二

09 JDK 内置命令行工具：工欲善其事，必先利其器

很多情况下，JVM 运行环境中并没有趁手的工具，所以掌握基本的内置工具是一项基本功。

JDK 自带的工具和程序可以分为 2 大类型：

1. 开发工具
2. 诊断分析工具

JDK 内置的开发工具

写过 Java 程序的同学，对 JDK 中的开发工具应该比较熟悉。 下面列举常用的部分：

工具	简介
java	Java 应用的启动程序
javac	JDK 内置的编译工具
javap	反编译 class 文件的工具
javadoc	根据 Java 代码和标准注释，自动生成相关的 API 说明文档
javah	JNI 开发时，根据 Java 代码生成需要的 .h 文件。
extcheck	检查某个 jar 文件和运行时扩展 jar 有没有版本冲突，很少使用
jdb	Java Debugger 可以调试本地和远端程序，属于 JPDA 中的一个 Demo 实现，供其他调试器参考。开发时很少使用

工具	简介
jdeps	探测 class 或 jar 包需要的依赖
jar	打包工具，可以将文件和目录打包成为 .jar 文件；.jar 文件本质上就是 zip 文件，只是后缀不同。使用时按顺序对应好选项和参数即可。
keytool	安全证书和密钥的管理工具（支持生成、导入、导出等操作）
jarsigner	jar 文件签名和验证工具
policytool	实际上这是一款图形界面工具，管理本机的 Java 安全策略

开发工具此处不做详细介绍，有兴趣的同学请参考文末的链接。

下面介绍诊断和分析工具。

命令行诊断和分析工具

JDK 内置了各种命令行工具，条件受限时我们可以先用命令行工具快速查看 JVM 实例的基本情况。

> macOS X、Windows 系统的某些账户权限不够，有些工具可能会报错/失败，假如出问题了请排除这个因素。

JPS 工具简介

我们知道，操作系统提供一个工具叫做 ps，用于显示进程状态（Process Status）。

Java也 提供了类似的命令行工具，叫做 JPS，用于展示 Java 进程信息（列表）。

需要注意的是，JPS 展示的是当前用户可看见的 Java 进程，如果看不见某些进程可能需要 sudo、su 之类的命令来切换权限。

查看帮助信息：

```
$ jps -help
```

```
usage: jps [-help]
       jps [-q] [-mlvV] [<hostid>]
Definitions:
    <hostid>:      <hostname>[:<port>]
```

可以看到， 这些参数分为了多个组， `-help` 、 `-q` 、 `-mlvV` ， 同一组可以共用一个 `-` 。

常用参数是小写的 `-v` ，显示传递给 JVM 的启动参数。

> $ jps -v

```
15883 Jps -Dapplication.home=/usr/local/jdk1.8.0_74 -Xms8m
6446 Jstatd -Dapplication.home=/usr/local/jdk1.8.0_74 -Xms8m
        -Djava.security.policy=/etc/java/jstatd.all.policy
32383 Bootstrap -Xmx4096m -XX:+UseG1GC -verbose:gc
        -XX:+PrintGCDateStamps -XX:+PrintGCDetails
        -Xloggc:/xxx-tomcat/logs/gc.log
        -Dcatalina.base=/xxx-tomcat -Dcatalina.home=/data/tomcat
```

看看输出的内容，其中最重要的信息是前面的进程 ID（PID）。

其他参数不太常用：

- `-q`： 只显示进程号。
- `-m`： 显示传给 main 方法的参数信息
- `-l`： 显示启动 class 的完整类名，或者启动 jar 的完整路径
- `-V`： 大写的 V，这个参数有问题，相当于没传一样。官方说的跟 `-q` 差不多。
- `<hostid>`： 部分是远程主机的标识符，需要远程主机启动 `jstatd` 服务器支持。

可以看到，格式为 `<hostname>[:<port>]` ，不能用 IP，示例： `jps -v sample.com:1099` 。

知道 JVM 进程的 PID 之后，就可以使用其他工具来进行诊断了。

jstat 工具简介

jstat 用来监控 JVM 内置的各种统计信息，主要是内存和 GC 相关的信息。

查看 jstat 的帮助信息，大致如下：

```
$ jstat -help
```

```
Usage: jstat -help|-options
       jstat -<option> [-t] [-h<lines>] <vmid> [<interval> [<count>]]

Definitions:
  <option>      可用的选项，查看详情请使用 -options
  <vmid>        虚拟机标识符，格式：<lvmid>[@<hostname>[:<port>]]
  <lines>       标题行间隔的频率.
  <interval>    采样周期，<n>["ms"|"s"]，默认单位是毫秒 "ms"
  <count>       采用总次数
  -J<flag>      传给jstat底层JVM的 <flag> 参数
```

再来看看 `<option>` 部分支持哪些选项：

```
$ jstat -options
```

```
-class
-compiler
-gc
-gccapacity
-gccause
-gcmetacapacity
-gcnew
-gcnewcapacity
-gcold
-gcoldcapacity
-gcutil
-printcompilation
```

简单说明这些选项，不感兴趣可以跳着读。

- `-class`：类加载（Class loader）信息统计。
- `-compiler`：JIT 即时编译器相关的统计信息。
- `-gc`：GC 相关的堆内存信息，用法：`jstat -gc -h 10 -t 864 1s 20`。
- `-gccapacity`：各个内存池分代空间的容量。
- `-gccause`：看上次 GC、本次 GC（如果正在 GC 中）的原因，其他输出和 `-gcutil` 选项一致。
- `-gcnew`：年轻代的统计信息（New = Young = Eden + S0 + S1）。
- `-gcnewcapacity`：年轻代空间大小统计。

- `-gcold`：老年代和元数据区的行为统计。
- `-gcoldcapacity`：old 空间大小统计。
- `-gcmetacapacity`：meta 区大小统计。
- `-gcutil`：GC 相关区域的使用率（utilization）统计。
- `-printcompilation`：打印 JVM 编译统计信息。

实例：

```
jstat -gcutil -t 864
```

`-gcutil` 选项是统计 GC 相关区域的使用率（utilization），结果如下：

Timestamp	S0	S1	E	O	M	CCS	YGC	YGCT
14251645.5	0.00	13.50	55.05	71.91	83.84	69.52	113767	206.036

`-t` 选项的位置是固定的，不能在前也不能在后。可以看出是用于显示时间戳，即 JVM 启动到现在的秒数。

简单分析一下：

- Timestamp 列：JVM 启动了 1425 万秒，大约 164 天。
- S0：就是 0 号存活区的百分比使用率。0% 很正常，因为 S0 和 S1 随时有一个是空的。
- S1：就是 1 号存活区的百分比使用率。
- E：就是 Eden 区，新生代的百分比使用率。
- O：就是 Old 区，老年代。百分比使用率。
- M：就是 Meta 区，元数据区百分比使用率。
- CCS：压缩 class 空间（Compressed class space）的百分比使用率。
- YGC（Young GC）：年轻代 GC 的次数。11 万多次，不算少。
- YGCT 年轻代 GC 消耗的总时间。206 秒，占总运行时间的万分之一不到，基本上可忽略。
- FGC：FullGC 的次数，可以看到只发生了 4 次，问题应该不大。

- FGCT：FullGC 的总时间，0.122 秒，平均每次 30ms 左右，大部分系统应该能承受。
- GCT：所有 GC 加起来消耗的总时间，即 YGCT + FGCT。

可以看到，`-gcutil` 这个选项出来的信息不太好用，统计的结果是百分比，不太直观。

再看看 `-gc` 选项，GC 相关的堆内存信息。

```
jstat -gc -t 864 1s
jstat -gc -t 864 1s 3
jstat -gc -t -h 10 864 1s 15
```

其中的 `1s` 占了 `<interval>` 这个槽位，表示每 1 秒输出一次信息。

`1s 3` 的意思是每秒输出 1 次，最多 3 次。

如果只指定刷新周期，不指定 `<count>` 部分，则会一直持续输出。 退出输出按 `CTRL+C` 即可。

`-h 10` 的意思是每 10 行输出一次表头。

结果大致如下：

Timestamp	S0C	S1C	S0U	S1U	EC	EU	OC	O
14254245.3	1152.0	1152.0	145.6	0.0	9600.0	2312.8	11848.0	85
14254246.3	1152.0	1152.0	145.6	0.0	9600.0	2313.1	11848.0	85
14254247.3	1152.0	1152.0	145.6	0.0	9600.0	2313.4	11848.0	85

上面的结果是精简过的，为了排版去掉了 GCT、CCSC、CCSU 这三列。看到这些单词可以试着猜一下意思，详细的解读如下：

- Timestamp 列：JVM 启动了 1425 万秒，大约 164 天。
- S0C：0 号存活区的当前容量（capacity），单位 kB。
- S1C：1 号存活区的当前容量，单位 kB。
- S0U：0 号存活区的使用量（utilization），单位 kB。

- S1U：1 号存活区的使用量，单位 kB。
- EC：Eden 区，新生代的当前容量，单位 kB。
- EU：Eden 区，新生代的使用量，单位 kB。
- OC：Old 区，老年代的当前容量，单位 kB。
- OU：Old 区，老年代的使用量，单位 kB。 （需要关注）
- MC：元数据区的容量，单位 kB。
- MU：元数据区的使用量，单位 kB。
- CCSC：压缩的 class 空间容量，单位 kB。
- CCSU：压缩的 class 空间使用量，单位 kB。
- YGC：年轻代 GC 的次数。
- YGCT：年轻代 GC 消耗的总时间。 （重点关注）
- FGC：Full GC 的次数
- FGCT：Full GC 消耗的时间。 （重点关注）
- GCT：垃圾收集消耗的总时间。

最重要的信息是 GC 的次数和总消耗时间，其次是老年代的使用量。

在没有其他监控工具的情况下， jstat 可以简单查看各个内存池和 GC 的信息，可用于判别是否是 GC 问题或者内存溢出。

jmap 工具

面试最常问的就是 jmap 工具了。jmap 主要用来 Dump 堆内存。当然也支持输出统计信息。

官方推荐使用 JDK 8 自带的 jcmd 工具来取代 jmap，但是 jmap 深入人心，jcmd 可能暂时取代不了。

查看 jmap 帮助信息：

```
$ jmap -help
```

```
Usage:
    jmap [option] <pid>
        (连接到本地进程)
    jmap [option] <executable <core>
```

```
        (连接到 core file)
    jmap [option] [server_id@]<remote-IP-hostname>
        (连接到远程 debug 服务)

where <option> is one of:
    <none>                  等同于 Solaris 的 pmap 命令
    -heap                   打印 Java 堆内存汇总信息
    -histo[:live]           打印 Java 堆内存对象的直方图统计信息
                            如果指定了 "live" 选项则只统计存活对象，强制触发一次 GC
    -clstats                打印 class loader 统计信息
    -finalizerinfo          打印等待 finalization 的对象信息
    -dump:<dump-options>    将堆内存 dump 为 hprof 二进制格式
                            支持的 dump-options:
                              live         只 dump 存活对象，不指定则导出全部。
                              format=b     二进制格式(binary format)
                              file=<file>  导出文件的路径
                            示例：jmap -dump:live,format=b,file=heap.bin <pid>
    -F                      强制导出，若 jmap 被 hang 住不响应，可断开后使用此选项。
                            其中 "live" 选项不支持强制导出。
    -h | -help              to print this help message
    -J<flag>                to pass <flag> directly to the runtime system
```

常用选项就 3 个：

- `-heap`：打印堆内存（/内存池）的配置和使用信息。
- `-histo`：看哪些类占用的空间最多，直方图。
- `-dump:format=b,file=xxxx.hprof`：Dump 堆内存。

示例：看堆内存统计信息。

```
$ jmap -heap 4524
```

输出信息：

```
Attaching to process ID 4524, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.65-b01

using thread-local object allocation.
Parallel GC with 4 thread(s)

Heap Configuration:
   MinHeapFreeRatio         = 0
   MaxHeapFreeRatio         = 100
   MaxHeapSize              = 2069889024 (1974.0MB)
   NewSize                  = 42991616 (41.0MB)
   MaxNewSize               = 689963008 (658.0MB)
```

```
   OldSize                  = 87031808 (83.0MB)
   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 = 24117248 (23.0MB)
   used     = 11005760 (10.49591064453125MB)
   free     = 13111488 (12.50408935546875MB)
   45.63439410665761% used
From Space:
   capacity = 1048576 (1.0MB)
   used     = 65536 (0.0625MB)
   free     = 983040 (0.9375MB)
   6.25% used
To Space:
   capacity = 1048576 (1.0MB)
   used     = 0 (0.0MB)
   free     = 1048576 (1.0MB)
   0.0% used
PS Old Generation
   capacity = 87031808 (83.0MB)
   used     = 22912000 (21.8505859375MB)
   free     = 64119808 (61.1494140625MB)
   26.32600715361446% used

12800 interned Strings occupying 1800664 bytes.
```

- Attached，连着；
- Detached，分离。

可以看到堆内存和内存池的相关信息。当然，这些信息有多种方式可以得到，比如 JMX。

看看直方图：

```
$ jmap -histo 4524
```

结果为：

```
 num     #instances         #bytes  class name
----------------------------------------------
   1:         52214       11236072  [C
   2:        126872        5074880  java.util.TreeMap$Entry
   3:          5102        5041568  [B
```

```
   4:         17354        2310576  [I
   5:         45258        1086192  java.lang.String
......
```

简单分析，其中 `[C` 占用了 11MB 内存，没占用什么空间。

`[C` 表示 `chat[]`，`[B` 表示 `byte[]`，`[I` 表示 `int[]`，其他类似。这种基础数据类型很难分析出什么问题。

Java 中的大对象、巨无霸对象，一般都是长度很大的数组。

Dump 堆内存：

```
cd $CATALINA_BASE
jmap -dump:format=b,file=3826.hprof 3826
```

导出完成后，dump 文件大约和堆内存一样大。可以想办法压缩并传输。

分析 hprof 文件可以使用 jhat 或者 mat 工具。

jcmd 工具

诊断工具：jcmd 是 JDK 8 推出的一款本地诊断工具，只支持连接本机上同一个用户空间下的 JVM 进程。

查看帮助：

> `$ jcmd -help`

```
Usage: jcmd <pid | main class> <command ...|PerfCounter.print|-f file>
   or: jcmd -l
   or: jcmd -h

  command 必须是指定 JVM 可用的有效 jcmd 命令。
  可以使用 "help" 命令查看该 JVM 支持哪些命令。
  如果指定 pid 部分的值为 0，则会将 commands 发送给所有可见的 Java 进程。
  指定 main class 则用来匹配启动类。可以部分匹配。（适用同一个类启动多实例）。
  If no options are given, lists Java processes (same as -p).

  PerfCounter.print 命令可以展示该进程暴露的各种计数器
  -f  从文件读取可执行命令
  -l  列出（list）本机上可见的 JVM 进程
  -h  this help
```

查看进程信息：

```
jcmd
jcmd -l
jps -lm

11155 org.jetbrains.idea.maven.server.RemoteMavenServer
```

这几个命令的结果差不多。可以看到其中有一个 PID 为 11155 的进程，下面看看可以用这个 PID 做什么。

给这个进程发一个 help 指令：

```
jcmd 11155 help
jcmd RemoteMavenServer help
```

pid 和 main-class 输出信息是一样的：

```
11155:
The following commands are available:
VM.native_memory
ManagementAgent.stop
ManagementAgent.start_local
ManagementAgent.start
GC.rotate_log
Thread.print
GC.class_stats
GC.class_histogram
GC.heap_dump
GC.run_finalization
GC.run
VM.uptime
VM.flags
VM.system_properties
VM.command_line
VM.version
help
```

可以试试这些命令。查看 VM 相关的信息：

```
# JVM 实例运行时间
jcmd 11155 VM.uptime
```

```
9307.052 s

#JVM 版本号
jcmd 11155 VM.version
OpenJDK 64-Bit Server VM version 25.76-b162
JDK 8.0_76

# JVM 实际生效的配置参数
jcmd 11155 VM.flags
11155:
-XX:CICompilerCount=4 -XX:InitialHeapSize=268435456
-XX:MaxHeapSize=536870912 -XX:MaxNewSize=178782208
-XX:MinHeapDeltaBytes=524288 -XX:NewSize=89128960
-XX:OldSize=179306496 -XX:+UseCompressedClassPointers
-XX:+UseCompressedOops -XX:+UseParallelGC

# 查看命令行参数
jcmd 11155 VM.command_line
VM Arguments:
jvm_args: -Xmx512m -Dfile.encoding=UTF-8
java_command: org.jetbrains.idea.maven.server.RemoteMavenServer
java_class_path (initial): ...(xxx省略)...
Launcher Type: SUN_STANDARD

# 系统属性
jcmd 11155 VM.system_properties
...
java.runtime.name=OpenJDK Runtime Environment
java.vm.version=25.76-b162
java.vm.vendor=Oracle Corporation
user.country=CN
```

GC 相关的命令，统计每个类的实例占用字节数。

```
$ jcmd 11155 GC.class_histogram
```

```
 num     #instances         #bytes  class name
----------------------------------------------
   1:         11613        1420944  [C
   2:          3224         356840  java.lang.Class
   3:           797         300360  [B
   4:         11555         277320  java.lang.String
   5:          1551         193872  [I
   6:          2252         149424  [Ljava.lang.Object;
```

Dump 堆内存：

```
$ jcmd 11155 help GC.heap_dump
```

```
Syntax : GC.heap_dump [options] <filename>
Arguments: filename :  Name of the dump file (STRING, no default value)
Options:  -all=true 或者 -all=false (默认)

# 两者效果差不多; jcmd 需要指定绝对路径； jmap 不能指定绝对路径
jcmd 11155 GC.heap_dump -all=true ~/11155-by-jcmd.hprof
jmap -dump:file=./11155-by-jmap.hprof 11155
```

jcmd 坑的地方在于，必须指定绝对路径，否则导出的 hprof 文件就以 JVM 所在的目录计算。（因为是发命令交给 JVM 执行的）

其他命令用法类似，必要时请参考官方文档。

jstack 工具

命令行工具、诊断工具：jstack 工具可以打印出 Java 线程的调用栈信息（Stack Trace）。一般用来查看存在哪些线程，诊断是否存在死锁等。

这时候就看出来给线程（池）命名的必要性了（开发不规范，整个项目都是坑），具体可参考阿里巴巴的 Java 开发规范。

看看帮助信息：

> $ jstack -help

```
Usage:
    jstack [-l] <pid>
        (to connect to running process)
    jstack -F [-m] [-l] <pid>
        (to connect to a hung process)
    jstack [-m] [-l] <executable> <core>
        (to connect to a core file)
    jstack [-m] [-l] [server_id@]<remote server IP or hostname>
        (to connect to a remote debug server)

Options:
    -F  to force a thread dump. Use when jstack <pid> does not respond (process is
    -m  to print both java and native frames (mixed mode)
    -l  long listing. Prints additional information about locks
    -h or -help to print this help message
```

选项说明：

- `-F`：强制执行 Thread Dump，可在 Java 进程卡死（hung 住）时使用，此选项可能需要系统权限。
- `-m`：混合模式（mixed mode），将 Java 帧和 native 帧一起输出，此选项可能需要系统权限。
- `-l`：长列表模式，将线程相关的 locks 信息一起输出，比如持有的锁，等待的锁。

常用的选项是 `-l`，示例用法。

```
jstack 4524
jstack -l 4524
```

死锁的原因一般是锁定多个资源的顺序出了问题（交叉依赖），网上示例代码很多，比如搜索“Java 死锁 示例”。

在 Linux 和 macOS 上，`jstack pid` 的效果跟 `kill -3 pid` 相同。

jinfo 工具

诊断工具：jinfo 用来查看具体生效的配置信息以及系统属性，还支持动态增加一部分参数。

看看帮助信息：

```
$ jinfo -help
```

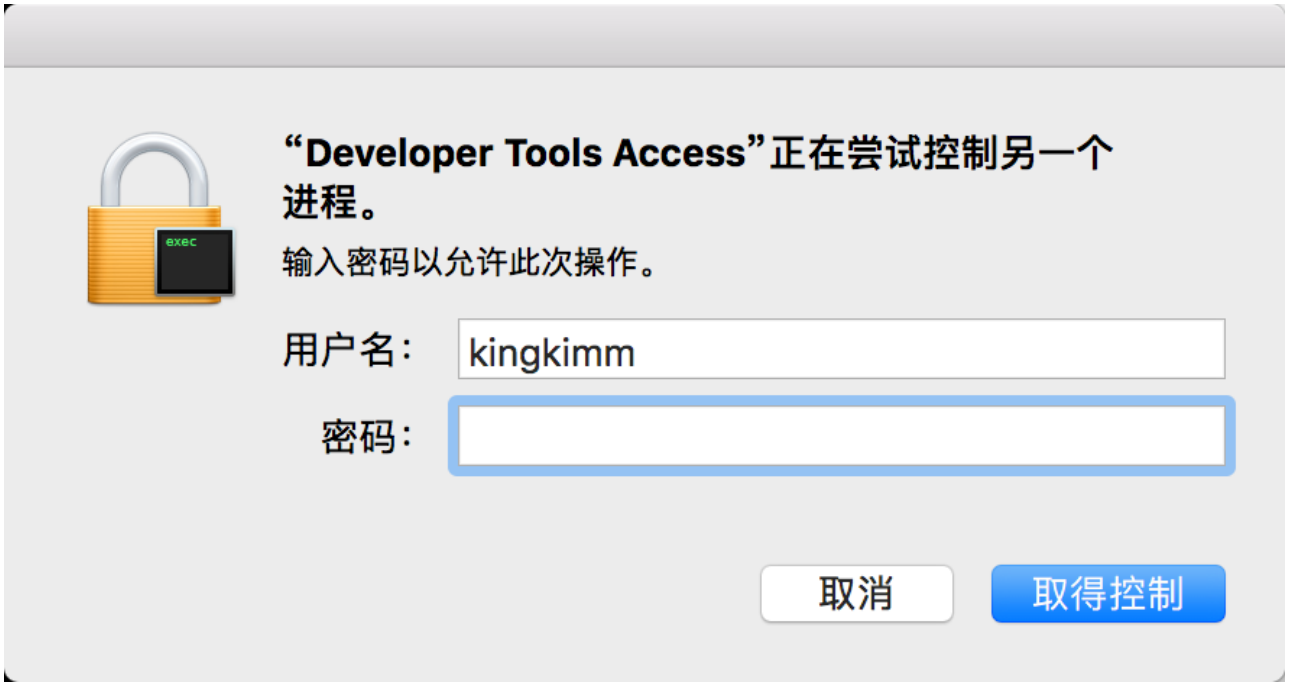
```
Usage:
    jinfo [option] <pid>
        (to connect to running process)
    jinfo [option] <executable <core>
        (to connect to a core file)
    jinfo [option] [server_id@]<remote-IP-hostname>
        (to connect to remote debug server)

where <option> is one of:
    -flag <name>         to print the value of the named VM flag
    -flag [+|-]<name>    to enable or disable the named VM flag
    -flag <name>=<value> to set the named VM flag to the given value
    -flags               to print VM flags
    -sysprops            to print Java system properties
    <no option>          to print both of the above
    -h | -help           to print this help message
```

使用示例：

```
jinfo 36663
jinfo -flags 36663
```

不加参数过滤，则打印所有信息。

jinfo 在 Windows 上比较稳定。在 macOS 上需要 root 权限，或是需要在提示下输入当前用户的密码。



然后就可以看到如下信息：

```
jinfo 36663
Attaching to process ID 36663, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.131-b11
Java System Properties:

java.runtime.name = Java(TM) SE Runtime Environment
java.vm.version = 25.131-b11
sun.boot.library.path = /Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/Contents
// 中间省略了几十行
java.ext.dirs = /Users/kimmking/Library/Java/Extensions:/Library/Java/JavaVirtualMa
sun.boot.class.path = /Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/Contents/H
java.vendor = Oracle Corporation
maven.home = /Users/kimmking/tools/apache-maven-3.5.0
file.separator = /
java.vendor.url.bug = http://bugreport.sun.com/bugreport/
```

```
sun.io.unicode.encoding = UnicodeBig
sun.cpu.endian = little
sun.cpu.isalist =

VM Flags:
Non-default VM flags: -XX:CICompilerCount=3 -XX:InitialHeapSize=134217728 -XX:MaxHe
Command line: -Dclassworlds.conf=/Users/kimmking/tools/apache-maven-3.5.0/bin/m2.co
```

可以看到所有的系统属性和启动使用的 VM 参数、命令行参数。非常有利于我们排查问题，特别是去排查一个已经运行的 JVM 里问题，通过 jinfo 我们就知道它依赖了哪些库，用了哪些参数启动。

如果在 Mac 和 Linux 系统上使用一直报错，则可能是没有权限，或者 jinfo 版本和目标 JVM 版本不一致的原因，例如：

```
Error attaching to process:
  sun.jvm.hotspot.runtime.VMVersionMismatchException:
    Supported versions are 25.74-b02. Target VM is 25.66-b17
```

jrunscript 和 jjs 工具

jrunscript 和 jjs 工具用来执行脚本，只要安装了 JDK 8+，就可以像 shell 命令一样执行相关的操作了。这两个工具背后，都是 JDK 8 自带的 JavaScript 引擎 Nashorn。

执行交互式操作：

```
$ jrunscript
nashorn> 66+88
154
```

或者：

```
$ jjs
jjs> 66+88
154
```

按 CTRL+C 或者输入 exit() 回车，退出交互式命令行。

其中 jrunscript 可以直接用来执行 JS 代码块或 JS 文件。比如类似 curl 这样的操作：

```
jrunscript -e "cat('http://www.baidu.com')"
```

或者这样：

```
jrunscript -e "print('hello,kk.jvm'+1)"
```

甚至可以执行 JS 脚本：

```
jrunscript -l js -f /XXX/XXX/test.js
```

而 jjs 则只能交互模式，但是可以指定 JavaScript 支持的 ECMAScript 语言版本，比如 ES5 或者 ES6。

这个工具在某些情况下还是有用的，还可以在脚本中执行 Java 代码，或者调用用户自己的 jar 文件或者 Java 类。详细的操作说明可以参考：

> jrunscript - command line script shell

如果是 JDK 9 及以上的版本，则有一个更完善的 REPL 工具——JShell，可以直接解释执行 Java 代码。

而这些性能诊断工具官方并不提供技术支持，所以如果碰到报错信息，请不要着急，可以试试其他工具。不行就换 JDK 版本。

参考文档

- JDK 内置程序和工具

上一页 下一页