

01 JVM实战篇

1.1 JVM参数

1.1.1 标准参数

```
-version  
-help  
-server  
-cp
```

```
[root@localhost ~]# java -version  
java version "1.8.0_191"  
Java(TM) SE Runtime Environment (build 1.8.0_191-b12)  
Java HotSpot(TM) 64-Bit Server VM (build 25.191-b12, mixed mode)
```

1.1.2 -X参数

非标准参数，也就是在JDK各个版本中可能会变动

```
-Xint      解释执行  
-Xcomp     第一次使用就编译成本地代码  
-Xmixed    混合模式，JVM自己来决定
```

```
[root@localhost ~]# java -version  
java version "1.8.0_191"  
Java(TM) SE Runtime Environment (build 1.8.0_191-b12)  
Java HotSpot(TM) 64-Bit Server VM (build 25.191-b12, mixed mode)  
[root@localhost ~]# java -Xint -version  
java version "1.8.0_191"  
Java(TM) SE Runtime Environment (build 1.8.0_191-b12)  
Java HotSpot(TM) 64-Bit Server VM (build 25.191-b12, interpreted mode)  
[root@localhost ~]# java -Xcomp -version  
java version "1.8.0_191"  
Java(TM) SE Runtime Environment (build 1.8.0_191-b12)  
Java HotSpot(TM) 64-Bit Server VM (build 25.191-b12, compiled mode)  
[root@localhost ~]# java -Xmixed -version  
java version "1.8.0_191"  
Java(TM) SE Runtime Environment (build 1.8.0_191-b12)  
Java HotSpot(TM) 64-Bit Server VM (build 25.191-b12, mixed mode)
```

1.1.3 -XX参数

使用得最多的参数类型

非标准化参数，相对不稳定，主要用于JVM调优和Debug

a. Boolean类型

格式: `-XX:[+-]<name>` +或-表示启用或者禁用name属性
比如: `-XX:+UseConcMarkSweepGC` 表示启用CMS类型的垃圾回收器
 `-XX:+UseG1GC` 表示启用G1类型的垃圾回收器

b. 非Boolean类型

格式: `-XX<name>=<value>`表示name属性的值是value
比如: `-XX:MaxGCPauseMillis=500`

1.1.4 其他参数

`-Xms1000`等价于`-XX:InitialHeapSize=1000`
`-Xmx1000`等价于`-XX:MaxHeapSize=1000`
`-Xss100`等价于`-XX:ThreadStackSize=100`

所以这块也相当于-XX类型的参数

1.1.5 查看参数

```
java -XX:+PrintFlagsFinal -version > flags.txt
```

```
[root@localhost bin]# java -XX:+PrintFlagsFinal -version > flags.txt
java version "1.8.0_191"
Java(TM) SE Runtime Environment (build 1.8.0_191-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.191-b12, mixed mode)
[root@localhost bin]# sz flags.txt
```

```
[Global flags]
  intx ActiveProcessorCount           = -1           {product}
  uintx AdaptiveSizeDecrementScaleFactor = 4           {product}
  uintx AdaptiveSizeMajorGCDecayTimeScale = 10          {product}
  uintx AdaptiveSizePausePolicy         = 0           {product}
  uintx AdaptiveSizePolicyCollectionCostMargin = 50          {product}
  uintx AdaptiveSizePolicyInitializingSteps = 20          {product}
  uintx AdaptiveSizePolicyOutputInterval = 0           {product}
  uintx AdaptiveSizePolicyWeight        = 10          {product}
  uintx AdaptiveSizeThroughPutPolicy     = 0           {product}
  uintx AdaptiveTimeWeight              = 25          {product}
  bool  AdjustConcurrency               = false        {product}
  bool  AggressiveHeap                 = false        {product}
  bool  AggressiveOpts                 = false        {product}
```

值得注意的是"="表示默认值, ":"表示被用户或JVM修改后的值

要想查看某个进程具体参数的值, 可以使用jinfo, 这块后面聊

一般要设置参数, 可以先查看一下当前参数是什么, 然后进行修改

1.1.6 设置参数的方式

- 开发工具中设置比如IDEA, eclipse
- 运行jar包的时候: `java -XX:+UseG1GC xxx.jar`
- web容器比如tomcat, 可以在脚本中的进行设置
- 通过jinfo实时调整某个java进程的参数(参数只有被标记为manageable的flags可以被实时修改)

1.1.7 实践和单位换算

1Byte(字节)=8bit(位)

1KB=1024Byte(字节)

1MB=1024KB

1GB=1024MB

1TB=1024GB

(1) 设置堆内存大小和参数打印

-Xmx100M -Xms100M -XX:+PrintFlagsFinal

(2) 查询+PrintFlagsFinal的值

:=true

(3) 查询堆内存大小MaxHeapSize

:= 104857600

(4) 换算

104857600(Byte)/1024=102400(KB)

102400(KB)/1024=100(MB)

(5) 结论

104857600是字节单位

1.1.8 常用参数含义

咕泡学院 只为更好的你

参数	含义	说明
-XX:CICompilerCount=3	最大并行编译数	如果设置大于1，虽然编译速度会提高，但是同样影响系统稳定性，会增加JVM崩溃的可能
-XX:InitialHeapSize=100M	初始化堆大小	简写-Xms100M
-XX:MaxHeapSize=100M	最大堆大小	简写-Xmx100M
-XX:NewSize=20M	设置年轻代的大小	
-XX:MaxNewSize=50M	年轻代最大大小	
-XX:OldSize=50M	设置老年代大小	
-XX:MetaspaceSize=50M	设置方法区大小	
-XX:MaxMetaspaceSize=50M	方法区最大大小	
-XX:+UseParallelGC	使用UseParallelGC	新生代，吞吐量优先
-XX:+UseParallelOldGC	使用UseParallelOldGC	老年代，吞吐量优先
-XX:+UseConcMarkSweepGC	使用CMS	老年代，停顿时间优先
-XX:+UseG1GC	使用G1GC	新生代，老年代，停顿时间优先
-XX:NewRatio	新老生代的比值	比如-XX:Ratio=4，则表示新生代:老年代=1:4，也就是新生代占整个堆内存的1/5
-XX:SurvivorRatio	两个S区和Eden区的比值	比如-XX:SurvivorRatio=8，也就是(S0+S1):Eden=2:8，也就是一个S占整个新生代的1/10
-XX:+HeapDumpOnOutOfMemoryError	启动堆内存溢出打印	当JVM堆内存发生溢出时，也就是OOM，自动生成dump文件
-XX:HeapDumpPath=heap.hprof	指定堆内存溢出打印目录	表示在当前目录生成一个heap.hprof文件
XX:+PrintGCDetails - XX:+PrintGCTimeStamps - XX:+PrintGCDateStamps Xloggc:\$CATALINA_HOME/logs/gc.log	打印出GC日志	可以使用不同的垃圾收集器，对比查看GC情况
-Xss128k	设置每个线程的堆栈大小	经验值是3000-5000最佳
-XX:MaxTenuringThreshold=6	提升年老代的最大临界值	默认值为 15
-XX:InitiatingHeapOccupancyPercent	启动并发GC周期时堆内存使用占比	G1之类的垃圾收集器用它来触发并发GC周期,基于整个堆的使用率,而不只是某一代内存的使用比. 值为 0 则表示“一直执行GC循环”. 默认值为 45.
-XX:G1HeapWastePercent	允许的浪费堆空间的占比	默认是10%，如果并发标记可回收的空间小于10%,则不会触发MixedGC。
-XX:MaxGCPauseMillis=200ms	G1最大停顿时间	暂停时间不能太小，太小的话就会导致出现G1跟不上垃圾产生的速度。最终退化成全GC。所以对这个参数的调优是一个持续的过程，逐步调整到最佳状态。
-XX:ConcGCThreads=n	并发垃圾收集器使用的线程数量	默认值随JVM运行的平台不同而不同
-XX:G1MixedGCLiveThresholdPercent=65	混合垃圾回收周期中要包括的旧区域设置占用率阈值	默认占用率为 65%
-XX:G1MixedGCCountTarget=8	设置标记周期完成后，对存活数据上限为G1MixedGCLiveThresholdPercent的旧区域执行混合垃圾回收的目标次数	默认8次混合垃圾回收，混合回收的目标是要控制在此目标次数以内
- XX:G1OldCSetRegionThresholdPercent=1	描述Mixed GC时，Old Region被加入到CSet中	默认情况下，G1只把10%的Old Region加入到CSet中

1.2 常用命令

1.2.1 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
2597 Bootstrap
2616 Jps
[root@localhost bin]# jps -l
2597 org.apache.catalina.startup.Bootstrap
2653 sun.tools.jps.Jps
```

1.2.2 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

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 = PID
```

(4) 查看曾经赋过值的一些参数

```
jinfo -flags PID
```

```
[root@localhost bin]# jinfo -flags 2597
Attaching to process ID 2597, 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.util
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
```

1.2.3 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`
次

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
```

1.2.4 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) {
            while(true) {
                synchronized(MyLock.obj1) {
                    System.out.println(Thread.currentThread().getName()+"----if
获得obj1锁");
                    synchronized(MyLock.obj2) {
                        System.out.println(Thread.currentThread().getName()+"---
-if获得obj2锁");
                    }
                }
            }
        }
    }
}
```



```

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.

```

1.2.5 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

```

-XX:+PrintFlagsFinal -Xms300M -Xmx300M
jmap -heap PID

```

```

[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
  GLHeapRegionSize       = 0 (0.0MB)

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

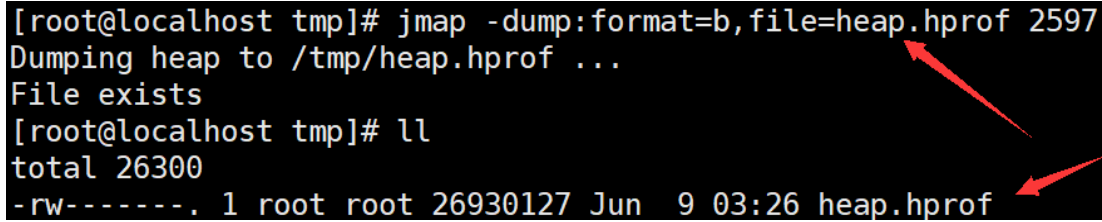
```

(3)dump出堆内存相关信息

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

```
jmap -dump:format=b,file=heap.hprof 44808
```

```
[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。

1.3 常用工具

参数也了解了，命令也知道了，关键是用起来不是很方便，要是图形化的界面就好了。

一定会有好事之者来做这件事情。

1.3.1 jconsole

JConsole工具是JDK自带的可视化监控工具。查看java应用程序的运行概况、监控堆信息、永久区使用情况、类加载情况等。

命令行中输入：jconsole

1.3.2 jvisualvm

1.3.2.1 监控本地Java进程

可以监控本地的java进程的CPU，类，线程等

1.3.2.2 监控远端Java进程

比如监控远端tomcat，演示部署在阿里云服务器上的tomcat

(1)在visualvm中选中“远程”，右击“添加”

(2)主机名上写服务器的ip地址，比如31.100.39.63，然后点击“确定”

(3)右击该主机“31.100.39.63”，添加“JMX”[也就是通过JMX技术具体监控远端服务器哪个Java进程]

(4)要想让服务器上的tomcat被连接，需要改一下 bin/catalina.sh 这个文件

注意下面的8998不要和服务器上其他端口冲突

```
JAVA_OPTS="$JAVA_OPTS -Dcom.sun.management.jmxremote -  
Djava.rmi.server.hostname=31.100.39.63 -Dcom.sun.management.jmxremote.port=8998  
-Dcom.sun.management.jmxremote.ssl=false -  
Dcom.sun.management.jmxremote.authenticate=true -  
Dcom.sun.management.jmxremote.access.file=../conf/jmxremote.access -  
Dcom.sun.management.jmxremote.password.file=../conf/jmxremote.password"
```

(5)在 `../conf` 文件中添加两个文件jmxremote.access和jmxremote.password

jmxremote.access 文件

```
guest readonly  
manager readwrite
```

jmxremote.password 文件

```
guest guest  
manager manager
```

授予权限: `chmod 600 *jmxremot*`

(6)将连接服务器地址改为公网ip地址

```
hostname -i 查看输出情况  
172.26.225.240 172.17.0.1  
vim /etc/hosts  
172.26.255.240 31.100.39.63
```

(7)设置上述端口对应的阿里云安全策略和防火墙策略

(8)启动tomcat, 来到bin目录

```
./startup.sh
```

(9)查看tomcat启动日志以及端口监听

```
tail -f ../logs/catalina.out  
lsof -i tcp:8080
```

(10)查看8998监听情况, 可以发现多开了几个端口

```
lsof -i:8998 得到PID  
netstat -antup | grep PID
```

(11)在刚才的JMX中输入8998端口, 并且输入用户名和密码则登录成功

```
端口:8998  
用户名:manager  
密码:manager
```

1.3.3 Arthas

github: <https://github.com/alibaba/arthas>

Arthas allows developers to troubleshoot production issues for Java applications without modifying code or restarting servers.

Arthas 是Alibaba开源的Java诊断工具，采用**命令行交互模式**，是排查jvm相关问题的利器。

Arthas 是Alibaba开源的Java诊断工具，深受开发者喜爱。

当你遇到以下类似问题而束手无策时，Arthas 可以帮助你解决：

1. 这个类从哪个 jar 包加载的？为什么会报各种类相关的 Exception？
1. 我改的代码为什么没有执行到？难道是我没 commit？分支搞错了？
2. 遇到问题无法在线上 debug，难道只能通过加日志再重新发布吗？
3. 线上遇到某个用户的数据处理有问题，但线上同样无法 debug，线下无法重现！
4. 是否有一个全局视角来查看系统的运行状况？
5. 有什么办法可以监控到JVM的实时运行状态？
6. 怎么快速定位应用的热点，生成火焰图？

Arthas 支持JDK 6+，支持Linux/Mac/Windows，采用命令行交互模式，同时提供丰富的 Tab 自动补全功能，进一步方便进行问题的定位和诊断。

1.3.3.1 下载安装

```
curl -O https://alibaba.github.io/arthas/arthas-boot.jar
java -jar arthas-boot.jar
# 然后可以选择一个Java进程
```

Print usage

```
java -jar arthas-boot.jar -h
```

1.3.3.2 常用命令

具体每个命令怎么使用，大家可以自己查阅资料

```
version: 查看arthas版本号
help: 查看命名帮助信息
cls: 清空屏幕
session: 查看当前会话信息
quit: 退出arthas客户端
---
dashboard: 当前进程的实时数据面板
thread: 当前JVM的线程堆栈信息
jvm: 查看当前JVM的信息
sysprop: 查看JVM的系统属性
---
sc: 查看JVM已经加载的类信息
dump: dump已经加载类的byte code到特定目录
jad: 反编译指定已加载类的源码
---
monitor: 方法执行监控
watch: 方法执行数据观测
trace: 方法内部调用路径，并输出方法路径上的每个节点上耗时
stack: 输出当前方法被调用的调用路径
.....
```

1.3.4 MAT

Java堆分析器，用于查找内存泄漏

Heap Dump，称为堆转储文件，是Java进程在某个时间内的快照

下载地址: <https://www.eclipse.org/mat/downloads.php>

1.3.4.1 Dump信息包含的内容

- All Objects

Class, fields, primitive values and references

- All Classes

ClassLoader, name, super class, static fields

- Garbage Collection Roots

Objects defined to be reachable by the JVM

- Thread Stacks and Local Variables

The call-stacks of threads at the moment of the snapshot, and per-frame information about local objects

1.3.4.2 获取Dump文件

- 手动

```
jmap -dump:format=b,file=heap.hprof 44808
```

- 自动

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

1.3.4.3 使用

- Histogram

Histogram可以列出内存中的对象，对象的个数及其大小

Class Name:类名称，java类名

Objects:类的对象的数量，这个对象被创建了多少个

Shallow Heap:一个对象内存的消耗大小，不包含对其他对象的引用

Retained Heap:是shallow Heap的总和，即该对象被GC之后所能回收到内存的总和

右击类名--->List Objects--->with incoming references--->列出该类的实例

右击Java对象名--->Merge Shortest Paths to GC Roots--->exclude all ...--->找到GC Root以及原因

- Leak Suspects

查找并分析内存泄漏的可能原因

Reports--->Leak Suspects--->Details

- Top Consumers

列出大对象

1.3.4 GC日志分析工具

要想分析日志的信息，得先拿到GC日志文件才行，所以得先配置一下

根据前面参数的学习，下面的配置很容易看懂

```
-XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps  
-Xloggc:gc.log
```

- 在线

<http://gceasy.io>

- GCViewer

咕泡学院 只为更好的你