调试排错 - Java问题排查：工具单

Java 调试入门工具

jps

> jps是jdk提供的一个查看当前java进程的小工具， 可以看做是JavaVirtual Machine Process Status Tool的缩写。

jps常用命令

```
jps # 显示进程的ID 和 类的名称
jps -l # 输出输出完全的包名，应用主类名，jar的完全路径名
jps -v # 输出jvm参数
jps -q # 显示java进程号
jps -m # main 方法
jps -l xxx.xxx.xx.xx # 远程查看
```

jps参数

```
-q：仅输出VM标识符，不包括classname,jar name,arguments in main method
-m：输出main method的参数
-l：输出完全的包名，应用主类名，jar的完全路径名
-v：输出jvm参数
-V：输出通过flag文件传递到JVM中的参数(.hotspotrc文件或-XX:Flags=所指定的文件
-Joption：传递参数到vm,例如:-J-Xms512m
```

jps原理

> java程序在启动以后，会在java.io.tmpdir指定的目录下，就是临时文件夹里，生成一个类似于hsperfdata_User的文件夹，这个文件夹里（在Linux中为/tmp/hsperfdata_{userName}/），有几个文件，名字就是java进程的pid，因此列出当前运行的java进程，只是把这个目录里的文件名列一下而已。 至于系统的参数什么，就可以解析这几个文件获得。

更多请参考 jps - Java Virtual Machine Process Status Tool (opens new window)

jstack

> jstack是jdk自带的线程堆栈分析工具，使用该命令可以查看或导出 Java 应用程序中线程堆栈信息。

jstack常用命令:

```
# 基本
jstack 2815

# java和native c/c++框架的所有栈信息
jstack -m 2815

# 额外的锁信息列表，查看是否死锁
jstack -l 2815
```

jstack参数：

```
-l 长列表. 打印关于锁的附加信息,例如属于java.util.concurrent 的 ownable synchronizers列表.

-F 当’jstack [-l] pid’没有相应的时候强制打印栈信息

-m 打印java和native c/c++框架的所有栈信息.

-h | -help 打印帮助信息
```

更多请参考: jvm 性能调优工具之 jstack (opens new window)

jinfo

> jinfo 是 JDK 自带的命令，可以用来查看正在运行的 java 应用程序的扩展参数，包括Java System属性和JVM命令行参数；也可以动态的修改正在运行的 JVM 一些参数。当系统崩溃时，jinfo可以从core文件里面知道崩溃的Java应用程序的配置信息

jinfo常用命令:

```
# 输出当前 jvm 进程的全部参数和系统属性
jinfo 2815

# 输出所有的参数
jinfo -flags 2815

# 查看指定的 jvm 参数的值
jinfo -flag PrintGC 2815

# 开启/关闭指定的JVM参数
jinfo -flag +PrintGC 2815

# 设置flag的参数
jinfo -flag name=value 2815

# 输出当前 jvm 进行的全部的系统属性
jinfo -sysprops 2815
```

jinfo参数：

```
no option 输出全部的参数和系统属性
-flag name 输出对应名称的参数
-flag [+|-]name 开启或者关闭对应名称的参数
-flag name=value 设定对应名称的参数
-flags 输出全部的参数
-sysprops 输出系统属性
```

更多请参考：jvm 性能调优工具之 jinfo (opens new window)

jmap

> 命令jmap是一个多功能的命令。它可以生成 java 程序的 dump 文件， 也可以查看堆内对象示例的统计信息、查看 ClassLoader 的信息以及 finalizer 队列。

两个用途

```
# 查看堆的情况
jmap -heap 2815

# dump
jmap -dump:live,format=b,file=/tmp/heap2.bin 2815
jmap -dump:format=b,file=/tmp/heap3.bin 2815

# 查看堆的占用
jmap -histo 2815 | head -10
```

jmap参数

```
no option： 查看进程的内存映像信息,类似 Solaris pmap 命令。
heap： 显示Java堆详细信息
histo[:live]： 显示堆中对象的统计信息
clstats：打印类加载器信息
finalizerinfo： 显示在F-Queue队列等待Finalizer线程执行finalizer方法的对象
dump:<dump-options>：生成堆转储快照
F： 当-dump没有响应时，使用-dump或者-histo参数. 在这个模式下,live子参数无效.
help：打印帮助信息
J<flag>：指定传递给运行jmap的JVM的参数
```

更多请参考：jvm 性能调优工具之 jmap (opens new window) 和 jmap - Memory Map (opens new window)

jstat

jstat参数众多，但是使用一个就够了

```
jstat -gcutil 2815 1000
```

jdb

jdb可以用来预发debug,假设你预发的java_home是/opt/java/，远程调试端口是8000.那么

```
jdb -attach 8000
```

出现以上代表jdb启动成功。后续可以进行设置断点进行调试。

具体参数可见oracle官方说明jdb - The Java Debugger (opens new window)

CHLSDB

CHLSDB感觉很多情况下可以看到更好玩的东西，不详细叙述了。 查询资料听说jstack和jmap等工具就是基于它的。

```
java -classpath /opt/taobao/java/lib/sa-jdi.jar sun.jvm.hotspot.CLHSDB
```

更详细的可见R大此贴 http://rednaxelafx.iteye.com/blog/1847971

Java 调试进阶工具

btrace

首当其冲的要说的是btrace。真是生产环境&预发的排查问题大杀器。 简介什么的就不说了。直接上代码干

- 查看当前谁调用了ArrayList的add方法，同时只打印当前ArrayList的size大于500的线程调用栈

```
@OnMethod(clazz = "java.util.ArrayList", method="add", location = @Location(value = Kind.CALL,
clazz = "/./", method = "/./"))
public static void m(@ProbeClassName String probeClass, @ProbeMethodName String probeMethod,
@TargetInstance Object instance, @TargetMethodOrField String method) {

    if(getInt(field("java.util.ArrayList", "size"), instance) > 479){
        println("check who ArrayList.add method:" + probeClass + "#" + probeMethod  + ",
method:" + method + ", size:" + getInt(field("java.util.ArrayList", "size"), instance));
        jstack();
        println();
        println("============================");
        println();
    }
}
```

- 监控当前服务方法被调用时返回的值以及请求的参数

```
@OnMethod(clazz = "com.taobao.sellerhome.transfer.biz.impl.C2CApplyerServiceImpl", method="nav",
location = @Location(value = Kind.RETURN))
public static void mt(long userId, int current, int relation, String check, String redirectUrl,
@Return AnyType result) {

    println("parameter# userId:" + userId + ", current:" + current + ", relation:" + relation +
", check:" + check + ", redirectUrl:" + redirectUrl + ", result:" + result);
}
```

btrace 具体可以参考这里：https://github.com/btraceio/btrace

注意:

- 经过观察，1.3.9的release输出不稳定，要多触发几次才能看到正确的结果
- 正则表达式匹配trace类时范围一定要控制，否则极有可能出现跑满CPU导致应用卡死的情况
- 由于是字节码注入的原理，想要应用恢复到正常情况，需要重启应用。

Greys

Greys是@杜琨的大作吧。说几个挺棒的功能(部分功能和btrace重合):

- `sc -df xxx`: 输出当前类的详情,包括源码位置和classloader结构
- `trace class method`: 打印出当前方法调用的耗时情况，细分到每个方法, 对排查方法性能时很有帮助。

Arthas

> Arthas是基于Greys。

具体请参考：调试排错 - Java应用在线调试Arthas

javOSize

就说一个功能:

- `classes`：通过修改了字节码，改变了类的内容，即时生效。 所以可以做到快速的在某个地方打个日志看看输出，缺点是对代码的侵入性太大。但是如果自己知道自己在干嘛，的确是不错的玩意儿。

其他功能Greys和btrace都能很轻易做的到，不说了。

更多请参考：官网 (opens new window)

JProfiler

之前判断许多问题要通过JProfiler，但是现在Greys和btrace基本都能搞定了。再加上出问题的基本上都是生产环境(网络隔离)，所以基本不怎么使用了，但是还是要标记一下。

更多请参考：官网 (opens new window)

其它工具

dmesg

如果发现自己的java进程悄无声息的消失了，几乎没有留下任何线索，那么dmesg一发，很有可能有你想要的。

sudo dmesg|grep -i kill|less 去找关键字oom_killer。找到的结果类似如下:

```
[6710782.021013] java invoked oom-killer: gfp_mask=0xd0, order=0, oom_adj=0, oom_scoe_adj=0
[6710782.070639] [<ffffffff81118898>] ? oom_kill_process+0x68/0x140
[6710782.257588] Task in /LXC011175068174 killed as a result of limit of /LXC011175068174
[6710784.698347] Memory cgroup out of memory: Kill process 215701 (java) score 854 or sacrifice
child
[6710784.707978] Killed process 215701, UID 679, (java) total-vm:11017300kB, anon-rss:7152432kB,
file-rss:1232kB
```

以上表明，对应的java进程被系统的OOM Killer给干掉了，得分为854. 解释一下OOM killer（Out-Of-Memory killer），该机制会监控机器的内存资源消耗。当机器内存耗尽前，该机制会扫描所有的进程（按照一定规则计算，内存占用，时间等），挑选出得分最高的进程，然后杀死，从而保护机器。

dmesg日志时间转换公式: log实际时间=格林威治1970-01-01+(当前时间秒数-系统启动至今的秒数+dmesg打印的log时间)秒数:

date -d "1970-01-01 UTC echo "$(date +%s)-$(cat /proc/uptime|cut -f 1 -d' ')+12288812.926194"|bc seconds"
剩下的，就是看看为什么内存这么大，触发了OOM-Killer了。