# 1.阿里

### Linux命令类

#### tail

最常用的tail -f

tail -300f shopbase.log #倒数300行并进入实时监听文件写入模式

#### grep

grep forest f.txt     #文件查找  
grep forest f.txt cpf.txt #多文件查找  
grep 'log' /home/admin -r -n #目录下查找所有符合关键字的文件  
cat f.txt | grep -i shopbase      
grep 'shopbase' /home/admin -r -n --include \*.{vm,java} #指定文件后缀  
grep 'shopbase' /home/admin -r -n --exclude \*.{vm,java} #反匹配  
seq 10 | grep 5 -A 3    #上匹配  
seq 10 | grep 5 -B 3    #下匹配  
seq 10 | grep 5 -C 3    #上下匹配，平时用这个就妥了  
cat f.txt | grep -c 'SHOPBASE'

#### awk

1 基础命令

awk '{print $4,$6}' f.txt  
awk '{print NR,$0}' f.txt cpf.txt      
awk '{print FNR,$0}' f.txt cpf.txt  
awk '{print FNR,FILENAME,$0}' f.txt cpf.txt  
awk '{print FILENAME,"NR="NR,"FNR="FNR,"$"NF"="$NF}' f.txt cpf.txt  
echo 1:2:3:4 | awk -F: '{print $1,$2,$3,$4}'

2 匹配

awk '/ldb/ {print}' f.txt   #匹配ldb  
awk '!/ldb/ {print}' f.txt  #不匹配ldb  
awk '/ldb/ && /LISTEN/ {print}' f.txt   #匹配ldb和LISTEN  
awk '$5 ~ /ldb/ {print}' f.txt #第五列匹配ldb

3 内建变量

NR:NR表示从awk开始执行后，按照记录分隔符读取的数据次数，默认的记录分隔符为换行符，因此默认的就是读取的数据行数，NR可以理解为Number of Record的缩写。

FNR:在awk处理多个输入文件的时候，在处理完第一个文件后，NR并不会从1开始，而是继续累加，因此就出现了FNR，每当处理一个新文件的时候，FNR就从1开始计数，FNR可以理解为File Number of Record。

NF: NF表示目前的记录被分割的字段的数目，NF可以理解为Number of Field。

#### find

sudo -u admin find /home/admin /tmp /usr -name \\*.log(多个目录去找)  
find . -iname \\*.txt(大小写都匹配)  
find . -type d(当前目录下的所有子目录)  
find /usr -type l(当前目录下所有的符号链接)  
find /usr -type l -name "z\*" -ls(符号链接的详细信息 eg:inode,目录)  
find /home/admin -size +250000k(超过250000k的文件，当然+改成-就是小于了)  
find /home/admin f -perm 777 -exec ls -l {} \; (按照权限查询文件)  
find /home/admin -atime -1  1天内访问过的文件  
find /home/admin -ctime -1  1天内状态改变过的文件      
find /home/admin -mtime -1  1天内修改过的文件  
find /home/admin -amin -1  1分钟内访问过的文件  
find /home/admin -cmin -1  1分钟内状态改变过的文件      
find /home/admin -mmin -1  1分钟内修改过的文件

#### pgm

批量查询vm-shopbase满足条件的日志

pgm -A -f vm-shopbase 'cat /home/admin/shopbase/logs/shopbase.log.2017-01-17|grep 2069861630'

#### tsar

tsar是咱公司自己的采集工具。很好用, 将历史收集到的数据持久化在磁盘上，所以我们快速来查询历史的系统数据。当然实时的应用情况也是可以查询的啦。大部分机器上都有安装。

tsar  ###可以查看最近一天的各项指标

tsar --live ###可以查看实时指标，默认五秒一刷

tsar -d 20161218 ###指定查看某天的数据，貌似最多只能看四个月的数据

tsar --mem  
tsar --load  
tsar --cpu  
###当然这个也可以和-d参数配合来查询某天的单个指标的情况

#### top

top除了看一些基本信息之外，剩下的就是配合来查询vm的各种问题了

ps -ef | grep java  
top -H -p pid

获得线程10进制转16进制后jstack去抓看这个线程到底在干啥

#### 其他

netstat -nat|awk  '{print $6}'|sort|uniq -c|sort -rn   
#查看当前连接，注意close\_wait偏高的情况，比如如下

### 排查利器

#### 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

1. 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();  
   }  
}

1. 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);  
}

其他功能集团的一些工具或多或少都有，就不说了。感兴趣的请移步。  
https://github.com/btraceio/btrace

注意:

1. 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
2. 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
3. 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: 相当喜欢这个功能! 很早前可以早JProfiler看到这个功能。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，比如我之前这篇就是使用了trace命令来的:http://www.atatech.org/articles/52947。

其他功能部分和btrace重合，可以选用，感兴趣的请移步。  
http://www.atatech.org/articles/26247

另外相关联的是arthas，他是基于Greys的，感兴趣的再移步http://mw.alibaba-inc.com/products/arthas/docs/middleware-container/arthas.wiki/home.html?spm=a1z9z.8109794.header.32.1lsoMc

#### 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。

可以看看我之前写的一篇javOSize的简介http://www.atatech.org/articles/38546  
官网请移步http://www.javosize.com/

#### 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。  
官网请移步https://www.ej-technologies.com/products/jprofiler/overview.html

### 大杀器

#### eclipseMAT

可作为eclipse的插件，也可作为单独的程序打开。   
详情请移步http://www.eclipse.org/mat/

#### zprofiler

集团内的开发应该是无人不知无人不晓了。简而言之一句话:有了zprofiler还要mat干嘛  
详情请移步zprofiler.alibaba-inc.com

### java三板斧，噢不对，是七把

#### jps

我只用一条命令：

sudo -u admin /opt/taobao/java/bin/jps -mlvV

#### jstack

普通用法:

sudo -u admin /opt/taobao/install/ajdk-8\_1\_1\_fp1-b52/bin/jstack 2815

native+java栈:

sudo -u admin /opt/taobao/install/ajdk-8\_1\_1\_fp1-b52/bin/jstack -m 2815

#### jinfo

可看系统启动的参数，如下

sudo -u admin /opt/taobao/install/ajdk-8\_1\_1\_fp1-b52/bin/jinfo -flags 2815

#### jmap

两个用途

1.查看堆的情况

sudo -u admin /opt/taobao/install/ajdk-8\_1\_1\_fp1-b52/bin/jmap -heap 2815

2.dump

sudo -u admin /opt/taobao/install/ajdk-8\_1\_1\_fp1-b52/bin/jmap -dump:live,format=b,file=/tmp/heap2.bin 2815

或者

sudo -u admin /opt/taobao/install/ajdk-8\_1\_1\_fp1-b52/bin/jmap -dump:format=b,file=/tmp/heap3.bin 2815

3.看看堆都被谁占了? 再配合zprofiler和btrace，排查问题简直是如虎添翼

sudo -u admin /opt/taobao/install/ajdk-8\_1\_1\_fp1-b52/bin/jmap -histo 2815 | head -10

#### jstat

jstat参数众多，但是使用一个就够了

sudo -u admin /opt/taobao/install/ajdk-8\_1\_1\_fp1-b52/bin/jstat -gcutil 2815 1000

#### jdb

时至今日，jdb也是经常使用的。   
jdb可以用来预发debug,假设你预发的java\_home是/opt/taobao/java/，远程调试端口是8000.那么  
sudo -u admin /opt/taobao/java/bin/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http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html

#### 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。

sudo -u admin /opt/taobao/java/bin/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

### plugin of intellij idea

#### key promoter

快捷键一次你记不住，多来几次你总能记住了吧？

#### maven helper

分析maven依赖的好帮手。

### VM options

1. 你的类到底是从哪个文件加载进来的？

-XX:+TraceClassLoading  
结果形如[Loaded java.lang.invoke.MethodHandleImpl$Lazy from D:\programme\jdk\jdk8U74\jre\lib\rt.jar]

1. 应用挂了输出dump文件

-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/home/admin/logs/java.hprof  
集团的vm参数里边基本都有这个选项

### jar包冲突

把这个单独写个大标题不过分吧？每个人或多或少都处理过这种烦人的case。我特么下边这么多方案不信就搞不定你?

#### mvn dependency:tree > ~/dependency.txt

打出所有依赖

#### mvn dependency:tree -Dverbose -Dincludes=groupId:artifactId

只打出指定groupId和artifactId的依赖关系

#### -XX:+TraceClassLoading

vm启动脚本加入。在tomcat启动脚本中可见加载类的详细信息

#### -verbose

vm启动脚本加入。在tomcat启动脚本中可见加载类的详细信息

#### greys:sc

greys的sc命令也能清晰的看到当前类是从哪里加载过来的

#### tomcat-classloader-locate

通过以下url可以获知当前类是从哪里加载的  
curl http://localhost:8006/classloader/locate?class=org.apache.xerces.xs.XSObject

### ALI-TOMCAT带给我们的惊喜(感谢@务观)

列出容器加载的jar列表

curl http://localhost:8006/classloader/jars

列出当前当当前类加载的实际jar包位置，解决类冲突时有用

curl http://localhost:8006/classloader/locate?class=org.apache.xerces.xs.XSObject

### 其他

#### gpref

http://www.atatech.org/articles/33317

#### 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。