

常用命令

3.2.1 jps

查看java进程

The jps command lists the instrumented Java HotSpot VMs on the target system. The command is limited to reporting information on JVMs for which it has the access permissions.

```
[root@localhost bin]# jps
3162 Jps
2908 Bootstrap
[root@localhost bin]# jps -l
3172 sun.tools.jps.Jps
2908 org.apache.catalina.startup.Bootstrap
[root@localhost bin]# jinfo -flag MaxHeapSize 2908
-XX:MaxHeapSize=257949696
```

3.2.2 jinfo

(1) 实时查看和调整JVM配置参数

The jinfo command prints Java configuration information for a specified Java process or core file or a remote debug server. The configuration information includes Java system properties and Java Virtual Machine (JVM) command-line flags.

(2) 查看用法

jinfo -flag name PID 查看某个java进程的name属性的值

```
jinfo -flag MaxHeapSize PID
jinfo -flag UseG1GC PID
```

```
[root@localhost bin]# jinfo -flag MaxHeapSize 2597
-XX:MaxHeapSize=257949696
[root@localhost bin]# jinfo -flag UseG1GC 2597
-XX:-UseG1GC
```

(3) 修改

参数只有被标记为manageable的flags可以被实时修改

```
jinfo -flag [+|-] PID
jinfo -flag <name>=<value> PID
```

(4) 查看曾经赋过值的一些参数

```
jinfo -flags PID
```

```
[root@localhost bin]# jinfo -flags 2908
Attaching to process ID 2908, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.191-b12
Non-default VM flags: -XX:CICompilerCount=2 -XX:InitialHeapSize=16777216 -XX:+ManagementServer -XX:MaxHeapSize=257949696
-XX:MaxNewSize=85983232 -XX:MinHeapDeltaBytes=196608 -XX:NewSize=5570560 -XX:OldSize=11206656 -XX:+UseCompressedClassPo
inters -XX:+UseCompressedOops -XX:+UseFastUnorderedTimeStamps
Command line: -Djava.util.logging.config.file=/usr/local/tomcat/apache-tomcat-8.5.37/conf/logging.properties -Djava.uti
l.logging.manager=org.apache.juli.ClassLoaderLogManager -Djdk.tls.ephemeralDHKeySize=2048 -Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.port=8999 -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremo
te.ssl=false -Djava.net.preferIPv4Stack=true -Djava.rmi.server.hostname=192.168.126.128 -Djava.protocol.handler.pkgs=org
.apache.catalina.webresources -Dorg.apache.catalina.security.SecurityListener.UMASK=0027 -Dignore.endorsed.dirs= -Dcatal
ina.base=/usr/local/tomcat/apache-tomcat-8.5.37 -Dcatalina.home=/usr/local/tomcat/apache-tomcat-8.5.37 -Djava.io.tmpdir=
/usr/local/tomcat/apache-tomcat-8.5.37/temp
```

3.2.3 jstat

(1) 查看虚拟机性能统计信息

The jstat command displays performance statistics for an instrumented Java Hotspot VM. The target JVM is identified by its virtual machine identifier, or vmid option.

(2) 查看类装载信息

jstat -class PID 1000 10 查看某个java进程的类装载信息，每1000毫秒输出一次，共输出10次

```
[root@localhost bin]# jstat -class 2597 1000 10
Loaded Bytes Unloaded Bytes Time
3042 5974.3 0 0.0 3.21
3042 5974.3 0 0.0 3.21
3042 5974.3 0 0.0 3.21
3042 5974.3 0 0.0 3.21
3042 5974.3 0 0.0 3.21
3042 5974.3 0 0.0 3.21
3042 5974.3 0 0.0 3.21
3042 5974.3 0 0.0 3.21
3042 5974.3 0 0.0 3.21
3042 5974.3 0 0.0 3.21
```

(3) 查看垃圾收集信息

jstat -gc PID 1000 10

```
[root@localhost bin]# jstat -gc 2597 1000 5
S0C S1C S0U S1U EC EU OC OU MC MU CCSC CCSU YGC YGCT FGC FGCT GCT
896.0 896.0 84.9 0.0 7488.0 794.3 18508.0 13830.4 18688.0 17984.0 2304.0 2039.3 14 0.118 1 0.023 0.141
896.0 896.0 84.9 0.0 7488.0 794.3 18508.0 13830.4 18688.0 17984.0 2304.0 2039.3 14 0.118 1 0.023 0.141
896.0 896.0 84.9 0.0 7488.0 794.3 18508.0 13830.4 18688.0 17984.0 2304.0 2039.3 14 0.118 1 0.023 0.141
896.0 896.0 84.9 0.0 7488.0 794.3 18508.0 13830.4 18688.0 17984.0 2304.0 2039.3 14 0.118 1 0.023 0.141
896.0 896.0 84.9 0.0 7488.0 882.8 18508.0 13830.4 18688.0 17984.0 2304.0 2039.3 14 0.118 1 0.023 0.141
```

3.2.4 jstack

(1) 查看线程堆栈信息

The jstack command prints Java stack traces of Java threads for a specified Java process, core file, or remote debug server.

(2) 用法

jstack PID

```
[root@localhost bin]# jstack 2597
2019-06-09 03:18:20
Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.191-b12 mixed mode):

"Attach Listener" #47 daemon prio=9 os_prio=0 tid=0x00007ff81c002000 nid=0xacf waiting on condition [0x0000000000000000]
java.lang.Thread.State: RUNNABLE

"ajp-nio-8009-AsyncTimeout" #45 daemon prio=5 os_prio=0 tid=0x00007ff844532800 nid=0xa5c waiting on condition [0x00007ff8134f3000]
java.lang.Thread.State: TIMED_WAITING (sleeping)
    at java.lang.Thread.sleep(Native Method)
    at org.apache.coyote.AbstractProtocol$AsyncTimeout.run(AbstractProtocol.java:1149)
    at java.lang.Thread.run(Thread.java:748)

"ajp-nio-8009-Acceptor-0" #44 daemon prio=5 os_prio=0 tid=0x00007ff844530800 nid=0xa5b runnable [0x00007ff8135f4000]
java.lang.Thread.State: RUNNABLE
    at sun.nio.ch.ServerSocketChannelImpl.accept0(Native Method)
    at sun.nio.ch.ServerSocketChannelImpl.accept(ServerSocketChannelImpl.java:422)
    at sun.nio.ch.ServerSocketChannelImpl.accept(ServerSocketChannelImpl.java:250)
    - locked <0x00000000f61143f0> (a java.lang.Object)
    at org.apache.tomcat.util.net.NioEndpoint$Acceptor.run(NioEndpoint.java:482)
    at java.lang.Thread.run(Thread.java:748)

"ajp-nio-8009-ClientPoller-0" #43 daemon prio=5 os_prio=0 tid=0x00007ff84452e800 nid=0xa5a runnable [0x00007ff8136f5000]
java.lang.Thread.State: RUNNABLE
    at sun.nio.ch.EPollArrayWrapper.epollWait(Native Method)
    at sun.nio.ch.EPollArrayWrapper.poll(EPollArrayWrapper.java:269)
    at sun.nio.ch.EPollSelectorImpl.doSelect(EPollSelectorImpl.java:93)
    at sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:86)
    - locked <0x00000000f1230e40> (a sun.nio.ch.Util$3)
    - locked <0x00000000f1230e30> (a java.util.Collections$UnmodifiableSet)
    - locked <0x00000000f1230e50> (a sun.nio.ch.EPollSelectorImpl)
    at sun.nio.ch.SelectorImpl.select(SelectorImpl.java:97)
    at org.apache.tomcat.util.net.NioEndpoint$Poller.run(NioEndpoint.java:825)
    at java.lang.Thread.run(Thread.java:748)
```

(4) 排查死锁案例

- DeadLockDemo

```
//运行主类
public class DeadLockDemo
{
    public static void main(String[] args)
    {
        DeadLock d1=new DeadLock(true);
        DeadLock d2=new DeadLock(false);
        Thread t1=new Thread(d1);
        Thread t2=new Thread(d2);
        t1.start();
        t2.start();
    }
}

//定义锁对象
class MyLock{
    public static Object obj1=new Object();
    public static Object obj2=new Object();
}

//死锁代码
class DeadLock implements Runnable{
    private boolean flag;
    DeadLock(boolean flag){
        this.flag=flag;
    }
    public void run() {
        if(flag) {
```



```
Found one Java-level deadlock:
=====
"Thread-1":
  waiting to lock monitor 0x000000001cd9d9c8 (object 0x0000000076b3eb910, a java.lang.Object),
  which is held by "Thread-0"
"Thread-0":
  waiting to lock monitor 0x000000001cda0468 (object 0x0000000076b3eb920, a java.lang.Object),
  which is held by "Thread-1"

Java stack information for the threads listed above:
=====
"Thread-1":
  at com.gupao.gupaojvm.demo.DeadLock.run(DeadLockDemo.java:45)
  - waiting to lock <0x0000000076b3eb910> (a java.lang.Object)
  - locked <0x0000000076b3eb920> (a java.lang.Object)
  at java.lang.Thread.run(Thread.java:748)
"Thread-0":
  at com.gupao.gupaojvm.demo.DeadLock.run(DeadLockDemo.java:35)
  - waiting to lock <0x0000000076b3eb920> (a java.lang.Object)
  - locked <0x0000000076b3eb910> (a java.lang.Object)
  at java.lang.Thread.run(Thread.java:748)

Found 1 deadlock.
```

3.2.5 jmap

(1) 生成堆转储快照

The jmap command prints shared object memory maps or heap memory details of a specified process, core file, or remote debug server.

(2) 打印出堆内存相关信息

```
jmap -heap PID
```

```
jinfo -flag UsePSAdaptiveSurvivorSizePolicy 35352
-XX:SurvivorRatio=8
```

```
[root@localhost bin]# jmap -heap 2597
Attaching to process ID 2597, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.191-b12

using thread-local object allocation.
Mark Sweep Compact GC

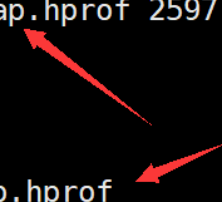
Heap Configuration:
  MinHeapFreeRatio      = 40
  MaxHeapFreeRatio      = 70
  MaxHeapSize           = 257949696 (246.0MB)
  NewSize               = 5570560 (5.3125MB)
  MaxNewSize            = 85983232 (82.0MB)
  OldSize               = 11206656 (10.6875MB)
  NewRatio              = 2
  SurvivorRatio         = 8
  MetaspaceSize         = 21807104 (20.796875MB)
  CompressedClassSpaceSize = 1073741824 (1024.0MB)
  MaxMetaspaceSize      = 17592186044415 MB
  G1HeapRegionSize      = 0 (0.0MB)

Heap Usage:
New Generation (Eden + 1 Survivor Space):
```

(3) dump出堆内存相关信息

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

```
[root@localhost tmp]# jmap -dump:format=b,file=heap.hprof 2597
Dumping heap to /tmp/heap.hprof ...
File exists
[root@localhost tmp]# ll
total 26300
-rw-----. 1 root root 26930127 Jun  9 03:26 heap.hprof
```



(4) 要是在发生堆内存溢出的时候，能自动dump出该文件就好了

一般在开发中，JVM参数可以加上下面两句，这样内存溢出时，会自动dump出该文件

```
-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=heap.hprof
```

设置堆内存大小：-Xms20M -Xmx20M
启动，然后访问localhost:9090/heap，使得堆内存溢出

(5) 关于dump下来的文件

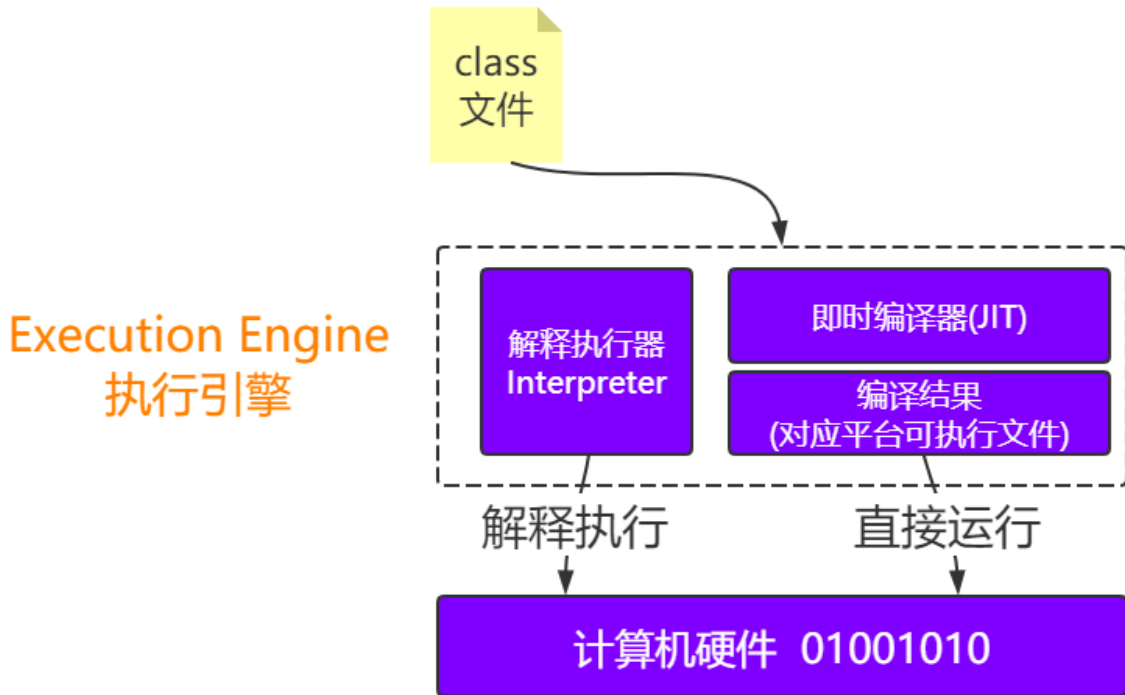
一般dump下来的文件可以结合工具来分析，这块后面再说。

执行引擎

Person.java源码文件是Java这门高级开发语言，对程序员友好，方便我们开发。

javac编译器将Person.java源码文件编译成class文件[我们把这里的编译称为前期编译]，交给JVM运行，因为JVM只能认识class字节码文件。同时在不同的操作系统上安装对应版本的JDK，里面包含了各自屏蔽操作系统底层细节的JVM，这样同一份class文件就能运行在不同的操作系统平台之上，得益于JVM。这也是Write Once, Run Anywhere的原因所在。

最终JVM需要把字节码指令转换为机器码，可以理解为是0101这样的机器语言，这样才能运行在不同的机器上，那么由字节码转变为机器码是谁来做的呢？说白了就是谁来执行这些字节码指令的呢？这就是执行引擎。



3.4.1 解释执行

Interpreter，解释器逐条把字节码翻译成机器码并执行，跨平台的保证。

刚开始执行引擎只采用了解释执行的，但是后来发现某些方法或者代码块被调用执行的特别频繁时，就会把这些代码认定为“热点代码”。

3.4.2 即时编译器

Just-In-Time compilation(JIT)，即时编译器先将字节码编译成对应平台的可执行文件，运行速度快。

即时编译器会把这些热点代码编译成与本地平台关联的机器码，并且进行各层次的优化，保存到内存中。

3.4.3 JVM采用哪种方式

JVM采取的是混合模式，也就是解释+编译的方式，对于大部分不常用的代码，不需要浪费时间将其编译成机器码，只需要用到的时候再以解释的方式运行；对于小部分的热点代码，可以采取编译的方式，追求更高的运行效率。

3.4.4 即使编译器类型

(1) HotSpot虚拟机里面内置了两个JIT：C1和C2

C1也称为Client Compiler，适用于执行时间短或者对启动性能有要求的程序
C2也称为Server Compiler，适用于执行时间长或者对峰值性能有要求的程序

(2) Java7开始，HotSpot会使用分层编译的方式

也就是会结合C1的启动性能优势和C2的峰值性能优势，热点方法会先被C1编译，然后热点方法中的热点会被C2再次编译

3.4.5 AOT和Graal VM

3.4.5.1 AOT

在Java9中，引入了AOT(Ahead-Of-Time)编译器

即时编译器是在程序运行过程中，将字节码翻译成机器码。而AOT是在程序运行之前，将字节码转换为机器码

优势：这样不需要在运行过程中消耗计算机资源来进行即时编译

劣势：AOT 编译无法得知程序运行时的信息，因此也无法进行基于类层次分析的完全虚方法内联，或者基于程序 profile 的投机性优化（并非硬性限制，我们可以通过限制运行范围，或者利用上一次运行的程序 profile 来绕过这两个限制）

3.4.5.2 Graal VM

官网：<https://www.oracle.com/tools/graalvm-enterprise-edition.html>

GraalVM core features include:

- GraalVM Native Image, available as an early access feature -- allows scripted applications to be compiled ahead of time into a native machine-code binary
- GraalVM Compiler -- generates compiled code to run applications on a JVM, standalone, or embedded in another system
- Polyglot Capabilities -- supports Java, Scala, Kotlin, JavaScript, and Node.js
- Language Implementation Framework -- enables implementing any language for the GraalVM environment
- LLVM Runtime-- permits native code to run in a managed environment in GraalVM Enterprise

在Java10中，新的JIT编译器Graal被引入

它是一个以Java为主要编程语言，面向字节码的编译器。跟C++实现的C1和C2相比，模块化更加明显，也更加容易维护。

Graal既可以作为动态编译器，在运行时编译热点方法；也可以作为静态编译器，实现AOT编译。

除此之外，它还移除了编程语言之间的边界，并且支持通过即时编译技术，将混杂了不同的编程语言的代码编译到同一段二进制码之中，从而实现不同语言之间的无缝切换。