC次数
- FGCT：老年代垃圾回收消耗时间
- GCT：垃圾回收消耗总时间

老年代内存统计

```
D:\>jstat -gcoldcapacity 7824
   OGCMN       OGCMX        OGC         OC       YGC   FGC    FGCT     GCT
    87552.0   1383424.0     93696.0     93696.0    14     2    0.225    0.348
```

- OGCMN：老年代最小空间
- OGCMX：老年代最大空间
- OGC：当前老年代空间
- OC：当前老年代空间
- YGC：年轻代GC次数
- FGC：老年代GC次数
- FGCT：老年代垃圾回收消耗时间
- GCT：垃圾回收消耗总时间

元空间内存统计

```
D:\>jstat -gcmetacapacity 7824
   MCMN       MCMX        MC       CCSMN      CCSMX      CCSC     YGC   FGC    FGCT     GCT
       0.0  1095680.0    54488.0        0.0  1048576.0     7424.0    14     2    0.225    0.348
```

- MCMN：最小元空间大小
- MCMX：最大元空间大小
- MC：当前元空间大小
- 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
- YGC：年轻代GC次数
- FGC：老年代GC次数
- FGCT：老年代垃圾回收消耗时间
- GCT：垃圾回收消耗总时间

总垃圾回收统计

```
D:\>jstat -gcutil 7824
  S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT     GCT
 69.46   0.00  51.34  46.89  95.29  93.57     14    0.123     2    0.225    0.348
```

- S0：第一个Survivor区当前使用比例
- S1：第二个Survivor区当前使用比例
- E：Eden区使用比例
- O：Old区使用比例
- M：元空间使用比例

- CCS：压缩使用比例
- YGC：年轻代垃圾回收次数
- FGC：老年代垃圾回收次数
- FGCT：老年代垃圾回收消耗时间
- GCT：垃圾回收消耗总时间

Jmap

可以用来查看内存信息

堆的对象统计

```
jmap -histo 7824 > xxx.txt
```

如图：

```
 num     #instances         #bytes  class name
----------------------------------------------
   1:         18829      143011888  [I
   2:        680830      125590192  [C
   3:       1164734       37271488  java.util.concurrent.locks.AbstractQueuedSynchronizer$Node
   4:         22790       11492832  [B
   5:        451949       10846776  java.lang.String
   6:         74841        4789824  java.net.URL
   7:         42656        3753728  java.lang.reflect.Method
   8:        115310        3689920  org.springframework.boot.loader.jar.StringSequence
   9:         57398        3214288  java.util.LinkedHashMap
  10:         46245        2323200  [Ljava.lang.Object;
  11:         30449        2083584  [Ljava.util.HashMap$Node;
  12:         48879        1955160  java.util.LinkedHashMap$Entry
  13:         59480        1903360  java.util.concurrent.ConcurrentHashMap$Node
  14:         90053        1898440  [Ljava.lang.Class;
  15:         60960        1463040  java.lang.StringBuffer
  16:         12314        1359880  java.lang.Class
  17:         56533        1356792  org.springframework.boot.loader.jar.JarURLConnection$JarEntryName
  18:         32990        1055680  java.util.HashMap$Node
  19:         31574        1043656  [Ljava.lang.String;
  20:         27354         875328  java.lang.ref.WeakReference
  21:         11991         863352  java.lang.reflect.Field
  22:         10505         839184  [S
  23:           281         630064  [Ljava.util.concurrent.ConcurrentHashMap$Node;
  24:         12766         612768  java.util.HashMap
  25:         25378         609072  java.lang.StringBuilder
  26:          7258         522576  org.springframework.core.type.classreading.AnnotationMetadataReadingVisitor
  27:         12935         517400  java.lang.ref.SoftReference
  28:         12700         506776  [Ljava.lang.reflect.Method;
  29:         30525         488400  java.lang.Object
  30:         12199         487960  java.util.HashMap$KeyIterator
  31:          9820         471360  org.springframework.core.ResolvableType
  32:         29333         469328  java.util.LinkedHashSet
  33:         17418         418032  java.util.ArrayList
  34:          5615         404280  org.springframework.core.annotation.AnnotationAttributes
  35:          7189         402584  java.beans.MethodDescriptor
```

- Num：序号
- Instances：实例数量
- Bytes：占用空间大小
- Class Name：类名

堆信息

```
D:\>jmap -histo 7824 > log.txt

D:\>jmap -heap 7824
Attaching to process ID 7824, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.73-b02

using thread-local object allocation.
Parallel GC with 8 thread(s)

Heap Configuration:
   MinHeapFreeRatio         = 0
   MaxHeapFreeRatio         = 100
   MaxHeapSize              = 2124414976 (2026.0MB)
   NewSize                  = 44564480 (42.5MB)
   MaxNewSize               = 707788800 (675.0MB)
   OldSize                  = 89653248 (85.5MB)
   NewRatio                 = 2
   SurvivorRatio            = 8
   MetaspaceSize            = 21807104 (20.796875MB)
   CompressedClassSpaceSize = 1073741824 (1024.0MB)
   MaxMetaspaceSize         = 17592186044415 MB
   G1HeapRegionSize         = 0 (0.0MB)

Heap Usage:
PS Young Generation
Eden Space:
   capacity = 632815616 (603.5MB)
   used     = 329940176 (314.6554718017578MB)
   free     = 302875440 (288.8445281982422MB)
   52.138437746770144% used
From Space:
   capacity = 12582912 (12.0MB)
   used     = 8740536 (8.335624694824219MB)
   free     = 3842376 (3.6643753051757812MB)
   69.46353912353516% used
To Space:
   capacity = 13107200 (12.5MB)
   used     = 0 (0.0MB)
   free     = 13107200 (12.5MB)
   0.0% used
PS Old Generation
   capacity = 95944704 (91.5MB)
   used     = 44983840 (42.899932861328125MB)
   free     = 50960864 (48.600067138671875MB)
   46.885172526041664% used

26884 interned Strings occupying 3314192 bytes.
```

堆内存dump

```
D:\>jmap -dump:format=b,file=eureka.hprof 7824
Dumping heap to D:\eureka.hprof ...
Heap dump file created
```

也可以在设置内存溢出的时候自动导出dump文件（内存很大的时候，可能会导不出来）

1.-XX:+HeapDumpOnOutOfMemoryError

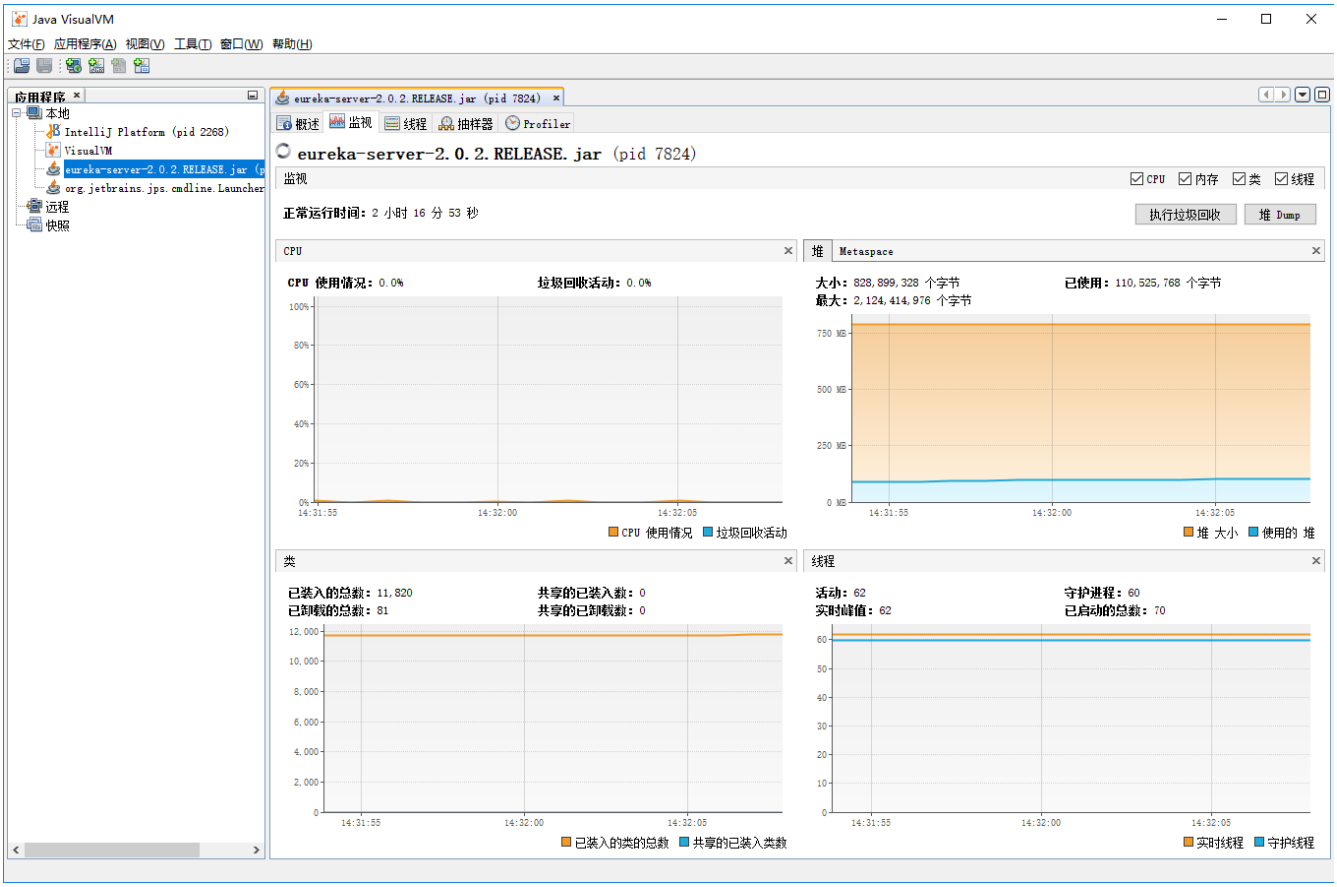
2.-XX:HeapDumpPath=输出路径

```
-Xms10m -Xmx10m -XX:+PrintGCDetails -XX:+HeapDumpOnOutOfMemoryError -
XX:HeapDumpPath=d:\oomdump.dump
```

```java
public class OutOfMemoryDump {

    /**
     *  设置JVM参数
     *  -Xms10m
     *  -Xmx10m
     *  -XX:+PrintGCDetails
     *  -XX:+HeapDumpOnOutOfMemoryError
     *  -XX:HeapDumpPath=./（路径）
     */
    public static void main(String[] args) {
        List<Object> list = new ArrayList<>();
        int i = 0;
        while(true){
            list.add(new User(i++,UUID.randomUUID().toString()));
        }
    }
```

可以使用jvisualvm命令工具导入文件分析



Jstack

jstack用于生成java虚拟机当前时刻的线程快照。

```
D:\>jstack 7824
2019-05-26 15:01:56
Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.73-b02 mixed mode):

"JMX server connection timeout 76" #76 daemon prio=5 os_prio=0 tid=0x000000001d359800 nid=0x3a7c in Object.wait() [0x000000002be6f000]
   java.lang.Thread.State: TIMED_WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        at com.sun.jmx.remote.internal.ServerCommunicatorAdmin$Timeout.run(ServerCommunicatorAdmin.java:168)
        - locked <0x00000000ff3b0178> (a [I)
        at java.lang.Thread.run(Thread.java:745)

"RMI Scheduler(0)" #75 daemon prio=5 os_prio=0 tid=0x000000001d355800 nid=0x3ba8 waiting on condition [0x000000002bd6f000]
   java.lang.Thread.State: TIMED_WAITING (parking)
        at sun.misc.Unsafe.park(Native Method)
        - parking to wait for  <0x0000000083c9ad38> (a java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject)
        at java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:215)
        at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.awaitNanos(AbstractQueuedSynchronizer.java:2078)
        at java.util.concurrent.ScheduledThreadPoolExecutor$DelayedWorkQueue.take(ScheduledThreadPoolExecutor.java:1093)
        at java.util.concurrent.ScheduledThreadPoolExecutor$DelayedWorkQueue.take(ScheduledThreadPoolExecutor.java:809)
        at java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:1067)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1127)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
        at java.lang.Thread.run(Thread.java:745)

"RMI TCP Accept-0" #73 daemon prio=5 os_prio=0 tid=0x00000000225db800 nid=0x3344 runnable [0x000000002ba6e000]
   java.lang.Thread.State: RUNNABLE
        at java.net.Dual1StackPlainSocketImpl.accept0(Native Method)
        at java.net.Dual1StackPlainSocketImpl.socketAccept(Dual1StackPlainSocketImpl.java:131)
        at java.net.AbstractPlainSocketImpl.accept(AbstractPlainSocketImpl.java:409)
        at java.net.PlainSocketImpl.accept(PlainSocketImpl.java:199)
        - locked <0x0000000083ac1830> (a java.net.SocksSocketImpl)
        at java.net.ServerSocket.implAccept(ServerSocket.java:545)
        at java.net.ServerSocket.accept(ServerSocket.java:513)
        at sun.management.jmxremote.Local1RMIServerSocketFactory$1.accept(Local1RMIServerSocketFactory.java:52)
        at sun.rmi.transport.tcp.TCPTransport$AcceptLoop.executeAcceptLoop(TCPTransport.java:400)
        at sun.rmi.transport.tcp.TCPTransport$AcceptLoop.run(TCPTransport.java:372)
        at java.lang.Thread.run(Thread.java:745)

"DestroyJavaVM" #72 prio=5 os_prio=0 tid=0x00000000225db000 nid=0x3a64 waiting on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"http-nio-8080-AsyncTimeout" #70 daemon prio=5 os_prio=0 tid=0x00000000225da000 nid=0x36bc waiting on condition [0x000000002af6f000]
   java.lang.Thread.State: TIMED_WAITING (sleeping)
        at java.lang.Thread.sleep(Native Method)
        at org.apache.coyote.AbstractProtocol$AsyncTimeout.run(AbstractProtocol.java:1133)
        at java.lang.Thread.run(Thread.java:745)
```