01 JVM参数

1.1 标准参数

> 不会因为Java版本的变化而变化

```
-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)
```

1.2 非标准参数

> 可能会因为Java版本的变化而变化

1.2.1 -X

```
-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
```

1.2.2 -XX

```
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
```

1.2.3 其他参数

```
-Xms1000M等价于 -XX:InitialHeapSize=1000M
-Xmx1000M等价于 -XX:MaxHeapSize=1000M
-Xss100等价于 -XX:ThreadStackSize=100
```

1.3 常见参数

官网: https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html

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VM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-Xms100M
-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:g1-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Full 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

1.4 在哪设置

（1）开发工具中设置比如IDEA，eclipse

（2）运行jar包的时候:java -XX:+UseG1GC xxx.jar

（3）中间件比如tomcat，可以在脚本中的进行设置

（4）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)

1.5 查看参数

（1）启动java进程时添加+PrintFlagsFinal参数

（2）通过jinfo命令查看，后面再聊

1.6 实践一下

```
(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
  设置成功，并且104857600是字节单位
```

02 常见命令

官网: https://docs.oracle.com/javase/8/docs/technotes/tools/unix/index.html

2.1 jps

```
jps
jps -l
```

2.2 jinfo

```
jinfo -flag name PID
jinfo -flag <name>=<value> PID
jinfo -flags PID
```

2.3 jstat

```
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
jstat -gc PID 1000 10
```

2.4 jstack

```
jstack PID
```

死锁案例

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

```
43   System.out.println(Thread.currentThread().getName()+"----否则获得obj1锁");
44   }
45   }
46   }
47   }
48   }
49 }
```

2.5 jmap

```
jmap -heap PID
jmap -dump:format=b,file=heap.hprof PID
-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=heap.hprof # 关于dump文件的分析，可以用一些工具，暂时先不讨论
```

03 JDK通用工具

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。

3.2 jvisualvm

可以监控某个java进程的CPU，类，线程等

> 连接远程java进程

（1）找一台linux机器，并且安装好tomcat

（2）修改bin/catalina.sh文件

```
JAVA_OPTS="$JAVA_OPTS -Dcom.sun.management.jmxremote -Djava.rmi.server.hostname=192.168.1.8 -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"
```

（3）在conf目录下添加两个文件

> jmxremote.password

```
guest guest
manager manager
```

> jmxremote.access

```
guest readonly
manager readwrite
```

（4）授予权限

```
chmod 600 jmxremote*
```

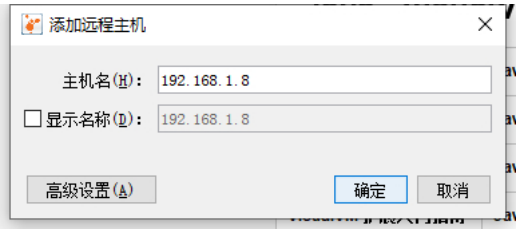
（5）启动tomcat并查看日志

```
./startup.sh
tail -f ../logs/catalina.out
```

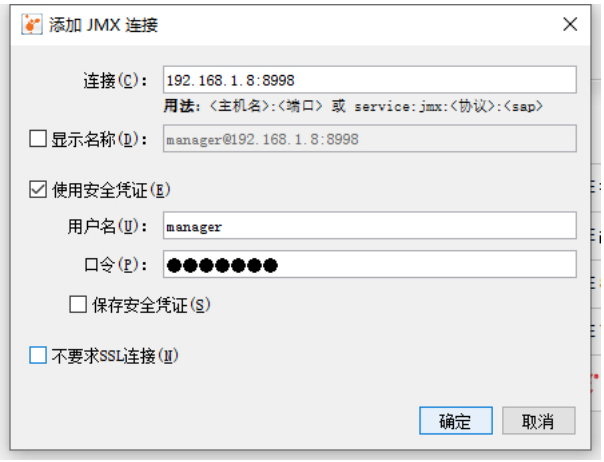
（6）查看端口监听情况

```
lsof -i:8080
lsof -i:8998
```

（7）打开本地的jvisualvm，并添加远程连接



（8）右击主机"192.168.1.8"，通过“jmx”进行添加，也就是通过JMX技术具体监控远端服务器哪个Java进程



04 第三方通用工具Arthas

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

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

（1）下载

```
curl -O https://alibaba.github.io/arthas/arthas-boot.jar
```

（2）启动

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

（3）常用命令

```
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
```

05 内存分析工具

5.1 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。
> 它在触发快照的时候保存了很多信息：Java对象和类信息。

（1）获取dump文件

```
jmap -dump:format=b,file=heap.hprof 44808
 或
-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=heap.hprof
```

（2）Histogram：可以列出内存中的对象、对象个数及大小

```
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
```

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

（4）右击Java对象名--->Merge Shortest Paths to GC Roots--->exclude all ...--->找到GCRoot以及原因

（5）Leak Suspects：查找并分析内存泄漏的可能原因Leak Suspects：查找并分析内存泄漏的可能原因

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

（6）Top Consumers：列出大对象

5.2 HeapHero

官网: https://heaphero.io/https://heaphero.io/

5.3 Perfma

官网: https://console.perfma.com

06 GC分析工具

6.1 GC日志

> 可以使用不同的参数设置不同的日志文件，比如

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

6.2 gcviewer

```
java -jar gcviewer-1.36-SNAPSHOT.jar
```

6.3 gceasy

官网: http://gceasy.io

6.4 gcplot

官网：https://it.gcplot.com/

```
docker run -d -p 8088:80 gcplot/gcplot
http://192.168.1.8:8088
```