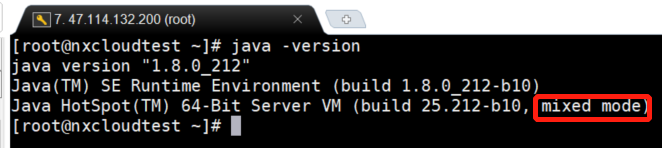
#### 基于JDK命令行工具的监控

##### JVM的参数类型

* 1. 标准参数
     1. -help
     2. -server -client
     3. -version -showversion
     4. -cp -classpath
  2. X参数（非标准化参数）
     1. -Xint：解释执行
     2. -Xcomp：第一次使用就编译成本地代码
     3. -Xmixed：混合模式，JVM自己来决定是否编译成本地代码
     4. 示例：



说明：mixed mode——由JVM自己决定是否编译成本地代码

* 1. XX参数
     1. Boolean类型
        1. 格式：-XX:[+-]<name>表示启用或者禁用name属性
        2. 示例：
           1. -XX:+UseConcMarkSweepGC 启用CMS垃圾收集器
           2. -XX:+UseG1GC 启用G1垃圾收集器
     2. 非Boolean类型
        1. 格式：-XX:<name>=<value> 表示name属性的值是value
        2. 示例：
           1. -XX:MaxGCPauseMillis=500 GC的最大停顿时间是500毫秒
           2. XX:GCTimeRatio=19
  2. 参数示例说明
     1. -Xmx -Xms 设置JVM的最大/最小内存
        1. 说明：不是X参数，而是XX参数
        2. 原型：
           1. -Xms 等价于 -XX:InitialHapSize
           2. -Xmx 等价于 -XX:MaxHeapSize
        3. 示例查询

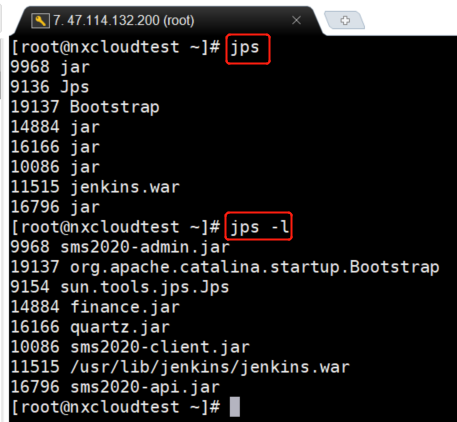


##### 运行时JVM参数查看

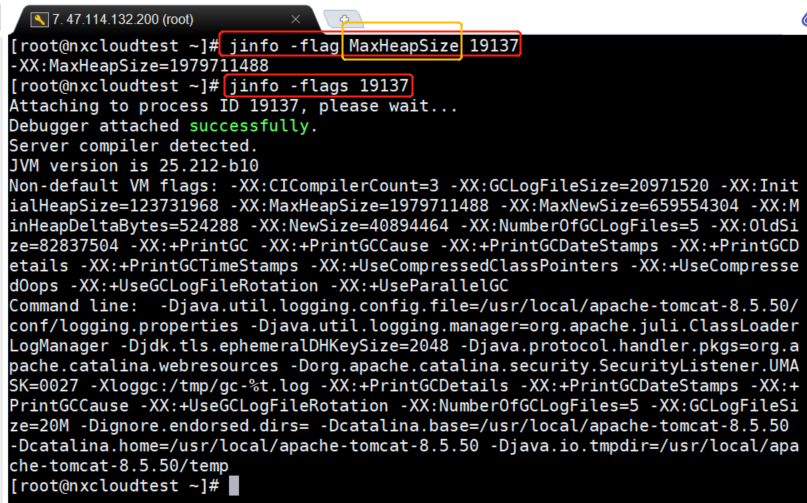
* 1. [Oracl官网查询JVM命令](https://docs.oracle.com/javase/8/docs/technotes/tools/unix/index.html)
  2. -XX:+PrintFlagsInitial 查看初始值
  3. -XX:+PrintFlagsFinal 查看最终的值
     1. 输入示例：java -XX:+PrintFlagsFinal
     2. 结果说明：



* 1. -XX:+UnlockExperimenttalVMOptions 解锁实验参数
  2. -XX:+UnlockDiagnosticVMOptions 解锁诊断参数
  3. -XX:+PrintCommandLineFlags 打印命令行参数
  4. 补充命令说明
     1. jps 查询在运行时程序的进程



* + 1. jinfo 查询在运行中的JVM参数

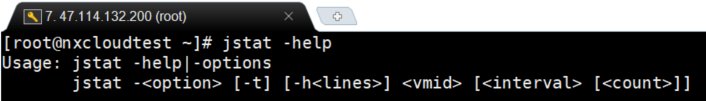


说明：可查询指定进程明确/全部的参数信息



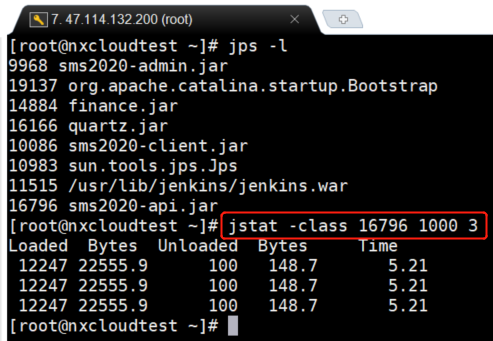
##### jstat查看虚拟机统计信息

* 1. 命令格式



options：-class，-compiler，-gc，-printcompilation

* 1. 类加载信息(-class)



填写参数说明：

16796：进程编号

1000：1000毫秒统计一次

3：总共统计三次

[返回信息说明](https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jstat.html" \l "BEHHGFAE) - 搜索“-class option”：

Loaded: 已加载的类数.

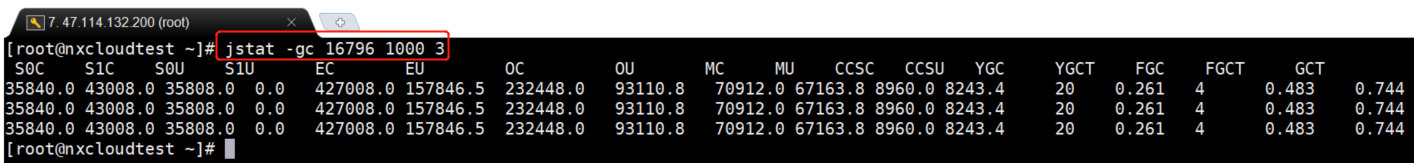
Bytes: 已加载的字节数.

Unloaded: 卸载的类数.

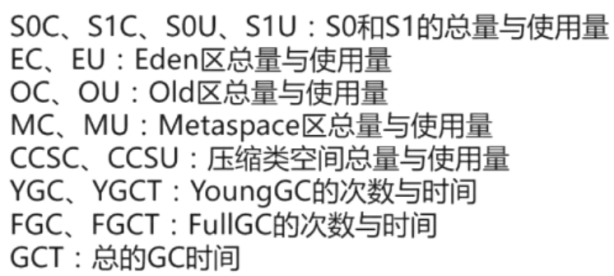
Bytes: 卸载的字节数.

Time: 执行类加载和卸载操作所花费的时间.

* 1. 垃圾收集信息(-gc、-gcutil、-gccause、-gcnew、-gcold.....)



[返回信息说明](https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jstat.html" \l "BEHHGFAE) - 搜索“-gc option”



标注：至于S0、S1有什么含义，后面讲

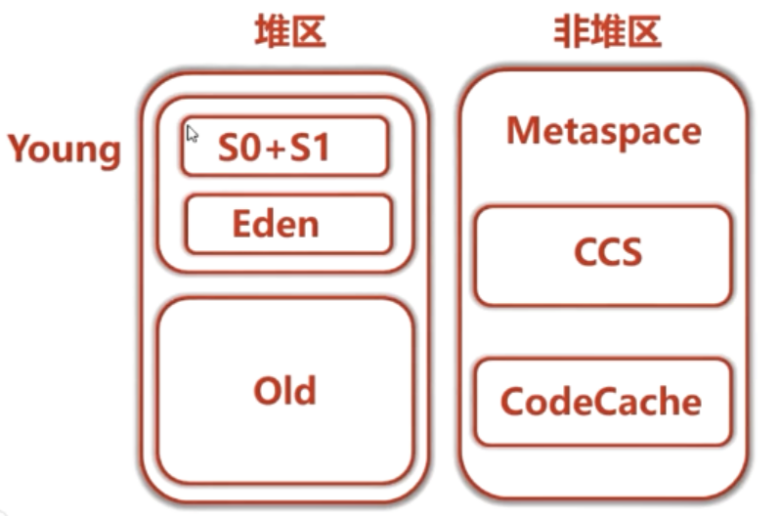
简单说明：S0 + S1 + E =young区

* 1. JIT编译信息(-compiler、-printcompilation)



[返回信息说明](https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jstat.html" \l "BEHHGFAE) - 搜索“-compiler option”

* 1. JVM的内存结构



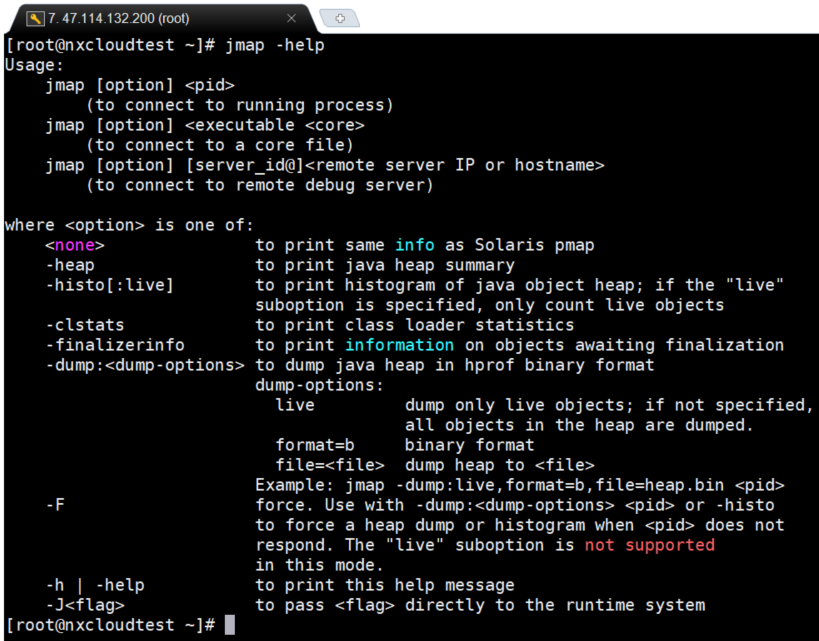
说明：S0 与 S1 大小一致，同个时间点上只有一个是启用的；CodeCache保存JIT的代码信息

##### jmap + MAT内存溢出(案例实战)

* 1. 代码编写堆内存溢出和非堆内存溢出

[MemoryDemo 内存溢出案例](https://github.com/gitjiangxb/studentCode/blob/java-juc/src/main/java/jvm/MemoryDemo.java)

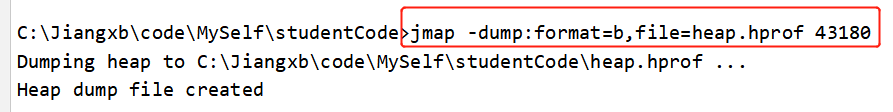
* 1. 如何导出内存映像文件
     1. 内存溢出自动导出(只需要配置参数)
        1. -XX:+HeapDumpOnOutOfMemoryError
        2. -XX:HeapDumpPath=./
     2. 使用jmap命令手动导出



option:-heap,-clstats,-dump:<dump-options>,-F

示例命令：

jmap -dump:format=b,file=heap.hprof 43180



参数说明：

format：格式

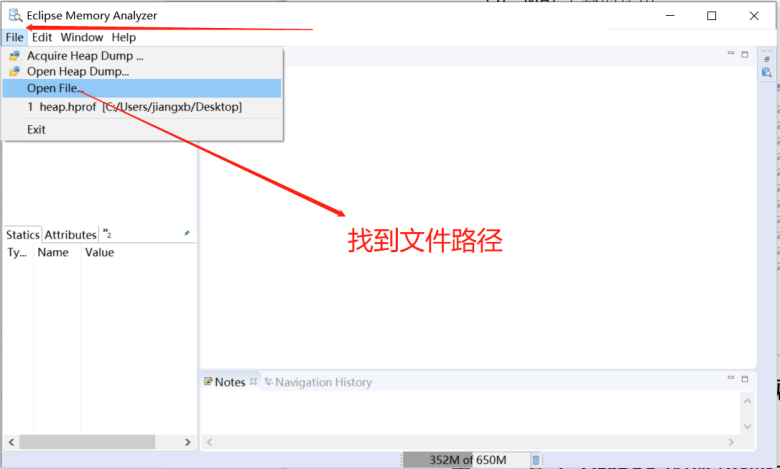
file：文件名称

43180：当前项目进程编号

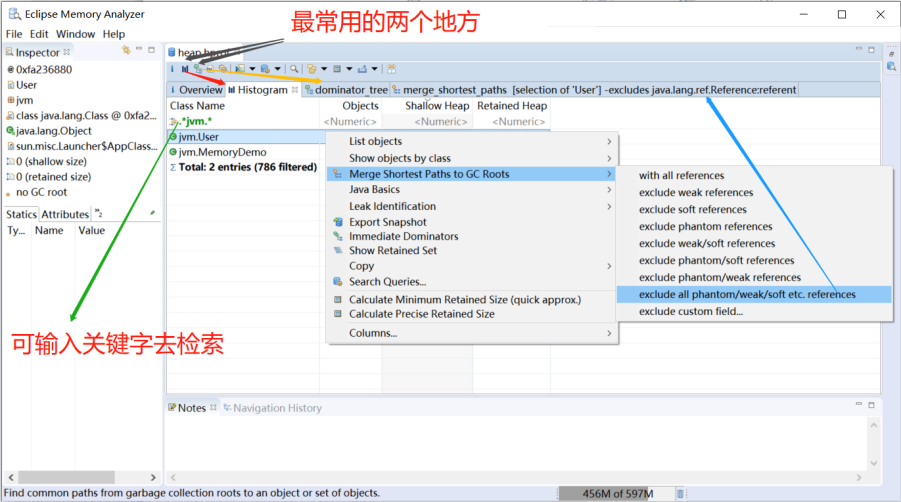
* 1. MAT工具的使用
     1. [下载地址](https://www.eclipse.org/downloads/download.php?file=/mat/1.10.0/rcp/MemoryAnalyzer-1.10.0.20200225-win32.win32.x86_64.zip)
     2. 解压完双击运行



* + 1. 导入内存映像文件

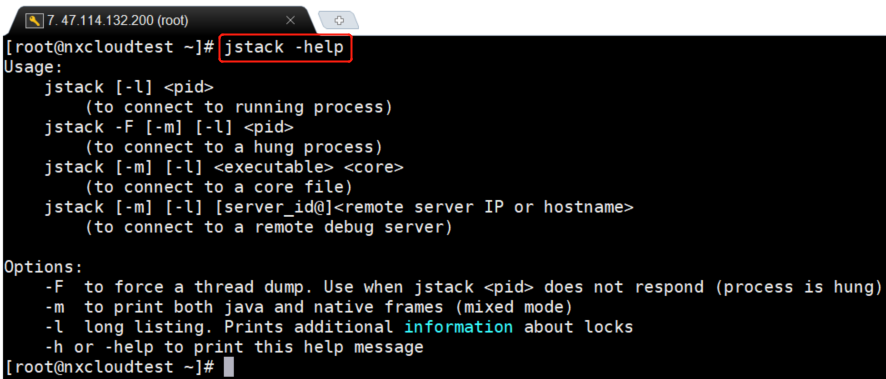


* + 1. 常用分析按钮



##### [jstack](https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jstack.html) 死循环与死锁(案例实战) -CPU过高

* 1. 命令格式

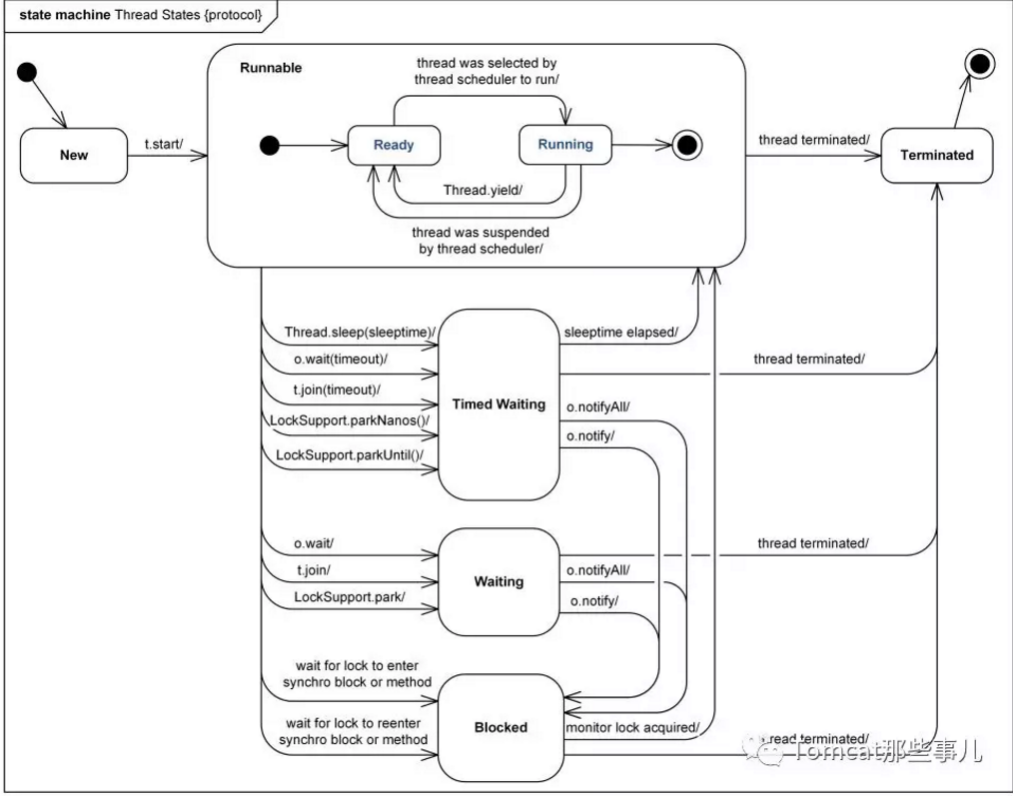


* 1. 案例分析

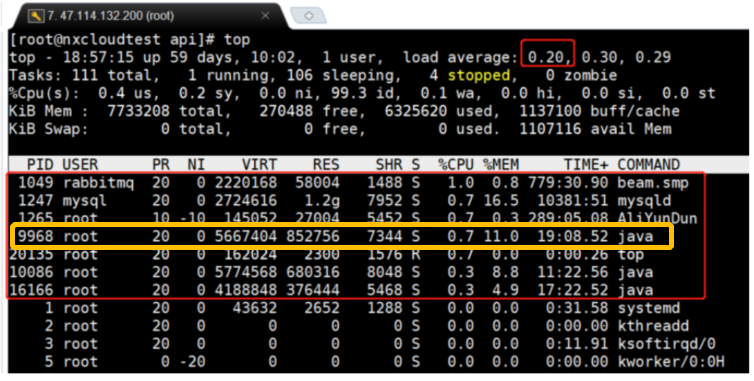


说明：注意跟java面试题(JUC+JVM+GC+Github).docx 的第二大点的第11点的区别

* 1. JAVA线程状态 [官网解释](https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/tooldescr034.html)
     1. NEW
     2. RUNNABLE
     3. BLOCKED
     4. WAITING
     5. TIMED\_WAITING
     6. TERMINATED
     7. [示例图地址](https://mp.weixin.qq.com/s/GsxeFM7QWuR--Kbpb7At2w)

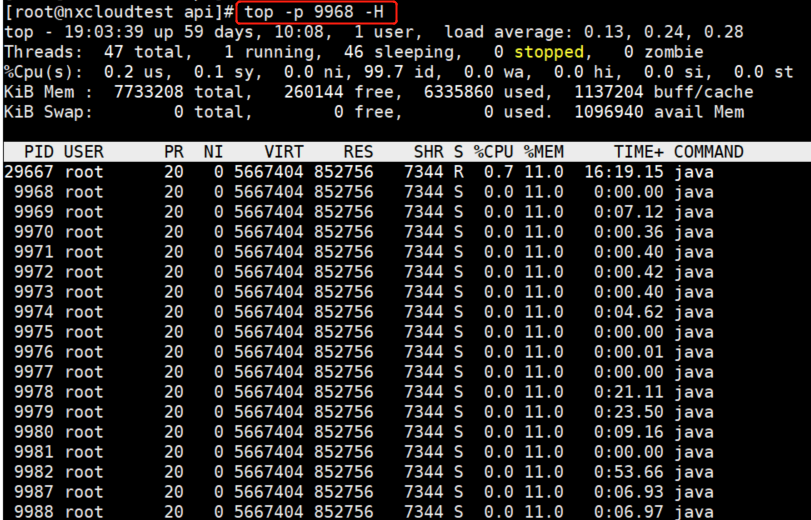


* 1. [代码实例（CPU过高-死循环）](https://github.com/gitjiangxb/studentCode/blob/java-juc/src/main/java/jvm/CPUDemo.java)
     1. 利用top命令打印出cpu负载



排在前面的就是有可能有问题的代码【假设黄线圈起来的就是有问题进程】

* + 1. 利用jstack命令：jstack 9968 > 9968.txt
    2. 然后打印出该进程(pid)中的线程：top -p 9968 -H



说明：排名前面的就是占用cpu高的线程【假设是：29667】

* + 1. 然后利用命令：printf "%x\n" 29967 ，将10进制的线程编号转换为16进制

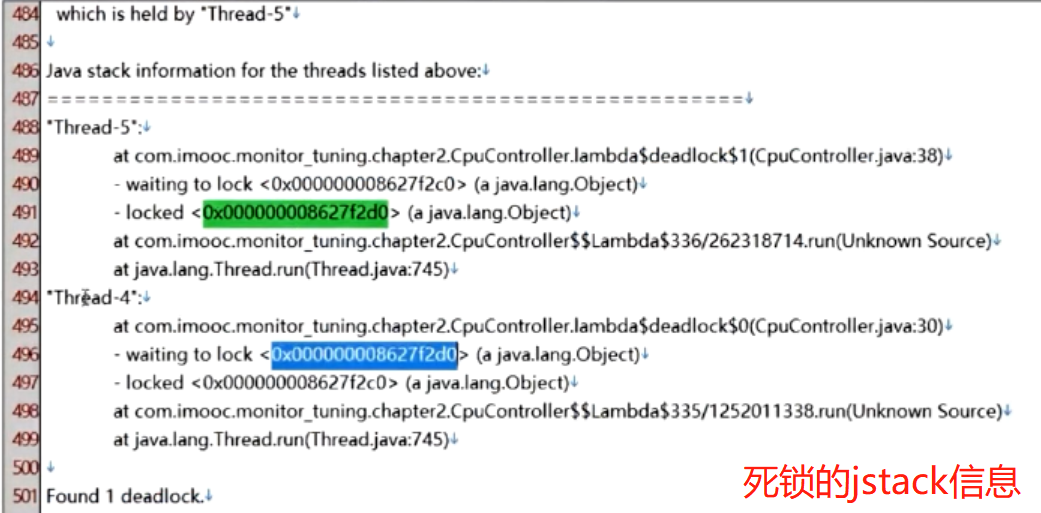
29967 转换后 750f

* + 1. 然后把转换后的“750f”带到刚刚的文件“ 9968.txt”里面去搜索。



说明：由于本例子没有部署到服务器，因此该图是借鉴

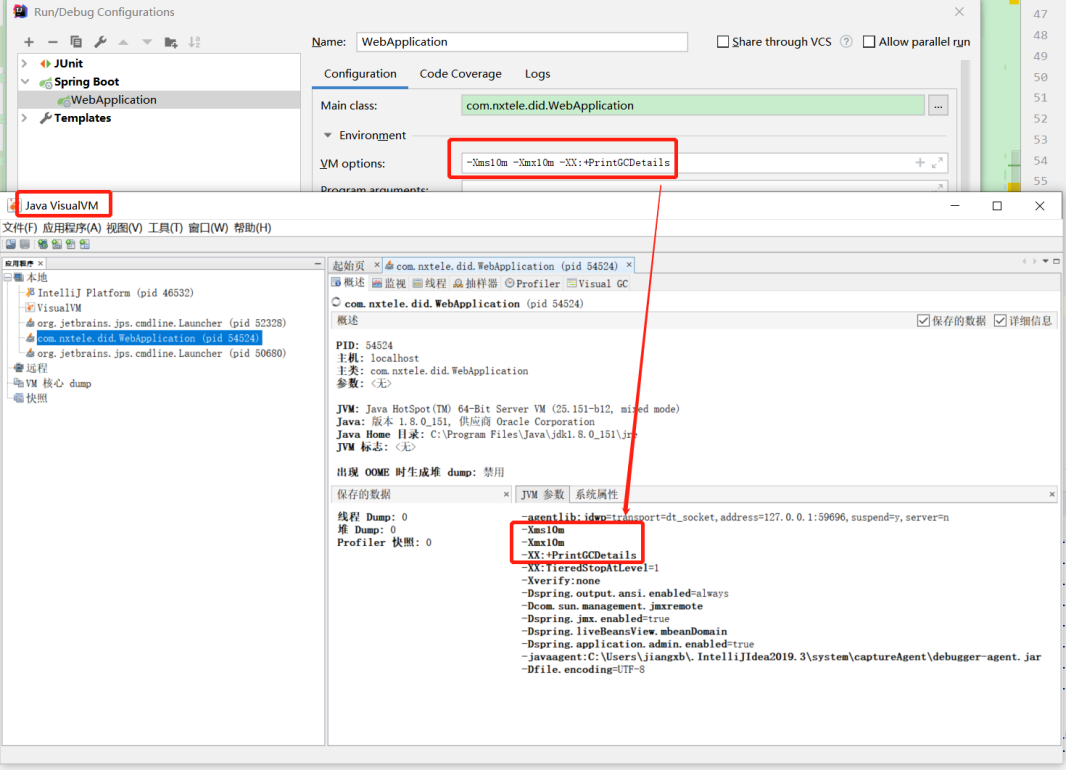
* + 1. 这样就快速定位出有问题的代码了
  1. 代码实例（代码里面的死锁）



#### 基于JVisualVM的可视化监控

[官网使用说明](https://visualvm.github.io/documentation.html)

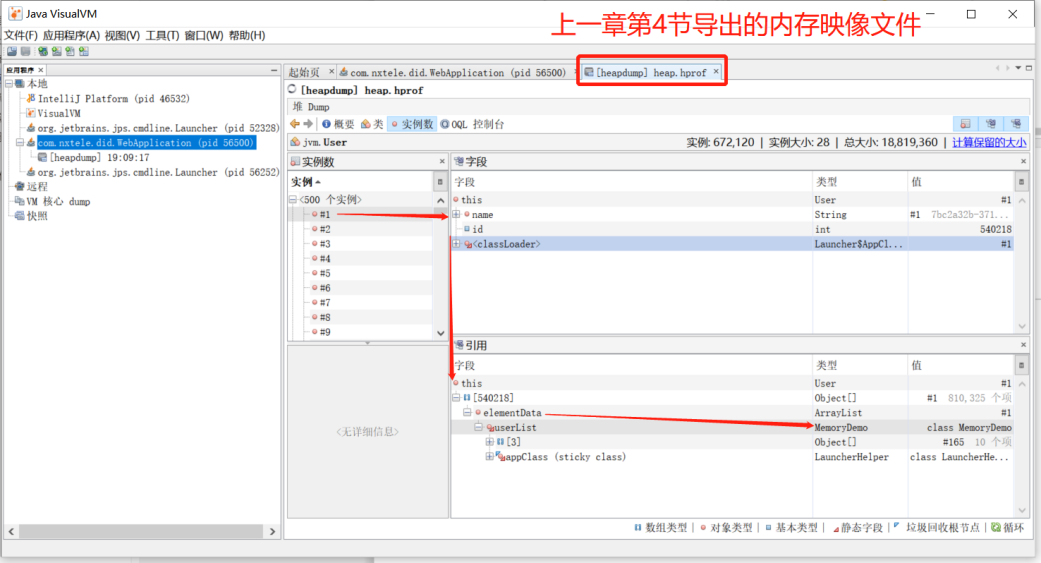
##### 监控本地的Tomcat/JAVA进程

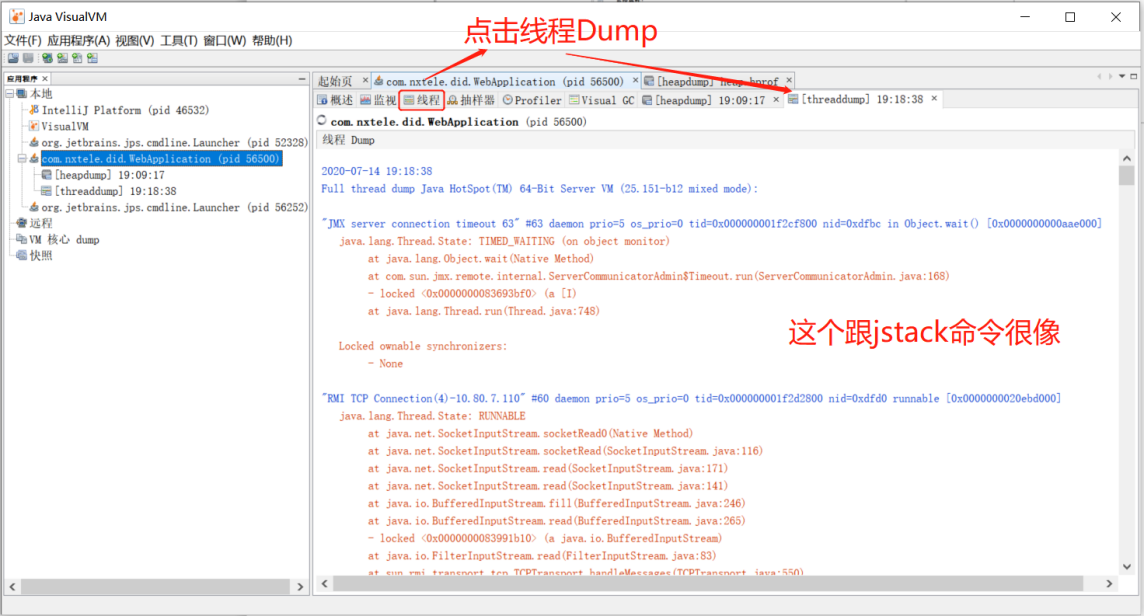


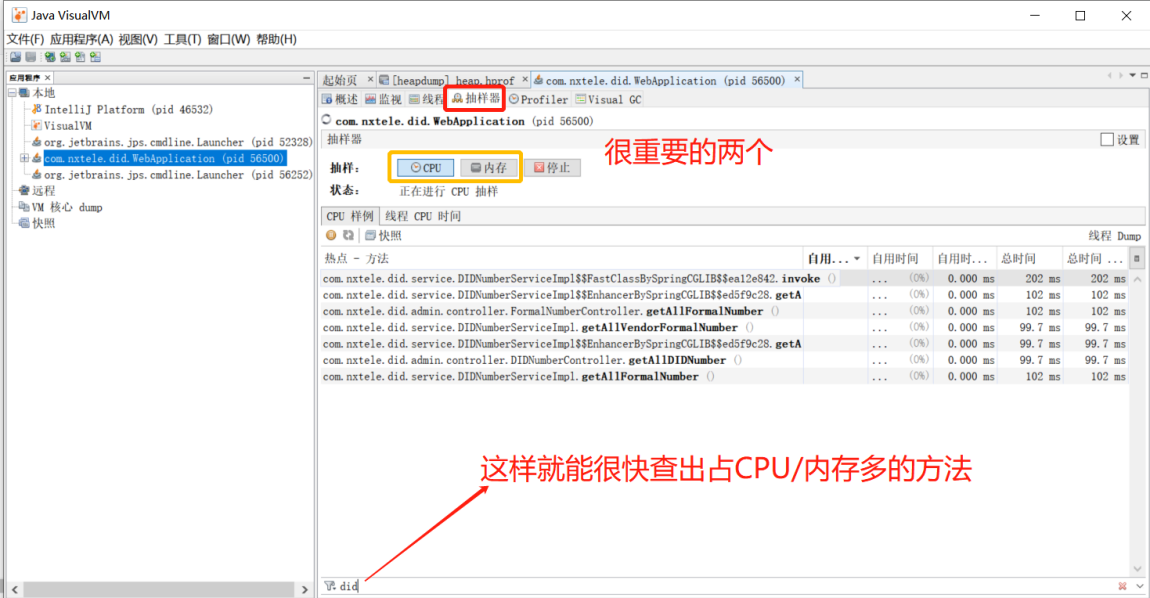
说明：本地起了一个springboot项目



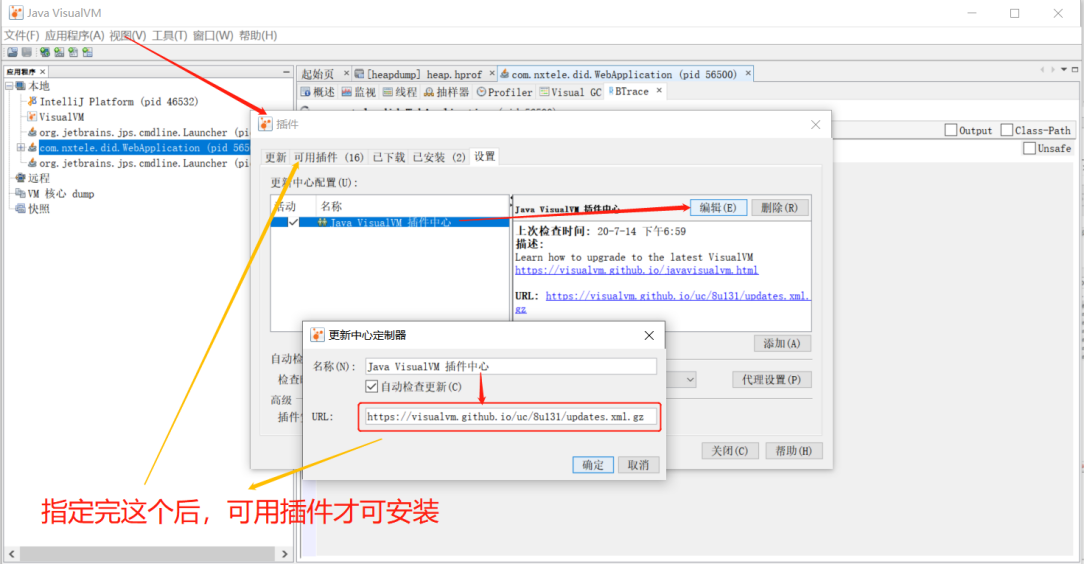
也可以导入内存映像文件进行分析：





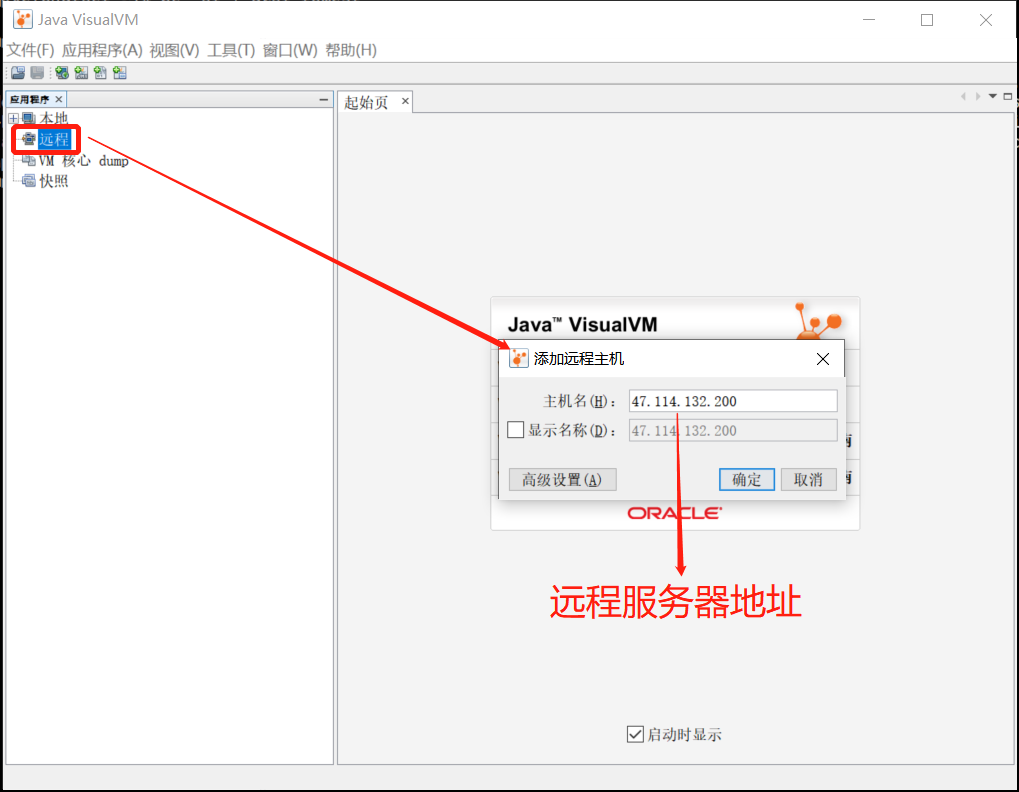


插件的安装：

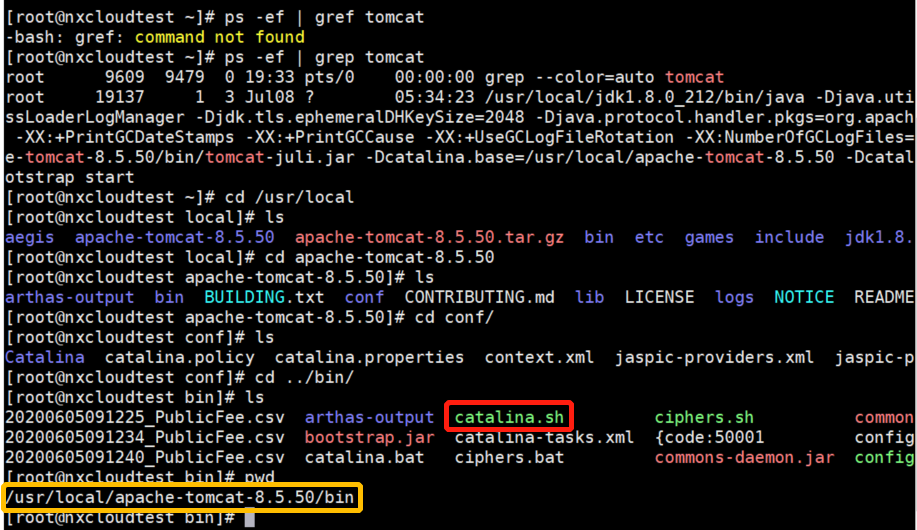


##### 监控远程Tomcat

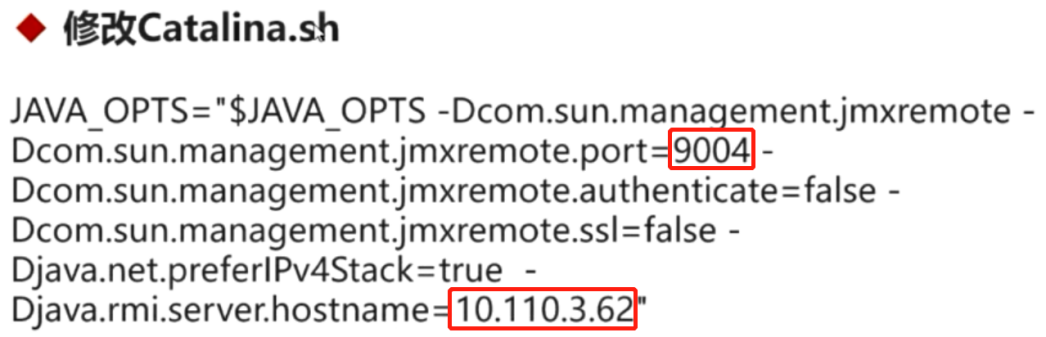
1. 、添加远程主机



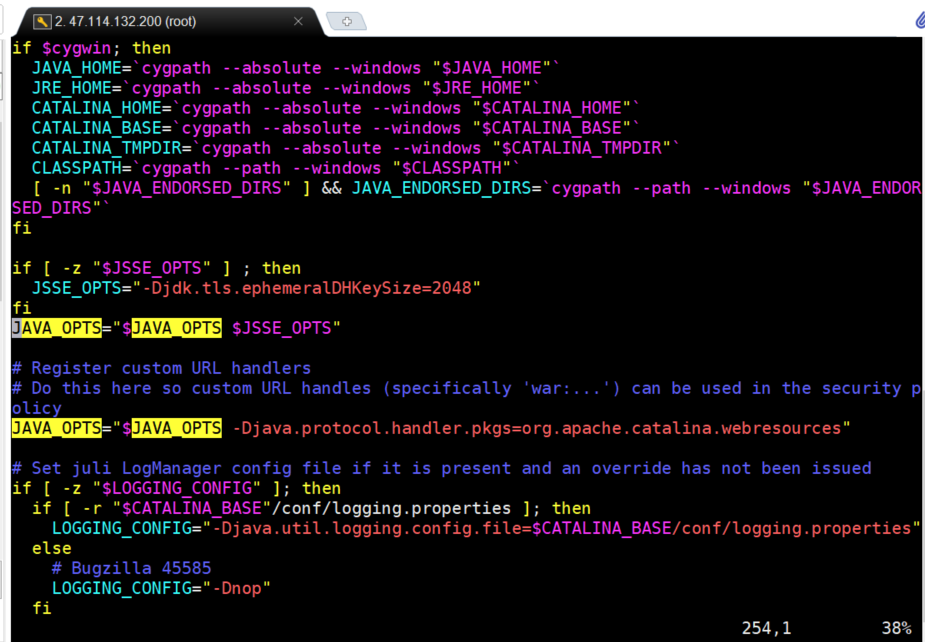
1. 、添加JMX连接
   1. 需要修改远程Tomcat的Catalina.sh



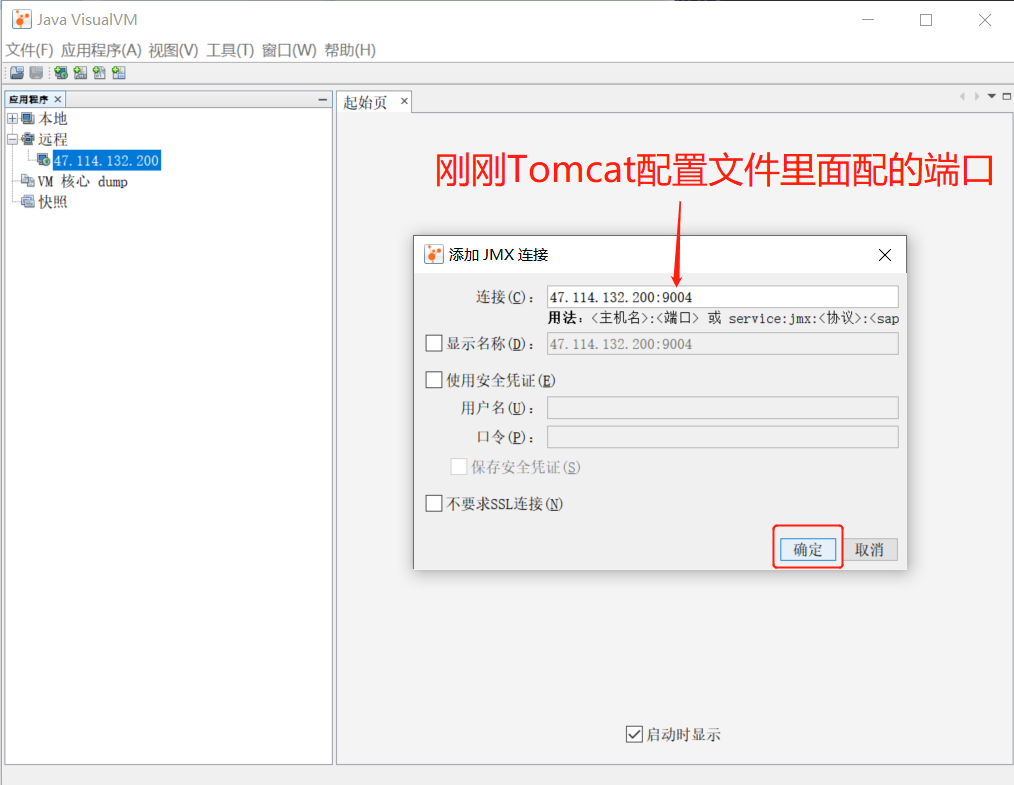
* 1. 修改内容为：



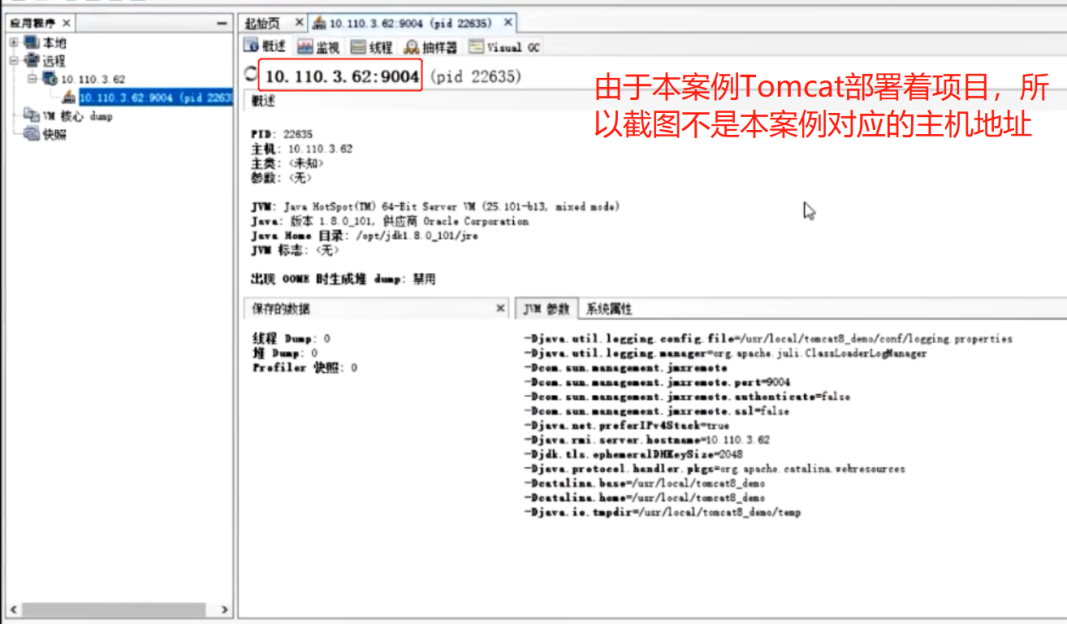
* 1. 利用vim 命令打开catalina.sh文件



* 1. 补充：文件内搜索命令” :/JAVA\_OPTS ”,然后按键盘“N”键，找下一个
  2. 保存，然后重新启动Tomcat
  3. 创建JMX连接

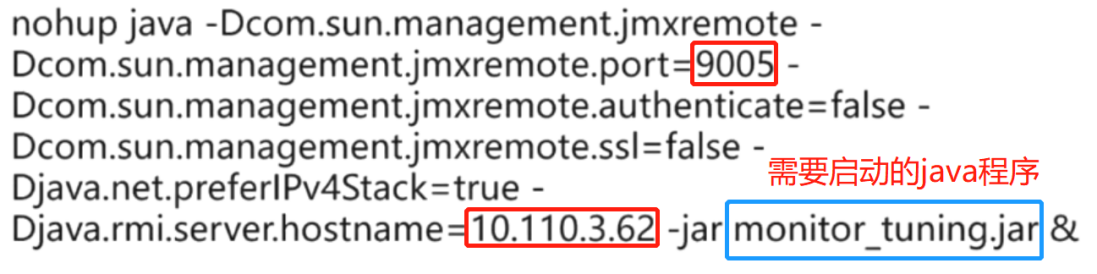


* 1. 连接后的效果



##### 监控远程普通的JAVA进程

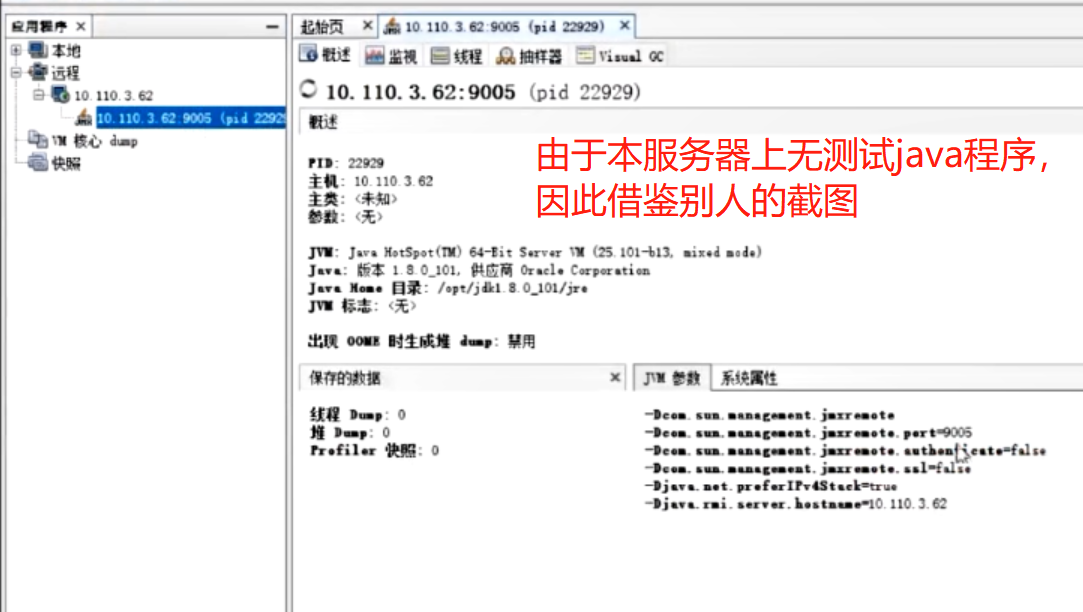
1. 、修改启动参数



1. 创建JMX连接

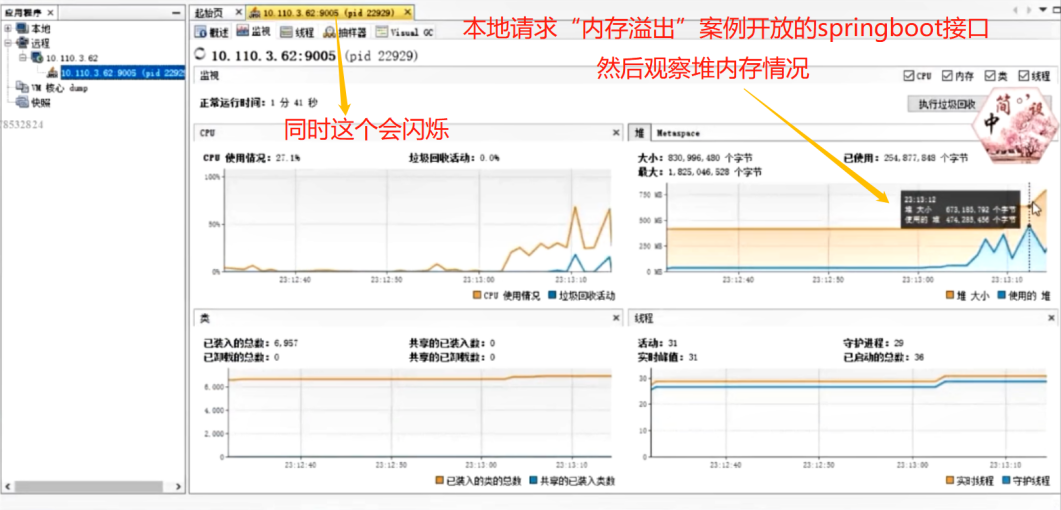


1. 启动后的效果



备注：第一章中的内存溢出案例程序

1. 访问内存溢出接口



1. 执行一下堆Dump，然后分析导出的内存映像文件(如第一章分析)

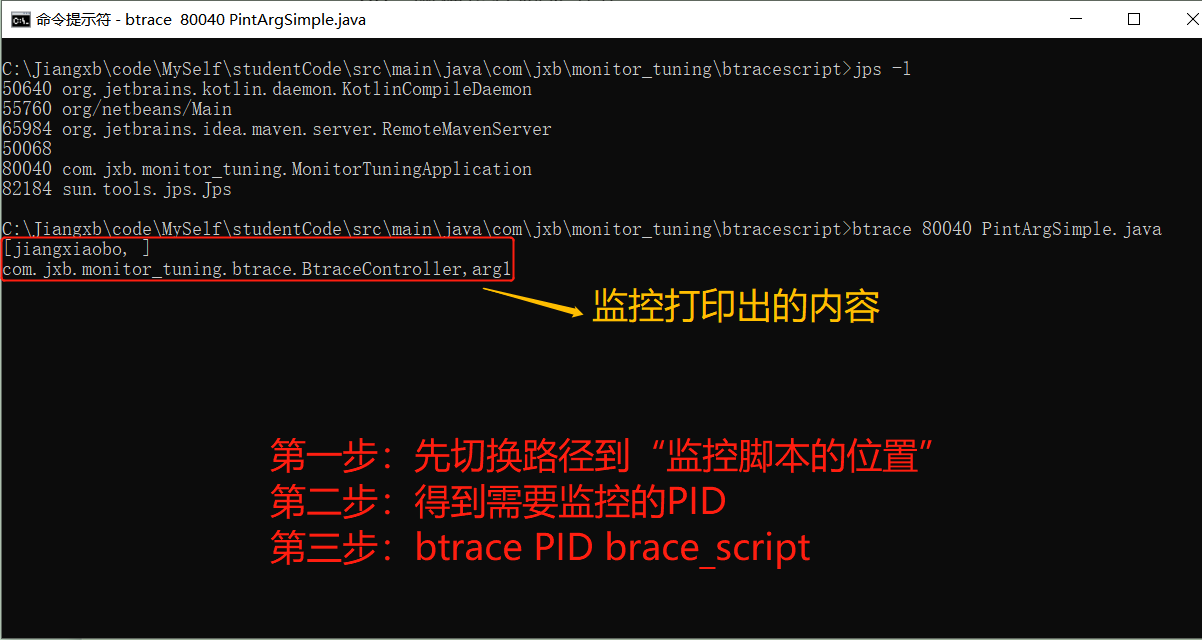
#### 基于Btrace的监控调试

##### 、[Btrace](https://github.com/btraceio/btrace)安装入门

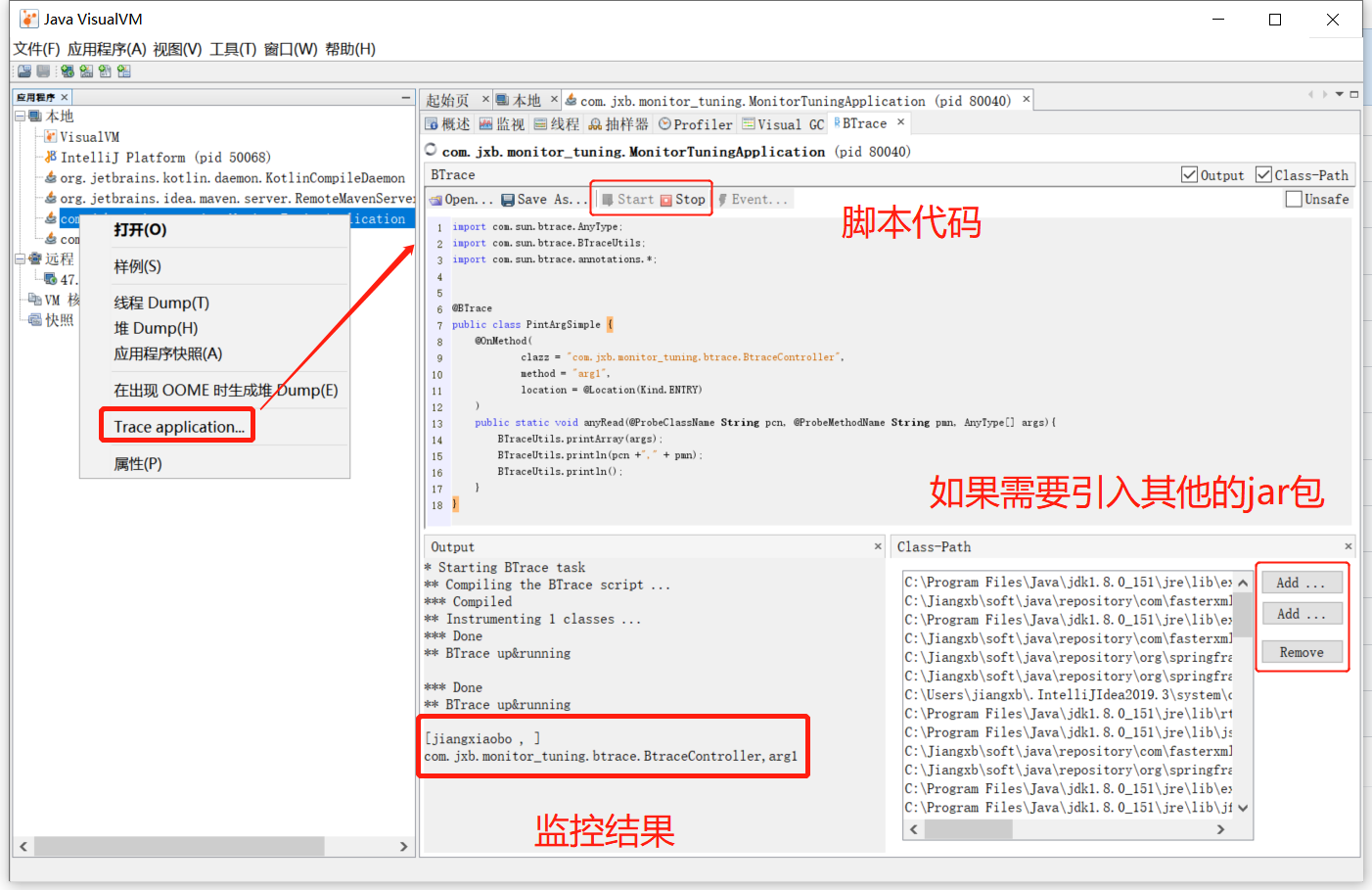
* 1. Btrace可以动态地向目标应用程序的字节码注入追踪代码【在应用程序不需要重启的情况下】
  2. 一些底层的技术
     1. JavaComplierApi、JVMTI、Agent、Instrumentation+ASM
  3. [Btrace安装](https://blog.csdn.net/xingbear/article/details/78091851)
     1. 新建环境变量 BTRACE\_HOME
     2. 添加Path：%BTRACE\_HOME%\bin
  4. 两种运行脚本方式
     1. 在JVisuaIVM中添加Btrace插件，添加classpath
     2. 使用命令行 btrace <pid> <trace\_script>

##### 、Btrace使用详解

* 1. 简单案例1 —— 监控的SpringBoot接口
     1. 编写完需要[监控的类](https://github.com/gitjiangxb/studentCode/blob/monitor_tuning/src/main/java/com/jxb/monitor_tuning/btrace/BtraceController.java)，然后编写[监控脚本](https://github.com/gitjiangxb/studentCode/blob/monitor_tuning/src/main/java/com/jxb/monitor_tuning/btracescript/PintArgSimple.java)
     2. 使用方式一：



* + 1. 使用方式二：



说明：需要先在VisuaIVM里面安装插件-BTrace Workbench

* 1. 拦截方法
     1. 普通方法

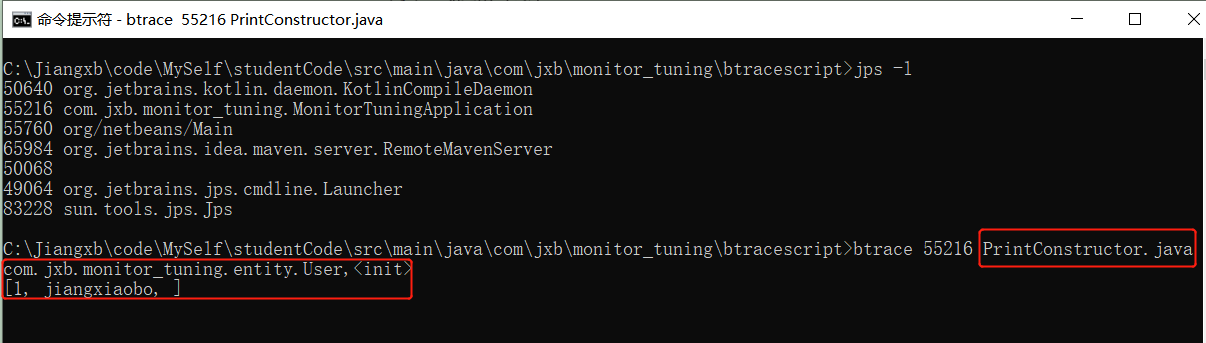
@OnMethod(clazz=””,method=””)

* + 1. 构造函数

@OnMethod(clazz=””,method=”<init>”)

[监控接口类](https://github.com/gitjiangxb/studentCode/blob/monitor_tuning/src/main/java/com/jxb/monitor_tuning/btrace/BtraceController.java" \l "L29-L38) [监控接口类的脚本](https://github.com/gitjiangxb/studentCode/blob/monitor_tuning/src/main/java/com/jxb/monitor_tuning/btracescript/PrintConstructor.java)

监控结果：

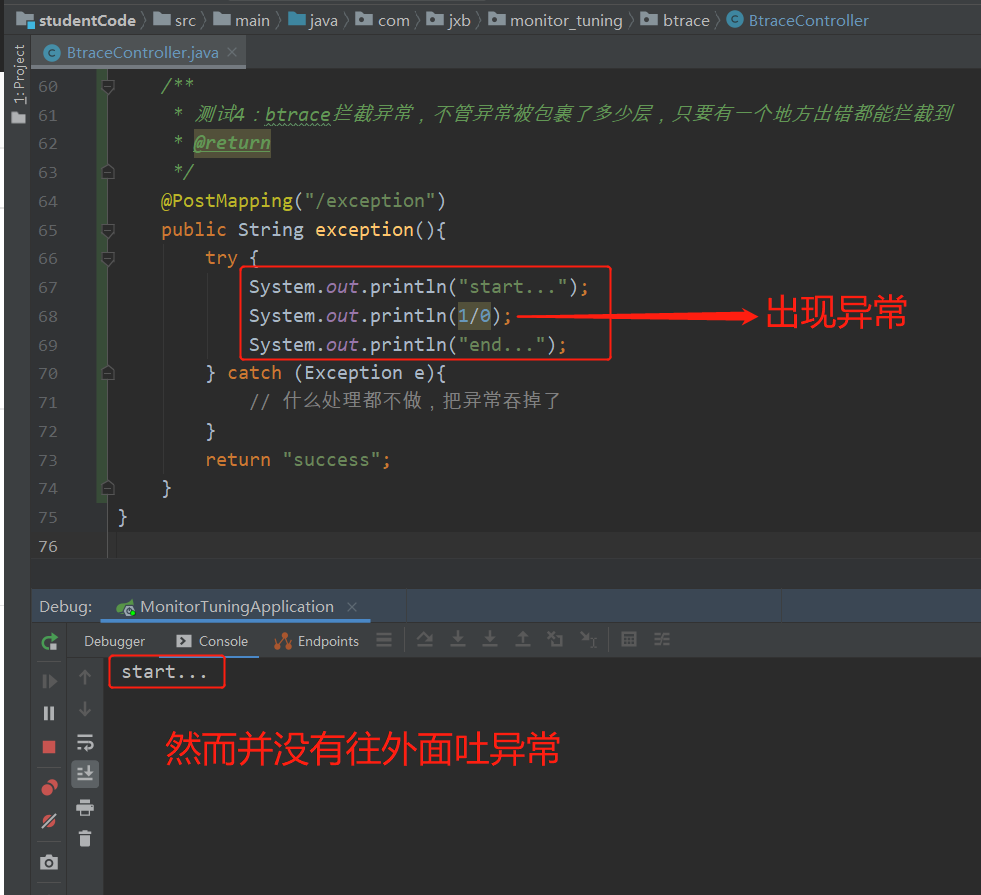


说明：测试接口用postman请求

* + 1. 拦截同名函数，用参数区分

[监控接口类](https://github.com/gitjiangxb/studentCode/blob/monitor_tuning/src/main/java/com/jxb/monitor_tuning/btrace/BtraceController.java" \l "L40-L58) [监控接口类的脚本](https://github.com/gitjiangxb/studentCode/blob/monitor_tuning/src/main/java/com/jxb/monitor_tuning/btracescript/PrintSame.java)

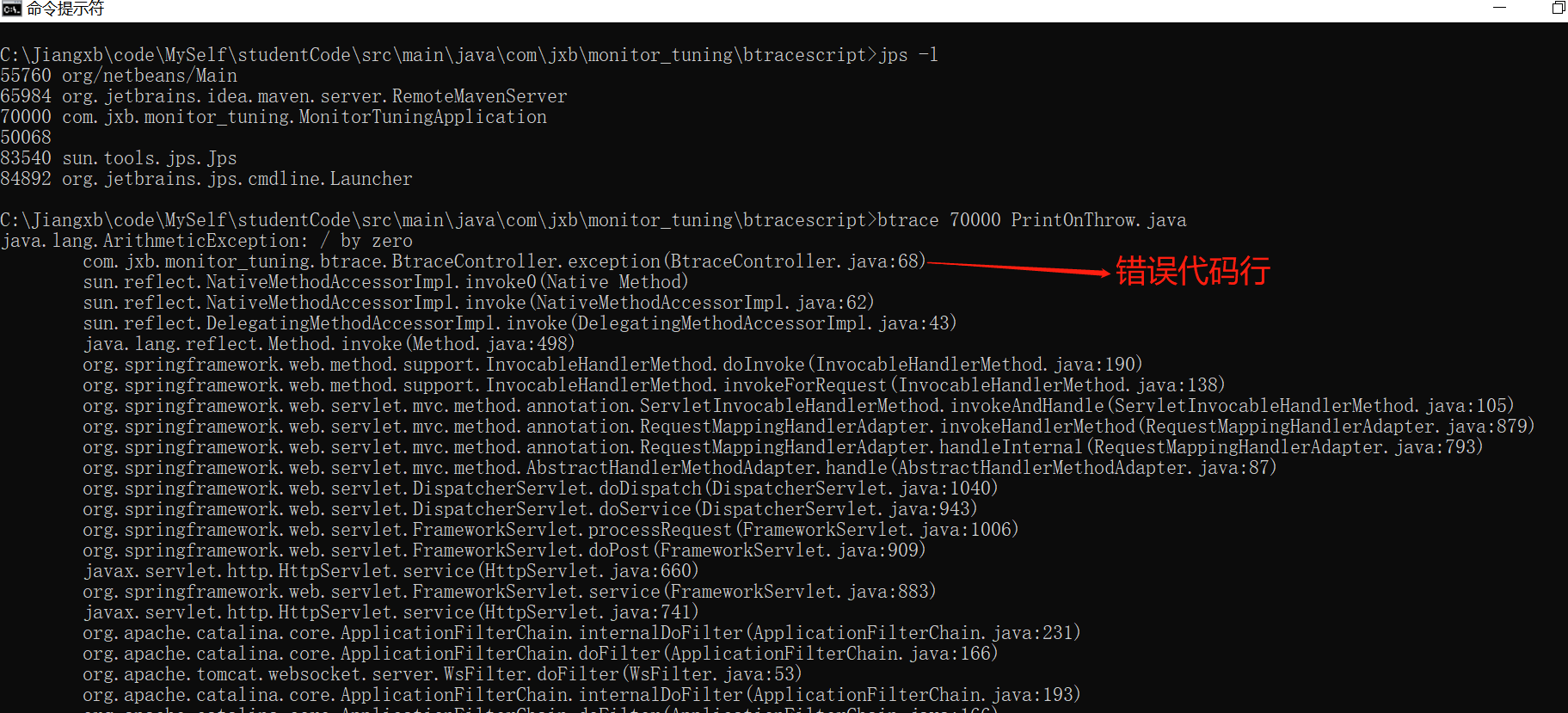
* 1. 拦截时机
     1. Kind.ENTRY：入口，默认值
     2. Kind.RETURN：[返回](https://github.com/gitjiangxb/studentCode/blob/monitor_tuning/src/main/java/com/jxb/monitor_tuning/btracescript/PintArgSimple.java" \l "L21-L30)
     3. Kind.THROW：异常



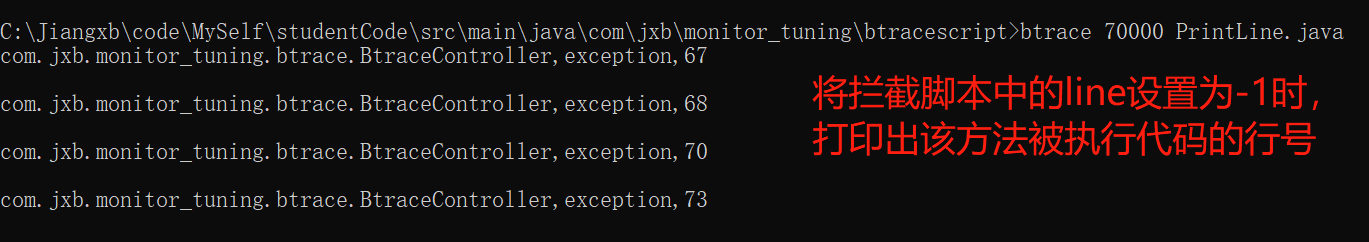
说明：利用postman进行请求

[监控接口类](https://github.com/gitjiangxb/studentCode/blob/monitor_tuning/src/main/java/com/jxb/monitor_tuning/btrace/BtraceController.java" \l "L60-L74) [监控接口类的脚本](https://github.com/gitjiangxb/studentCode/blob/monitor_tuning/src/main/java/com/jxb/monitor_tuning/btracescript/PrintOnThrow.java)

监控结果：



* + 1. Kind.Line：[行](https://github.com/gitjiangxb/studentCode/blob/monitor_tuning/src/main/java/com/jxb/monitor_tuning/btracescript/PrintLine.java)



说明：如果将line指定到明确的行号，若执行了，则打印出来

* 1. 拦截this、参数、返回值
     1. this：@Self
     2. 入参：可以用AnyType,也可以用真实类型，同名方法的用真实类型
     3. 返回值：@Return
     4. 获取对象的值
        1. 简单类型：直接获取
        2. 复杂类型：反射，类名+属性名
  2. 其他

#### Tomcat性能监控与调优

#### Nginx性能监控与调优

#### JVM层GC调优

#### JVM字节码与Java代码层调优