Arthas使用文档

**版本记录**

|  |  |  |  |
| --- | --- | --- | --- |
| 版本 | 修订日期 | 修订人员 | 修订纪要 |
| V1.0 | 2019-07-01 | 徐强辉 | 初稿 |

# 工具说明

Arthas 是基于 Greys 进行二次开发的全新在线诊断工具，利用Java6的Instrumentation特性，动态增强你所指定的类，获取你想要到的信息, 采用命令行交互模式，同时提供丰富的 Tab 自动补全功能，让你在定位、分析诊断问题时看每一个操作都看起来是那么的 666

# 工具作用.

* 这个类从哪个 jar 包加载的？为什么会报各种类相关的 Exception？
* 我改的代码为什么没有执行到？难道是我没 commit？分支搞错了？
* 遇到问题无法在预发 debug 一下，难道只能通过加日志再重新预发布吗？
* 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,怎么办?
* 是否有一个全局视角来查看系统的运行状况？
* 有什么办法可以监控到容器和中间件的实时运行状态？

# 工具使用

## 3.1 安装方式

### 3.1.1直接下载

wget https://alibaba.github.io/arthas/arthas-boot.jar

### 3.1.2 Cloud Toolkit插件

<https://alibaba.github.io/arthas/install-detail.html#cloud-toolkitarthas>

### 3.1.3 Docker

<https://alibaba.github.io/arthas/docker.html>

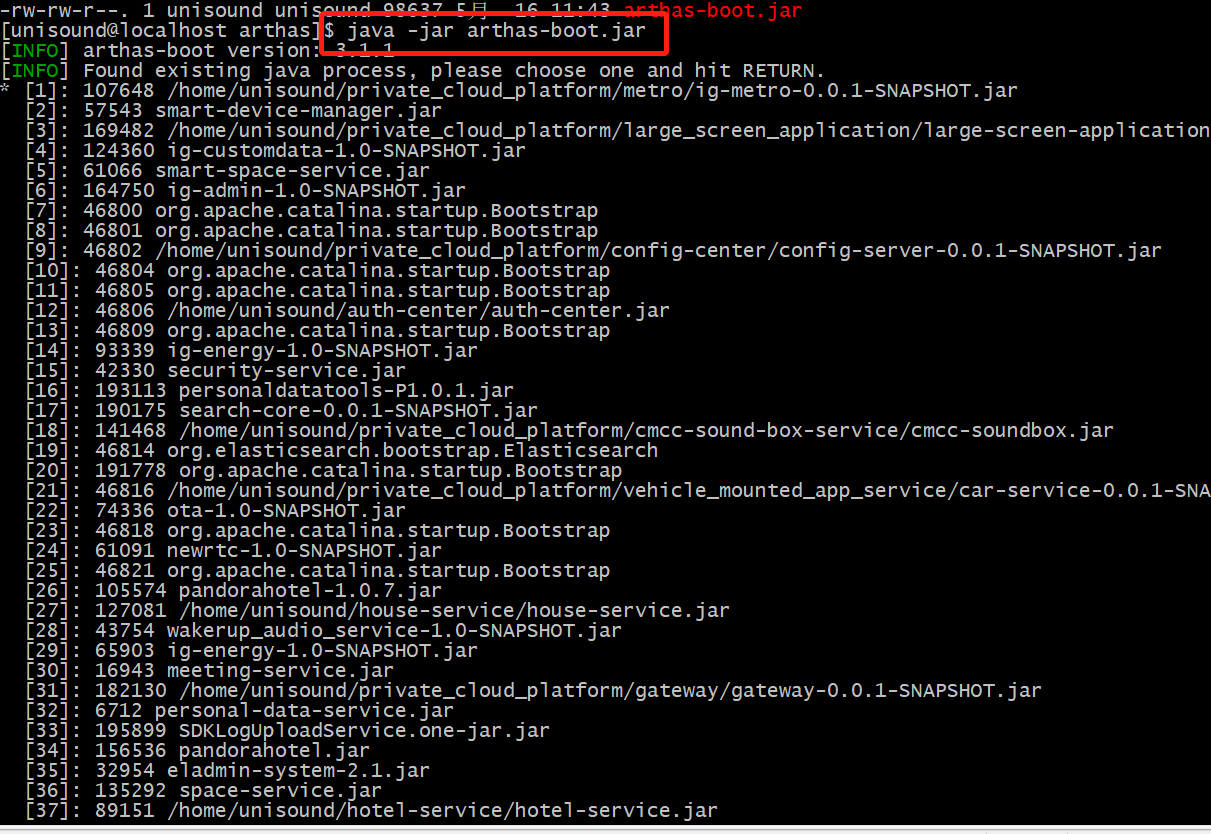
## 3.2 快速入门

### 3.2.1 启动

直接jar包启动.

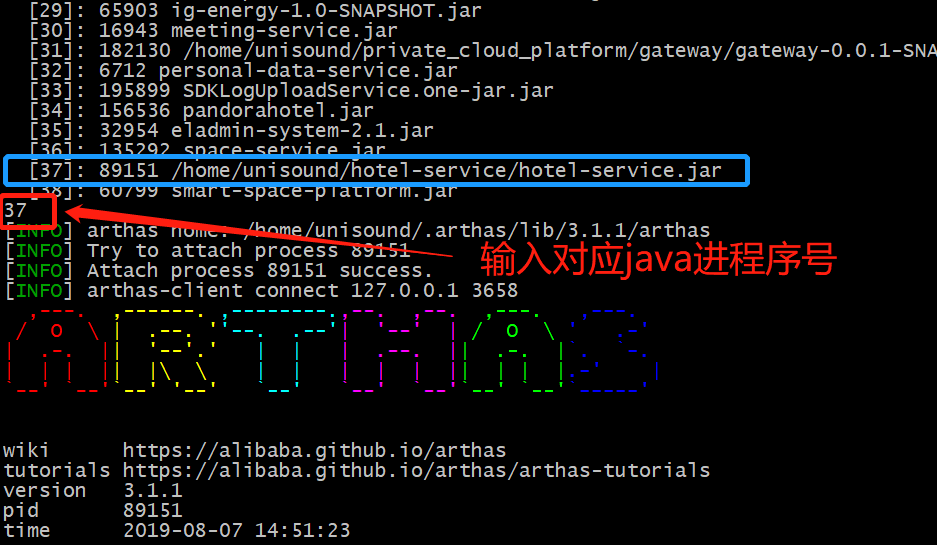
java -jar arthas-boot.jar

启动之后会 列出当前服务器所有jvm实例.



### 3.2.2 选择java进程

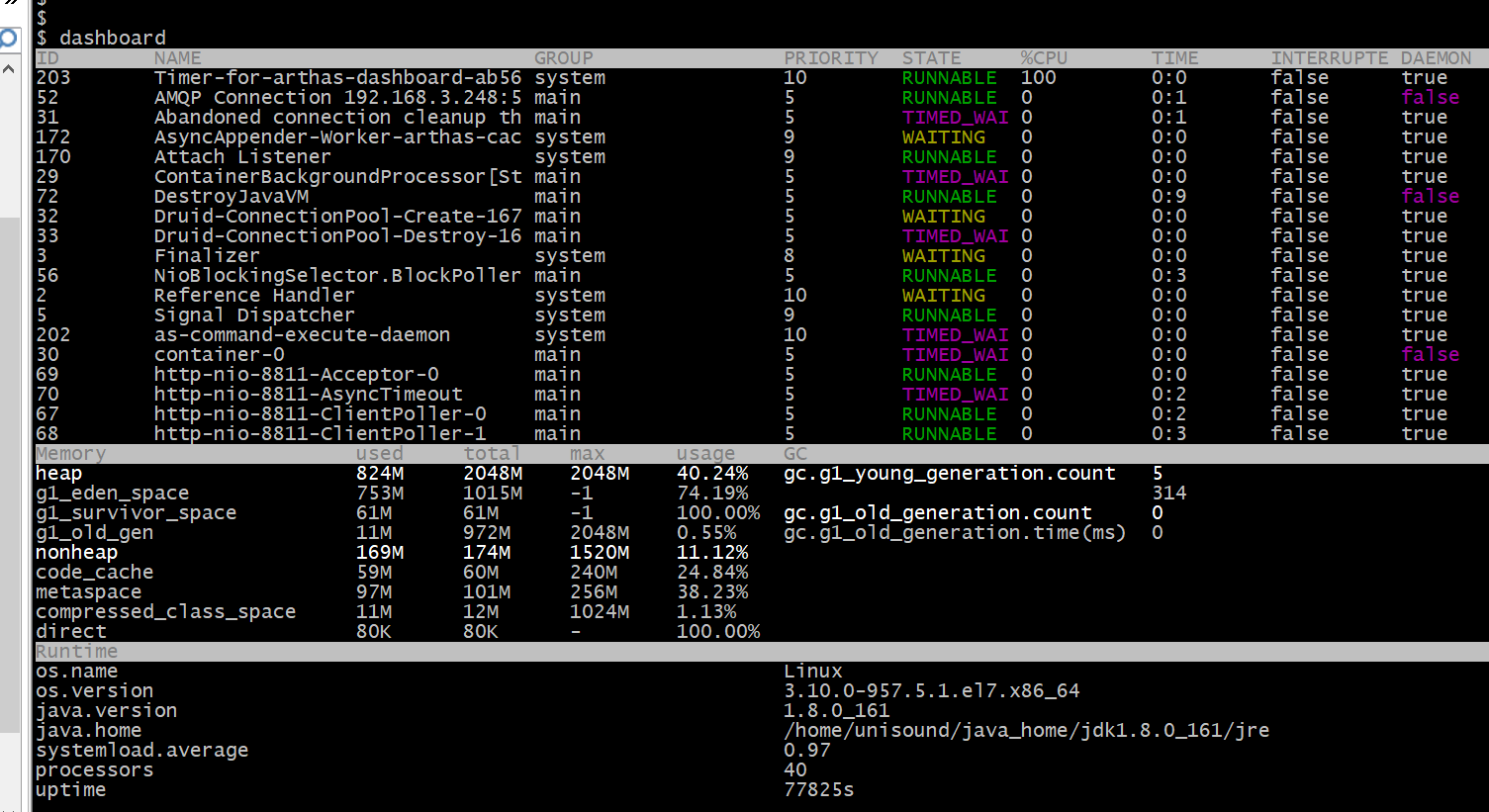
hotel-service 进程是 第37个, 输入37 按回车, Arthas 会attach到目标进程上.



### 3.2.3 基础命令

#### dashboard

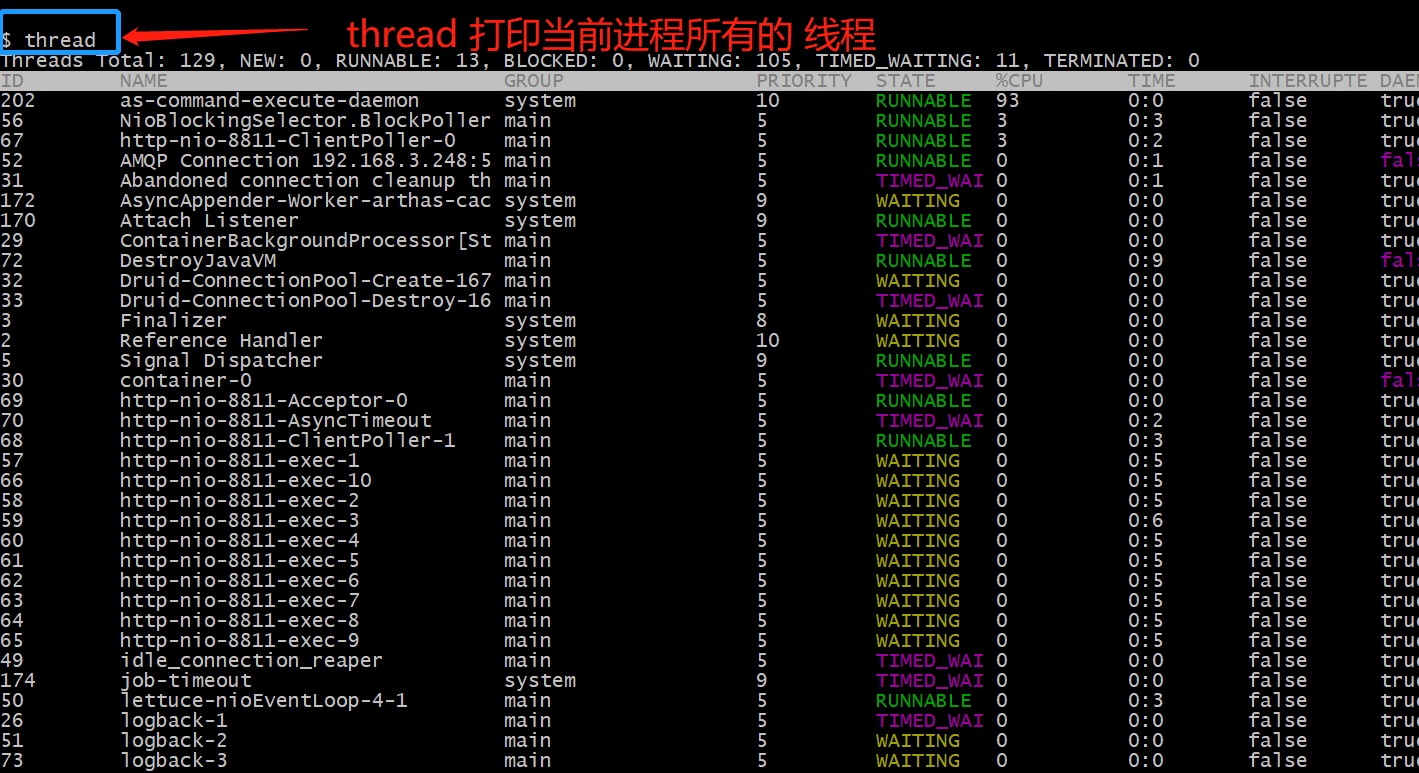
展示当前进程的信息.



#### thread

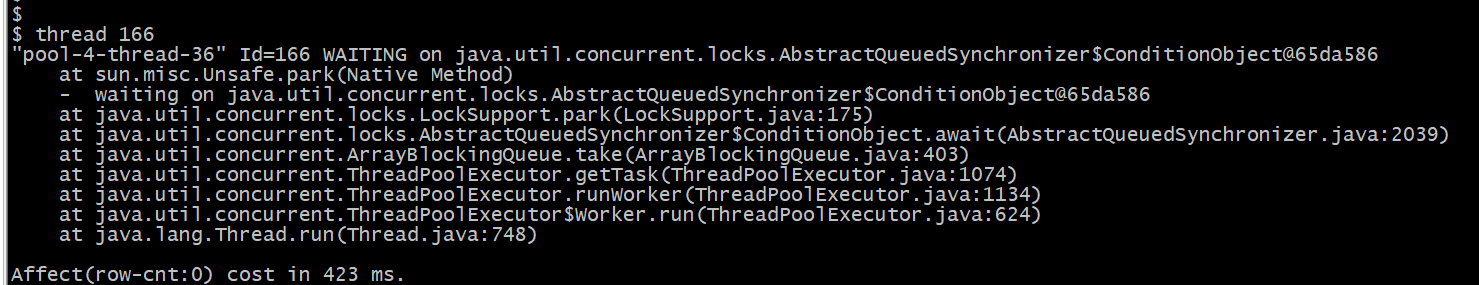
线程相关信息

thread 命令 展示所有线程信息.



thread + 线程序号

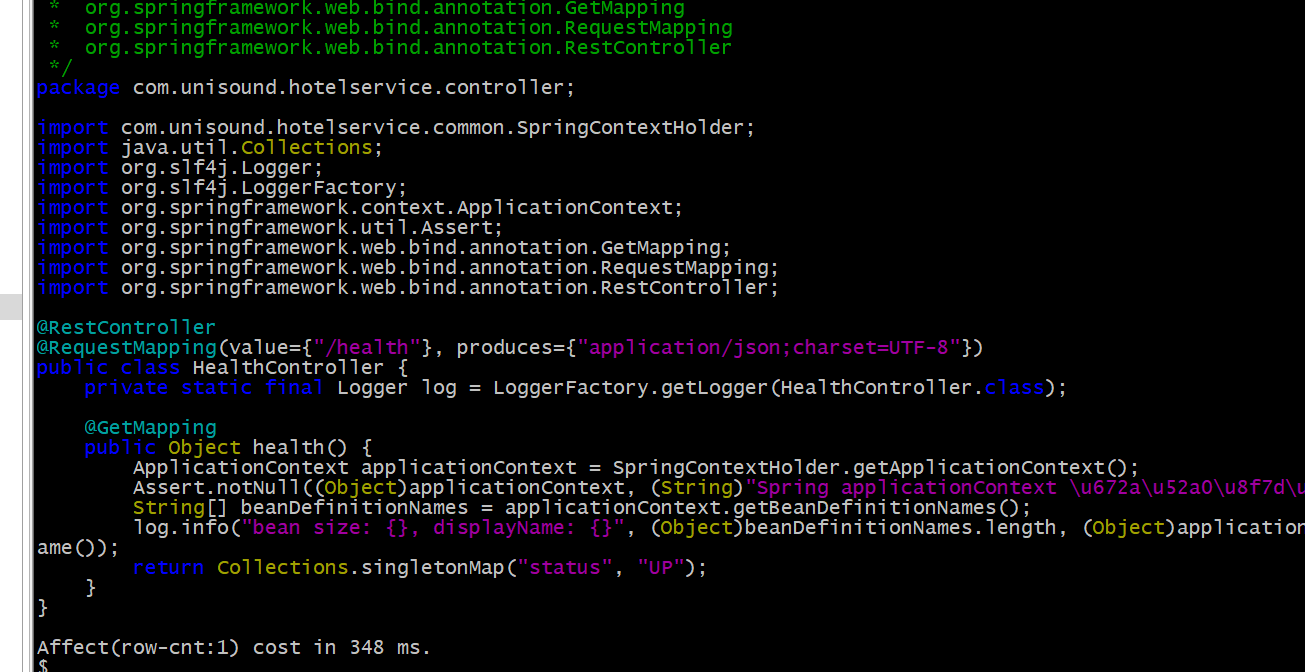
输出该线程的栈.



#### jad

反编译指令

jad com.unisound.hotelservice.controller.HealthController

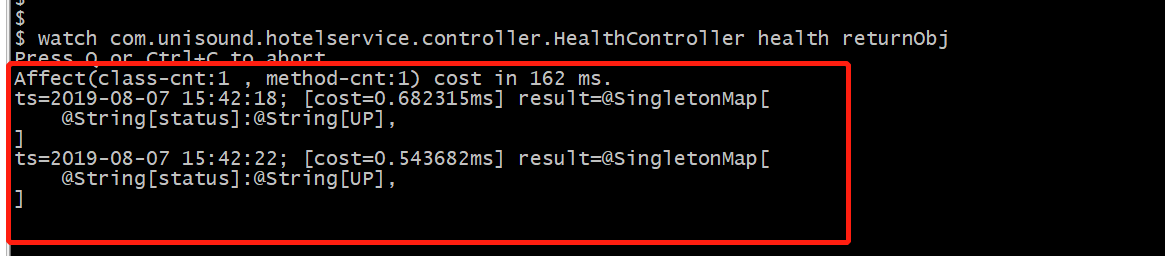


#### watch

监控输出某个类中某方法的 返回值.

watch com.unisound.hotelservice.controller.HealthController health returnObj

监控 HealthController.health() 的返回值



### 3.2.4 退出arthas

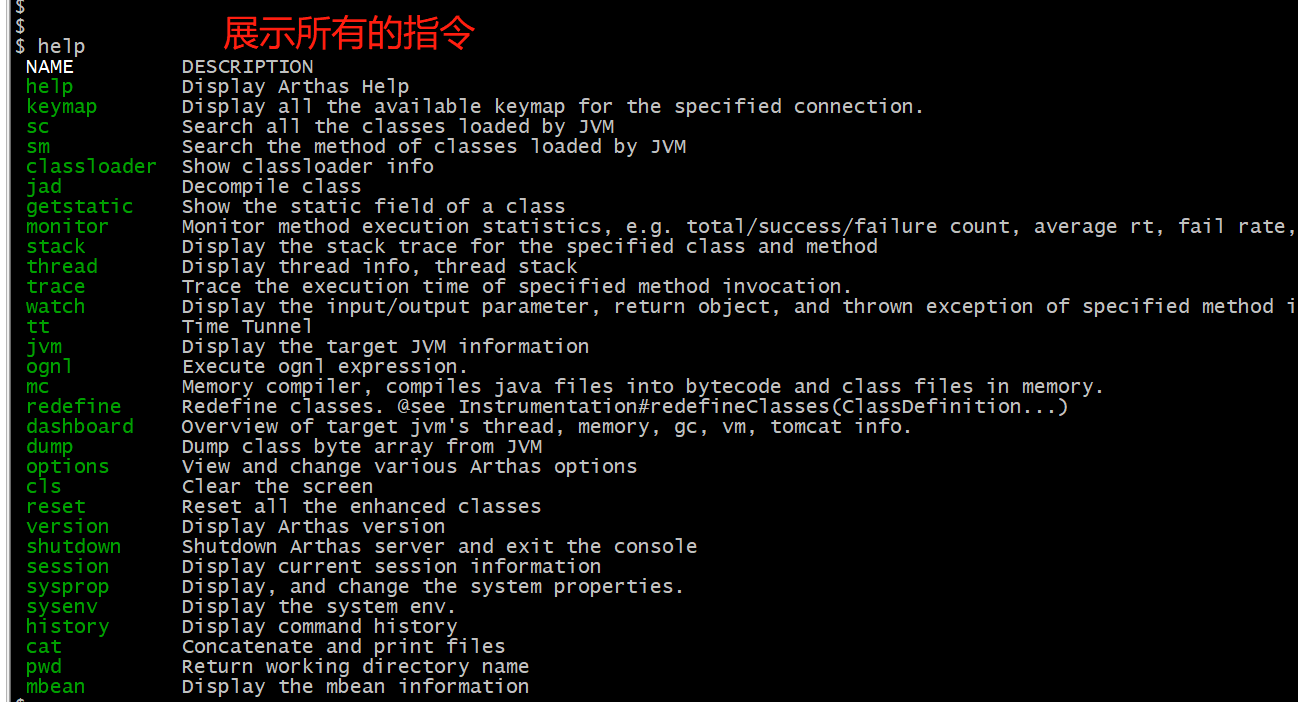
如果只是退出当前的连接，可以用quit或者exit命令。Attach到目标进程上的arthas还会继续运行，端口会保持开放，下次连接时可以直接连接上。

如果想完全退出arthas，可以执行shutdown命令。

# 进阶使用

## 基础命令

### help



### cat

打印文件内容，和linux里的cat命令类似

cat /tmp/a.txt

### pwd

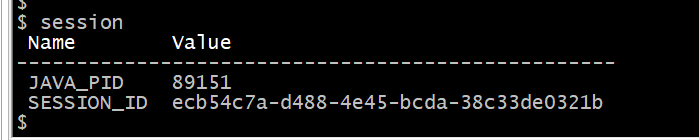
返回当前的工作目录，和linux命令类似

### cls

清空当前屏幕区域, 跟linux ctrl+l 命令类似

### session

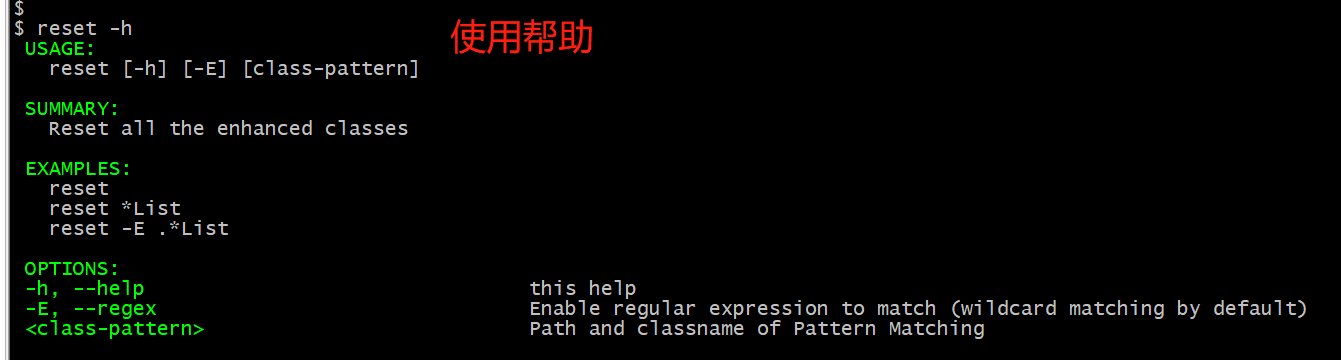
查看当前会话的信息



### reset

重置增强类，将被 Arthas 增强过的类全部还原，Arthas 服务端**shutdown**时会重置所有增强过的类

#### 使用参考

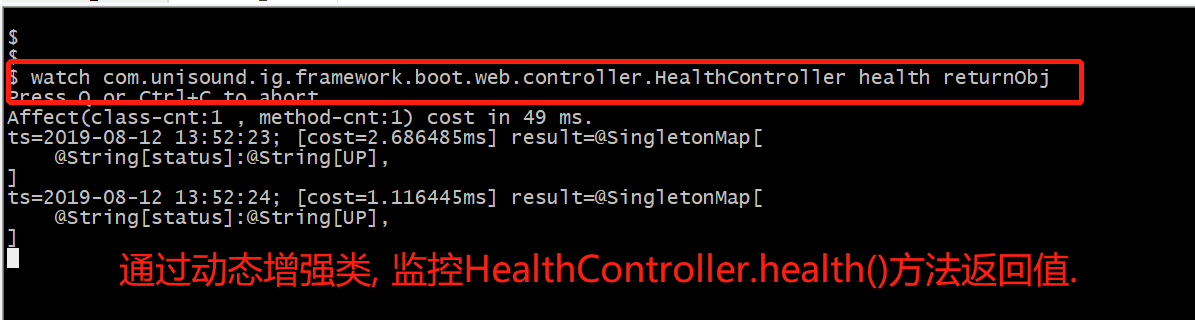


-E 开启正则表达式匹配

#### 示例

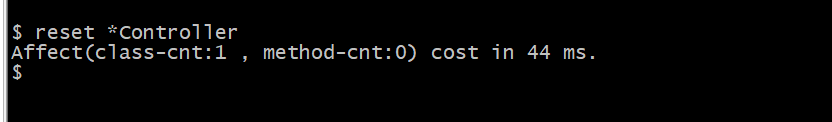
增强两个类

1. watch com.unisound.ig.framework.boot.web.controller.HealthController health returnObj
2. watch com.unisound.ig.house.controller.wechat.MiniProgramApiController addFamilyMember returnObj



重置增强类

1. reset com.unisound.ig.framework.boot.web.controller.HealthController
2. reset \*Controller
3. reset –E .\*Controller

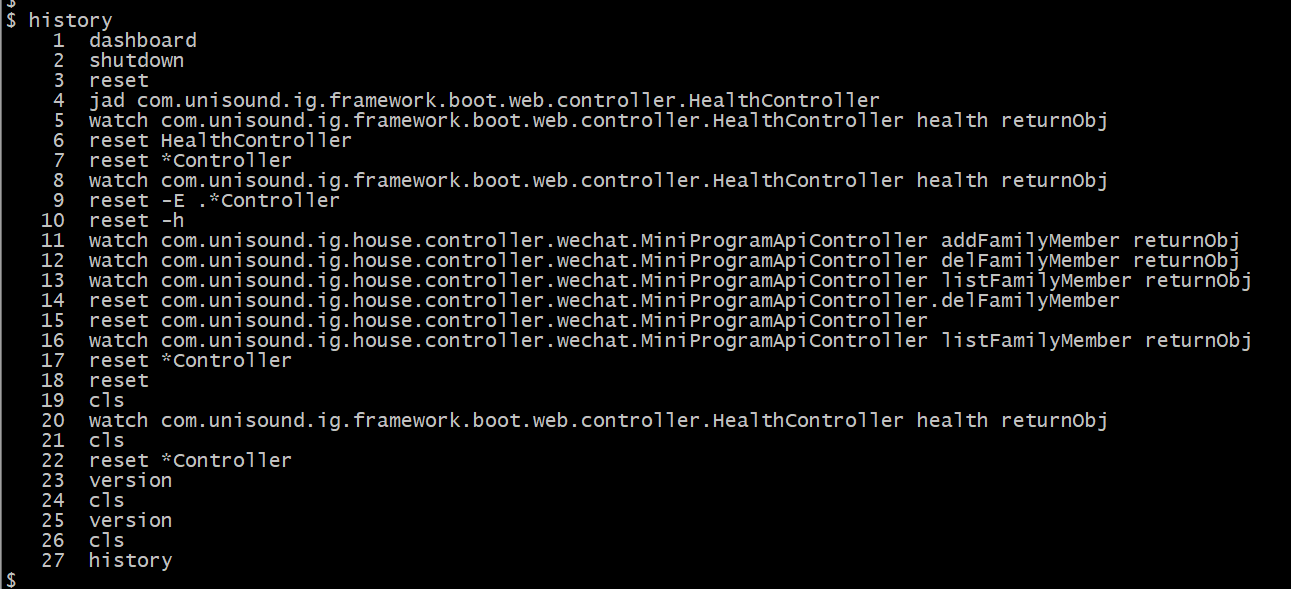


### version

输出当前目标 Java 进程所加载的 Arthas 版本号

### history

打印历史命令



### quit

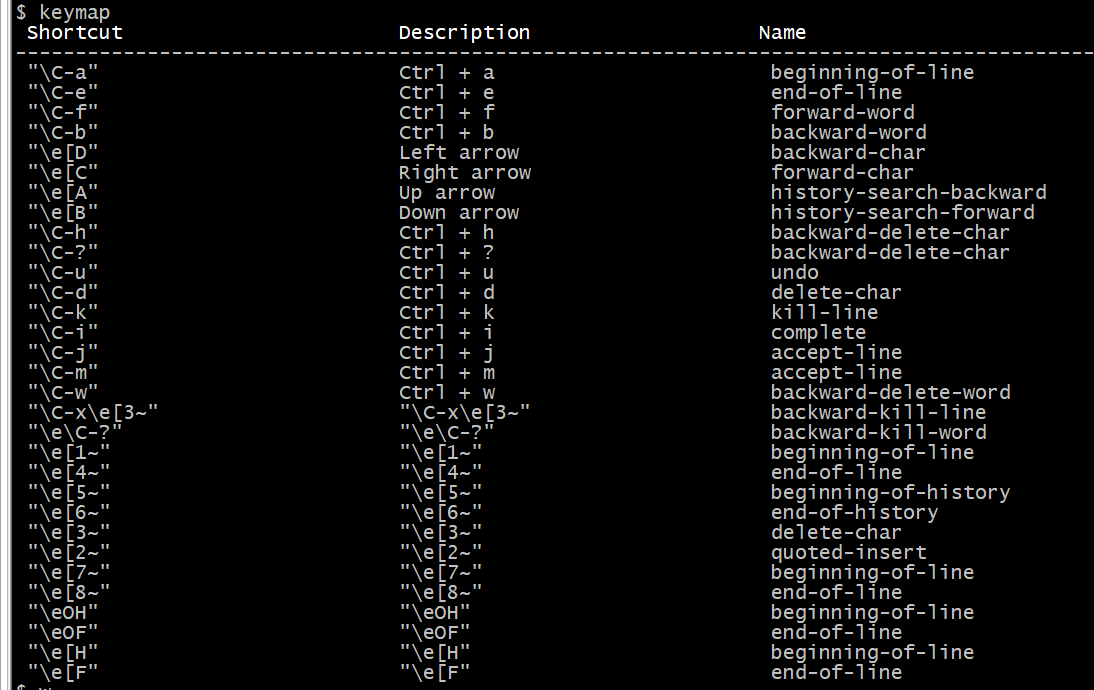
退出当前 Arthas 客户端，其他 Arthas 客户端不受影响

### shutdown

关闭 Arthas 服务端，所有 Arthas 客户端全部退出

### keymap

命令输出当前的快捷键映射表



#### 默认的快捷键

| **快捷键** | **快捷键说明** | **命令名称** | **命令说明** |
| --- | --- | --- | --- |
| "\C-a" | ctrl + a | beginning-of-line | 跳到行首 |
| "\C-e" | ctrl + e | end-of-line | 跳到行尾 |
| "\C-f" | ctrl + f | forward-word | 向前移动一个单词 |
| "\C-b" | ctrl + b | backward-word | 向后移动一个单词 |
| "\e[D" | 键盘左方向键 | backward-char | 光标向前移动一个字符 |
| "\e[C" | 键盘右方向键 | forward-char | 光标向后移动一个字符 |
| "\e[B" | 键盘下方向键 | next-history | 下翻显示下一个命令 |
| "\e[A" | 键盘上方向键 | previous-history | 上翻显示上一个命令 |
| "\C-h" | ctrl + h | backward-delete-char | 向后删除一个字符 |
| "\C-?" | ctrl + shift + / | backward-delete-char | 向后删除一个字符 |
| "\C-u" | ctrl + u | undo | 撤销上一个命令，相当于清空当前行 |
| "\C-d" | ctrl + d | delete-char | 删除当前光标所在字符 |
| "\C-k" | ctrl + k | kill-line | 删除当前光标到行尾的所有字符 |
| "\C-i" | ctrl + i | complete | 自动补全，相当于敲TAB |
| "\C-j" | ctrl + j | accept-line | 结束当前行，相当于敲回车 |
| "\C-m" | ctrl + m | accept-line | 结束当前行，相当于敲回车 |
| "\C-w" |  | backward-delete-word |  |
| "\C-x\e[3~" |  | backward-kill-line |  |
| "\e\C-?" |  | backward-kill-word |  |

#### 自定义快捷键

在当前用户目录下新建$USER\_HOME/.arthas/conf/inputrc文件，加入自定义配置。

假设我是vim的重度用户，我要把ctrl+h设置为光标向前一个字符，则设置如下，首先拷贝默认配置

"\C-a": beginning-of-line

"\C-e": end-of-line

"\C-f": forward-word

"\C-b": backward-word

"\e[D": backward-char

"\e[C": forward-char

"\e[B": next-history

"\e[A": previous-history

"\C-h": backward-delete-char

"\C-?": backward-delete-char

"\C-u": undo

"\C-d": delete-char

"\C-k": kill-line

"\C-i": complete

"\C-j": accept-line

"\C-m": accept-line

"\C-w": backward-delete-word

"\C-x\e[3~": backward-kill-line

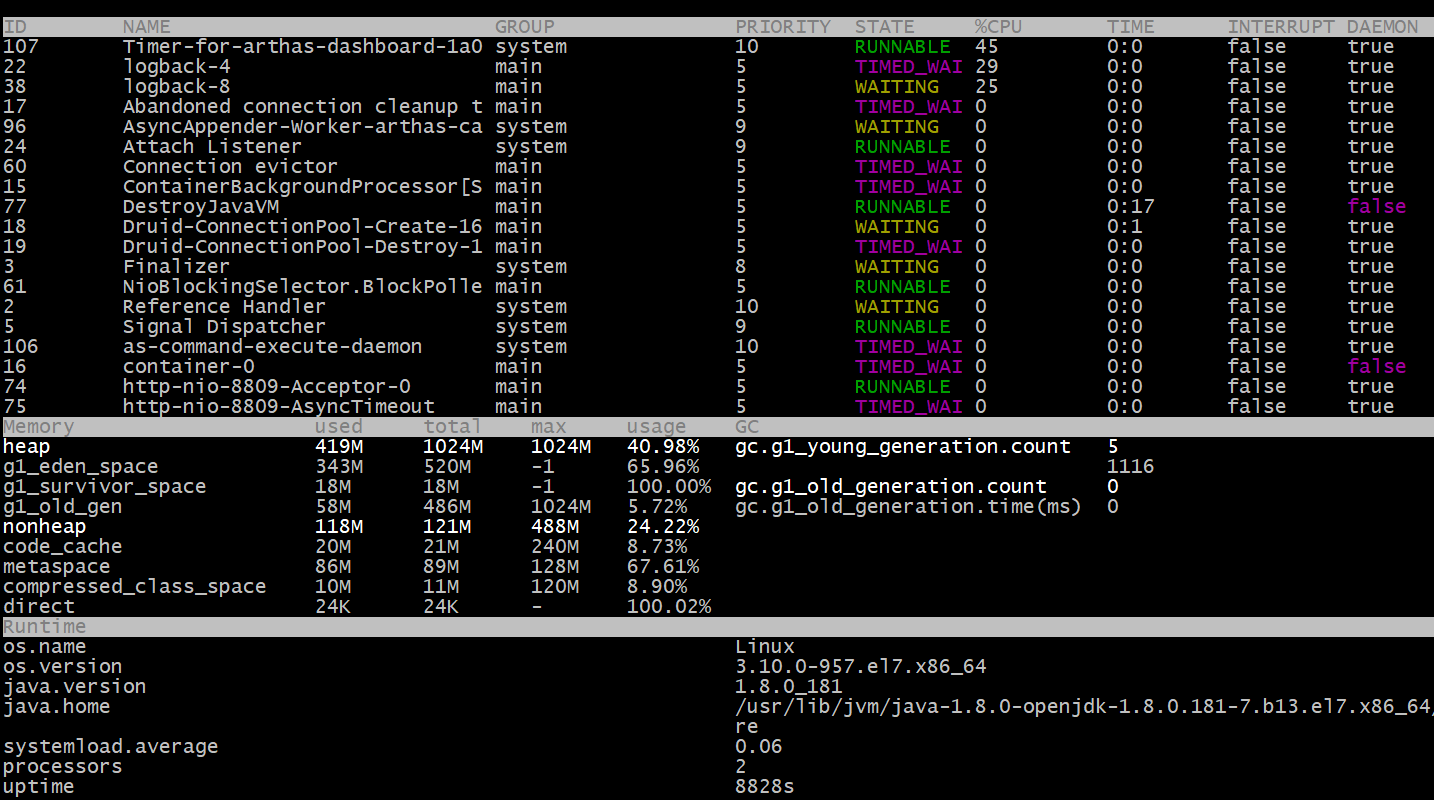
"\e\C-?": backward-kill-word

然后把"\C-h": backward-delete-char换成"\C-h": backward-char，然后重新连接即可。

## jvm相关

### dashboard

当前系统的实时数据面板

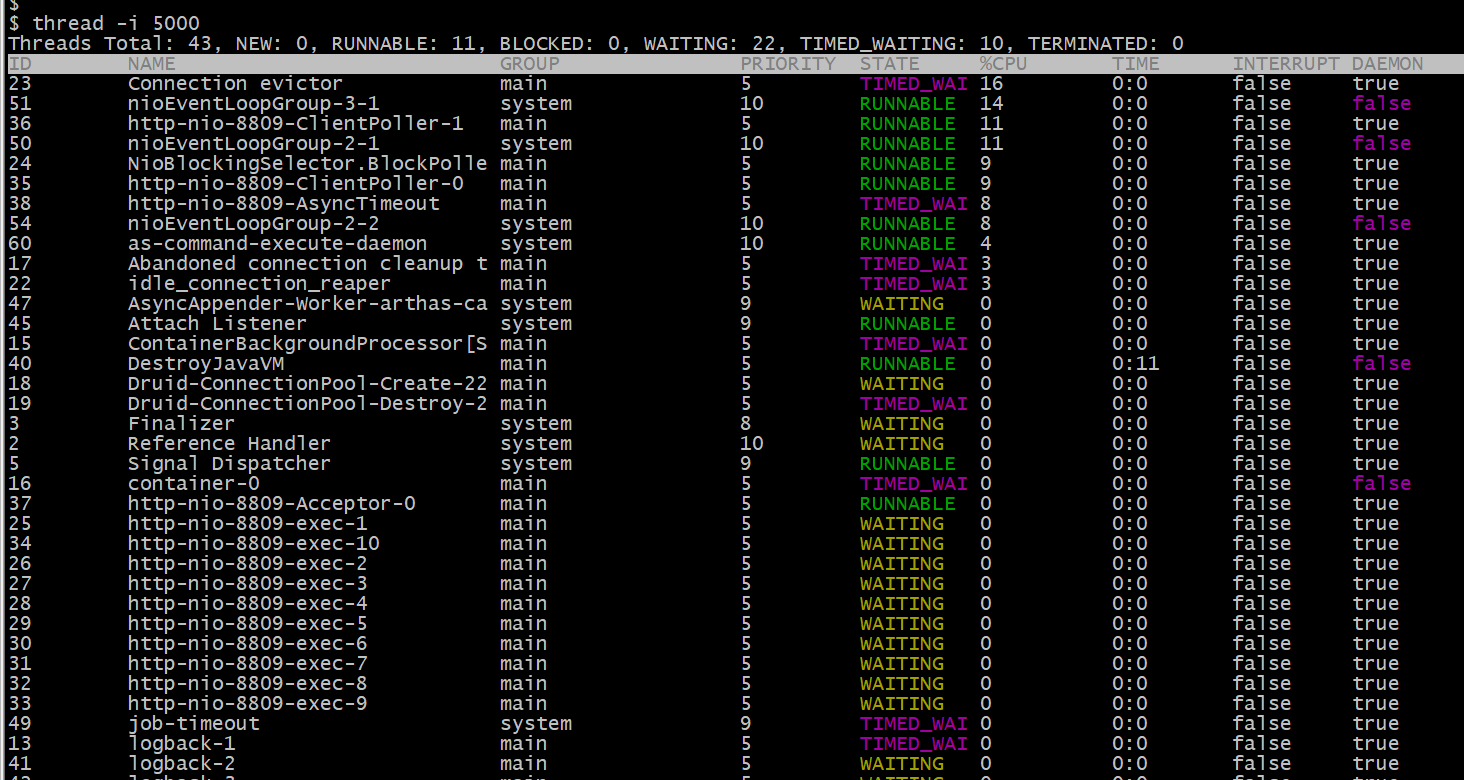


#### 数据说明

* ID: Java级别的线程ID，注意这个ID不能跟jstack中的nativeID一一对应
* NAME: 线程名
* GROUP: 线程组名
* PRIORITY: 线程优先级, 1~10之间的数字，越大表示优先级越高
* STATE: 线程的状态
* CPU%: 线程消耗的cpu占比，采样100ms，将所有线程在这100ms内的cpu使用量求和，再算出每个线程的cpu使用占比。
* TIME: 线程运行总时间，数据格式为分：秒
* INTERRUPTED: 线程当前的中断位状态
* DAEMON: 是否是daemon线程

### thread

查看当前jvm的线程堆栈信息.



#### 参数说明

| **参数名称** | **参数说明** |
| --- | --- |
| id | 线程id |
| [n:] | 指定最忙的前N个线程并打印堆栈 |
| [b] | 找出当前阻塞其他线程的线程 |
| [i <value>] | 指定cpu占比统计的采样间隔，单位为毫秒 |

Cpu占比是如何统计出来的?

这里的cpu统计的是，一段采样间隔内，当前JVM里各个线程所占用的cpu时间占总cpu时间的百分比。其计算方法为:

首先进行一次采样，获得所有线程的cpu的使用时间(调用的是java.lang.management.ThreadMXBean#getThreadCpuTime这个接口)，然后睡眠一段时间，默认100ms，可以通过-i参数指定，然后再采样一次，最后得出这段时间内各个线程消耗的cpu时间情况，最后算出百分比。

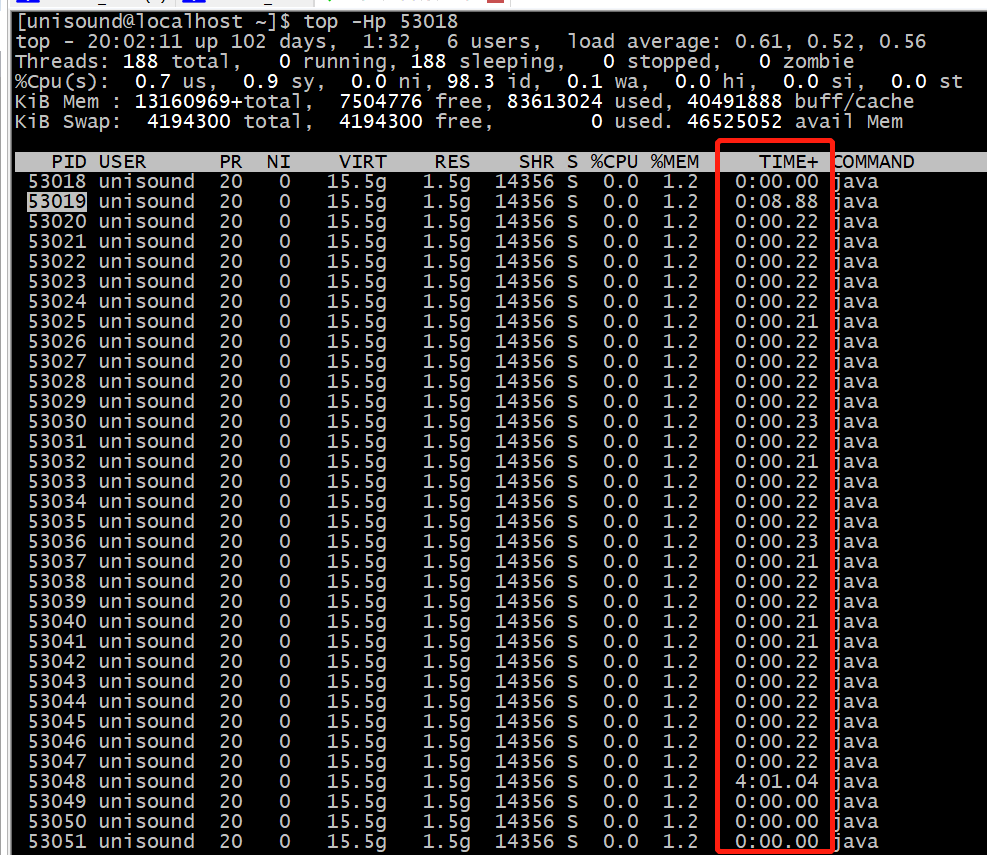
注意： 这个统计也会产生一定的开销（JDK这个接口本身开销比较大），因此会看到as的线程占用一定的百分比，为了降低统计自身的开销带来的影响，可以把采样间隔拉长一些，比如5000毫秒。

#### jvm启动到现在各线程cpu占比

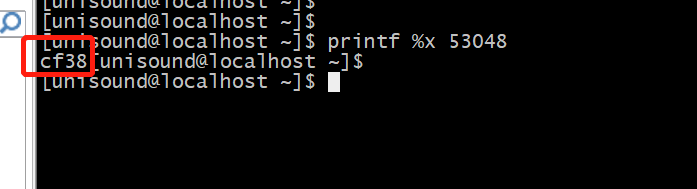
查找jvm启动到当前时间, 最耗cpu线程信息.

##### 方法一

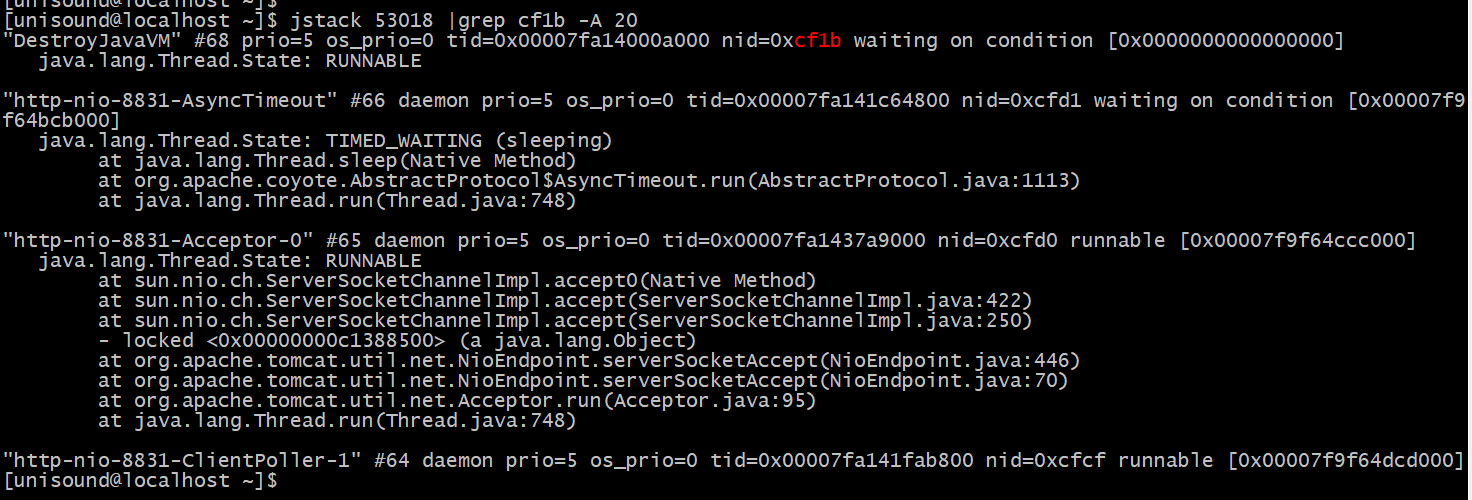
1. 查找到对应java进程pid
2. 使用top –Hp pid 获取该java进程 中所有线程占用cpu时间信息.



1. 从上一步获取到最耗时线程pid 53048, 手动转换成16进制.



1. 使用jstack {java进程ID} | grep cf38 -A 20



该方法比较繁琐.

##### 方法二

使用第三方脚本. <https://github.com/oldratlee/useful-scripts>

###### 安装脚本

* git clone git://github.com/oldratlee/useful-scripts.git
* cd useful-scripts
* git checkout release //使用发布版本
* 配置/etc/profile , 方便使用所有脚本..

###### 使用方式

<https://github.com/oldratlee/useful-scripts/blob/master/docs/java.md#-show-busy-java-threads>

# 从所有运行的Java进程中找出最消耗CPU的线程（缺省5个），打印出其线程栈

show-busy-java-threads

# 缺省会自动从所有的Java进程中找出最消耗CPU的线程，这样用更方便

# 当然你可以手动指定要分析的Java进程Id，以保证只会显示你关心的那个Java进程的信息

show-busy-java-threads -p <指定的Java进程Id>

show-busy-java-threads -c <要显示的线程栈数>

show-busy-java-threads <重复执行的间隔秒数> [<重复执行的次数>]

# 多次执行；这2个参数的使用方式类似vmstat命令

show-busy-java-threads -a <运行输出的记录到的文件>

# 记录到文件以方便回溯查看

show-busy-java-threads -S <存储jstack输出文件的目录>

# 指定jstack输出文件的存储目录，方便记录以后续分析

##############################

# 注意：

##############################

# 如果Java进程的用户 与 执行脚本的当前用户 不同，则jstack不了这个Java进程

# 为了能切换到Java进程的用户，需要加sudo来执行，即可以解决：

sudo show-busy-java-threads

show-busy-java-threads -s <指定jstack命令的全路径>

# 对于sudo方式的运行，JAVA\_HOME环境变量不能传递给root，

# 而root用户往往没有配置JAVA\_HOME且不方便配置，

# 显式指定jstack命令的路径就反而显得更方便了

# -m选项：执行jstack命令时加上-m选项，显示上Native的栈帧，一般应用排查不需要使用

show-busy-java-threads -m

# -F选项：执行jstack命令时加上 -F 选项（如果直接jstack无响应时，用于强制jstack），一般情况不需要使用

show-busy-java-threads -F

# -l选项：执行jstack命令时加上 -l 选项，显示上更多相关锁的信息，一般情况不需要使用

# 注意：和 -m -F 选项一起使用时，可能会大大增加jstack操作的耗时

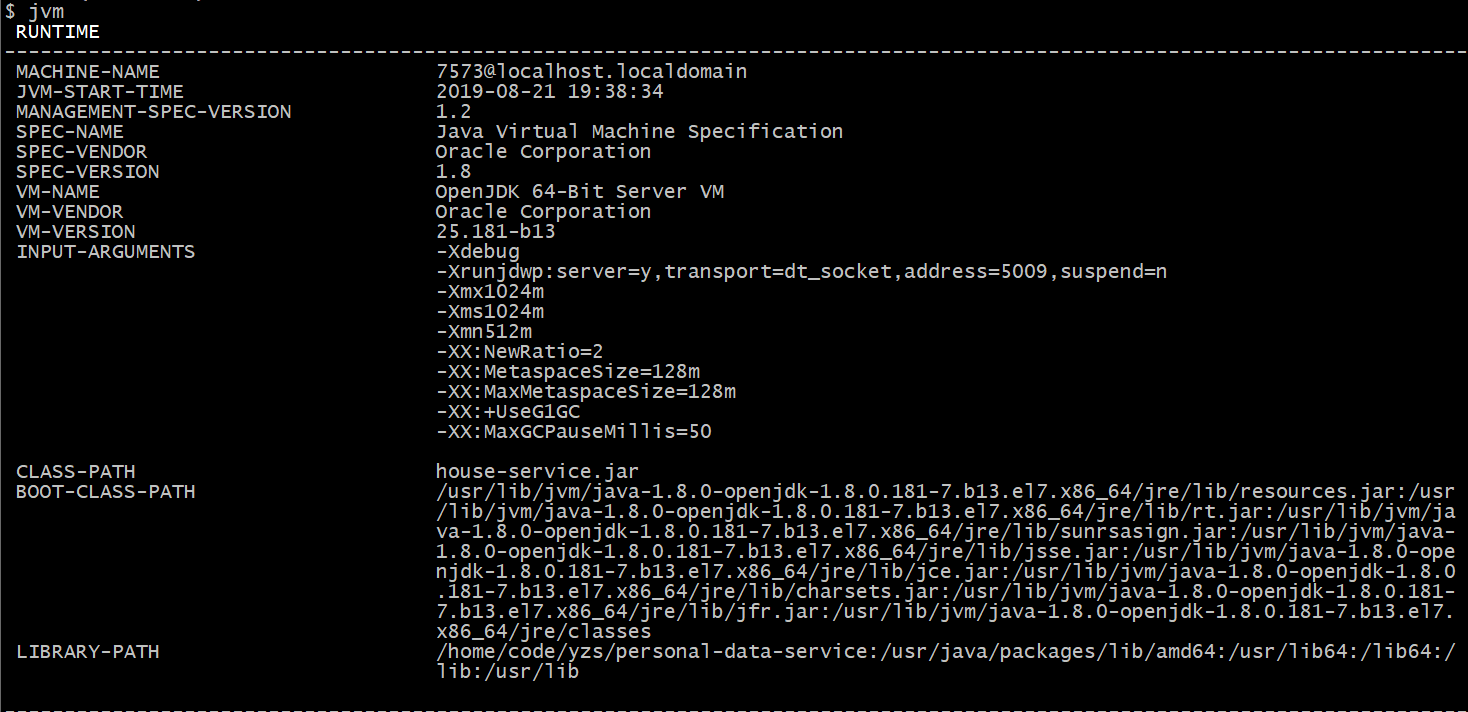
show-busy-java-threads -l

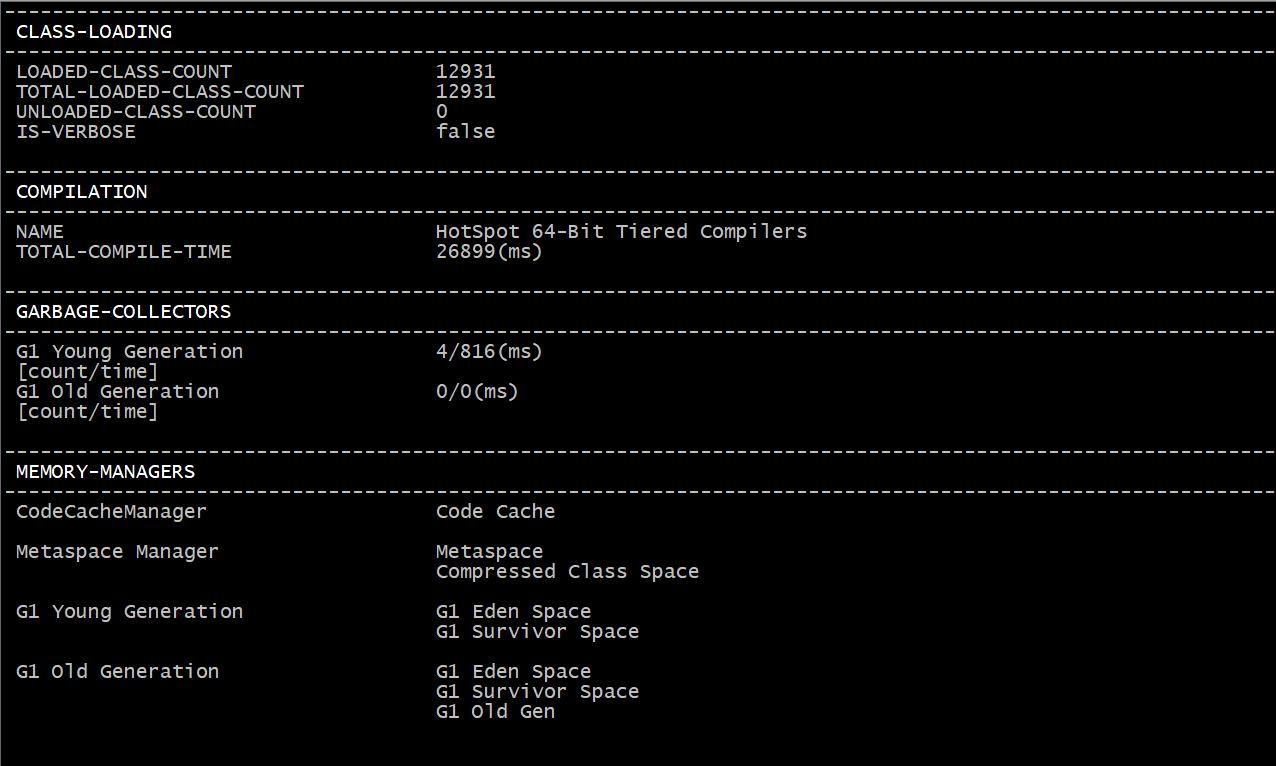
# 帮助信息

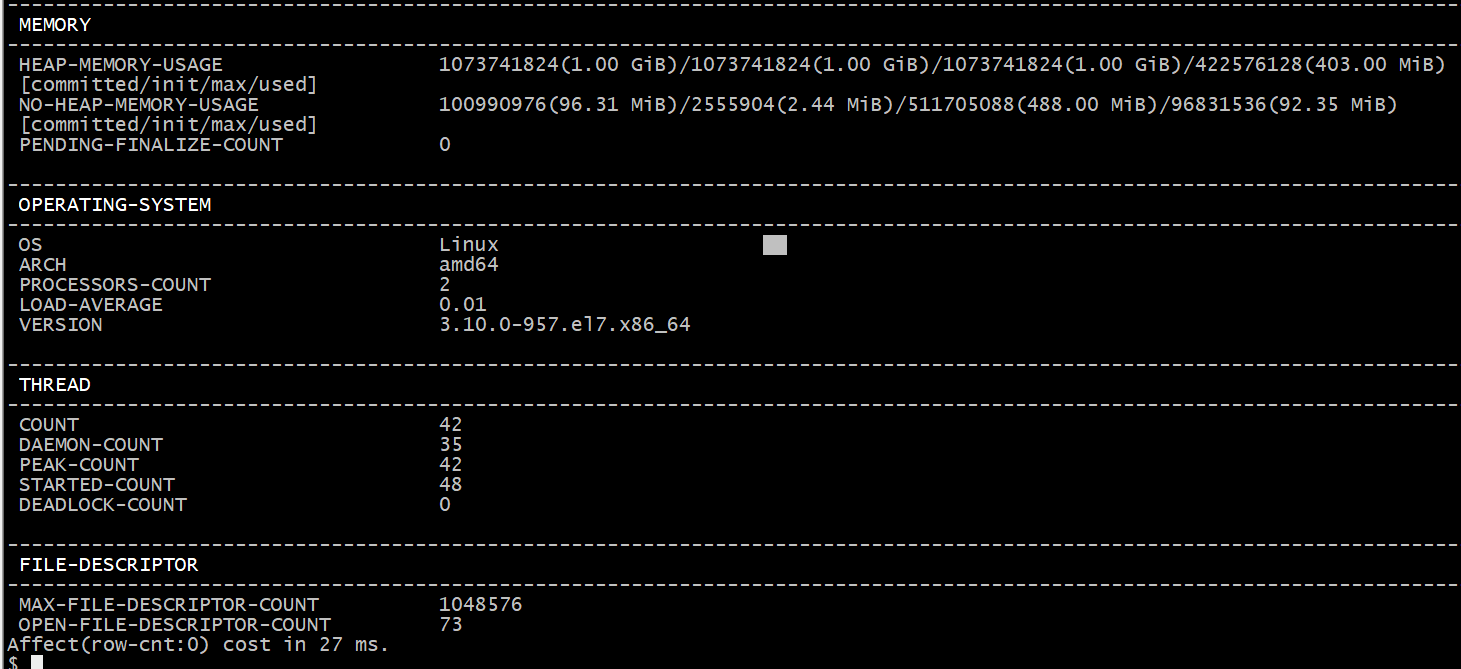
$ show-busy-java-threads -h

### jvm

查看当前JVM信息







#### THREAD相关

* COUNT: JVM当前活跃的线程数
* DAEMON-COUNT: JVM当前活跃的守护线程数
* PEAK-COUNT: 从JVM启动开始曾经活着的最大线程数
* STARTED-COUNT: 从JVM启动开始总共启动过的线程次数
* DEADLOCK-COUNT: JVM当前死锁的线程数

#### 文件描述符相关

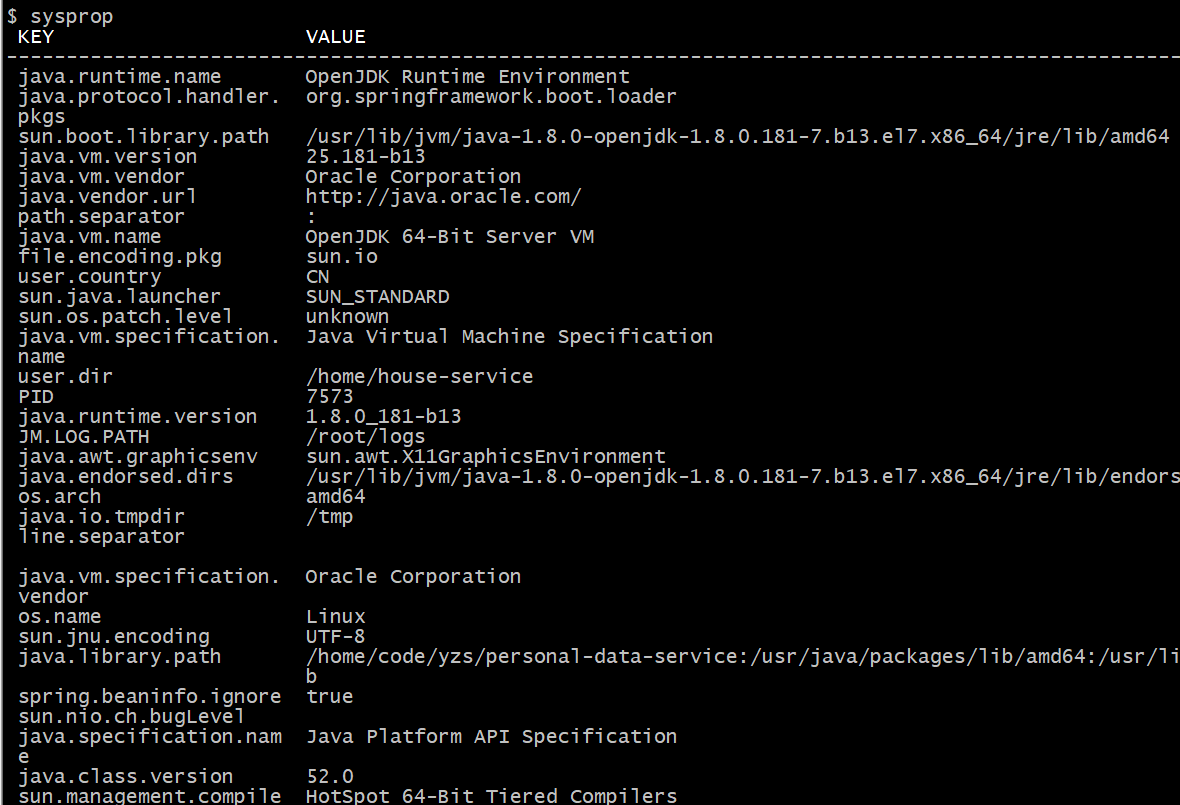
* MAX-FILE-DESCRIPTOR-COUNT：JVM进程最大可以打开的文件描述符数
* OPEN-FILE-DESCRIPTOR-COUNT：JVM当前打开的文件描述符数

### sysprop

查看和修改JVM的系统属性

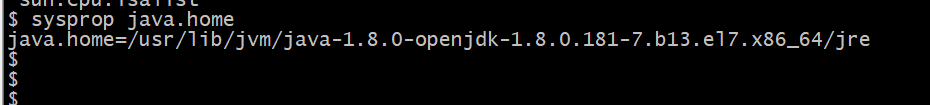
#### 查看所有属性

$ sysprop



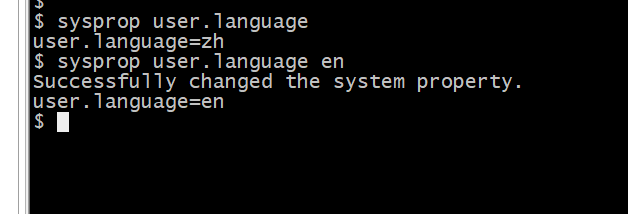
#### 查看单个属性

$ sysprop java.home



#### 修改单个属性

$ sysprop user.language en



### sysenv

查看jvm的环境变量

#### 查看所有环境变量

$ sysenv

查看单个环境变量

$ sysenv JRE\_HOME

### getstatic

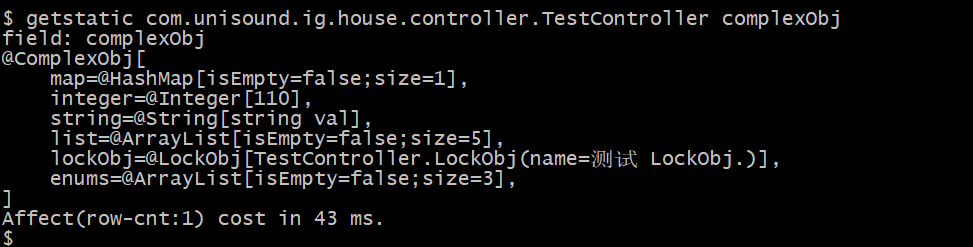
查看类的静态属性.

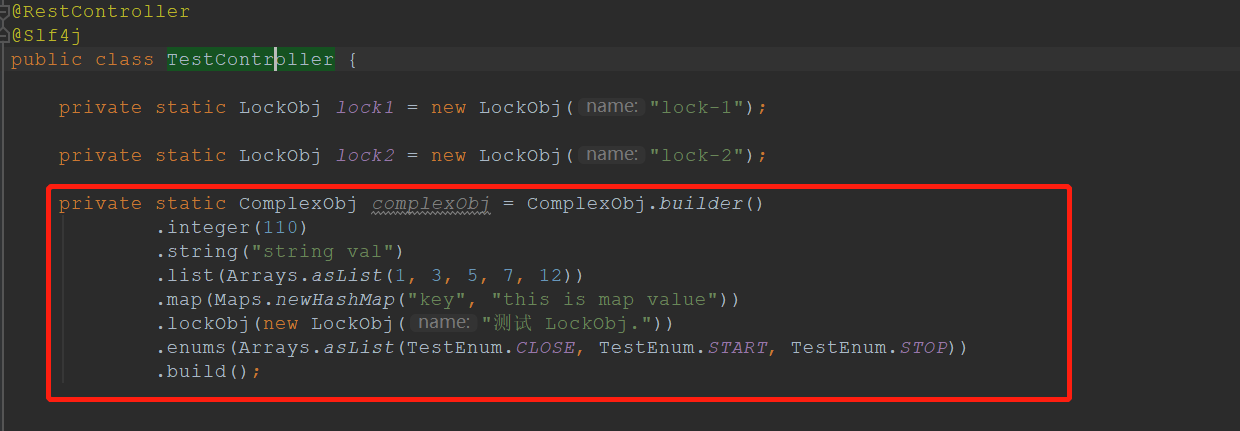
#### 使用方法

getstatic class\_name field\_name

#### 使用示例

$ getstatic com.unisound.ig.house.controller.TestController complexObj





支持OGNL表达式.

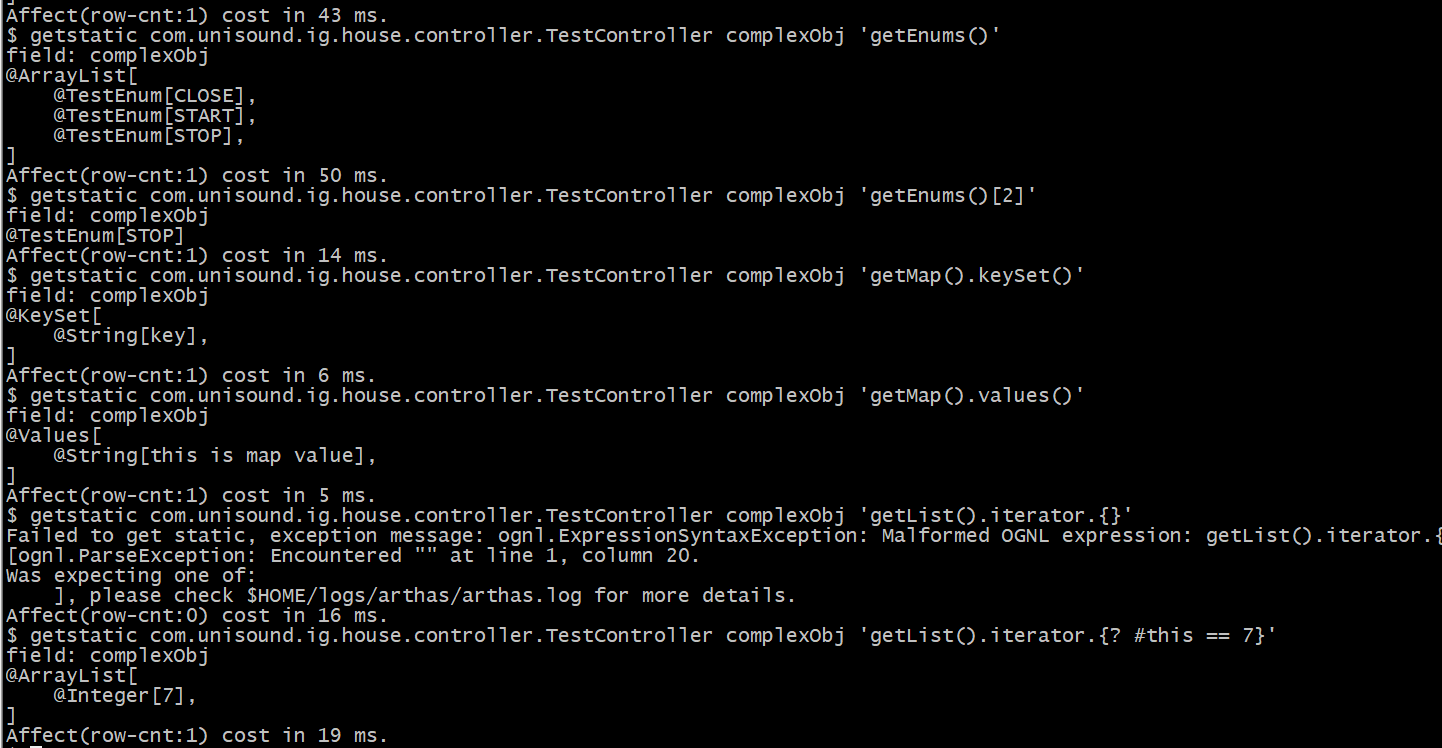
$ getstatic com.unisound.ig.house.controller.TestController complexObj 'getEnums()'

$ getstatic com.unisound.ig.house.controller.TestController complexObj 'getEnums()[2]'

$ getstatic com.unisound.ig.house.controller.TestController complexObj 'getMap().keySet()'

$ getstatic com.unisound.ig.house.controller.TestController complexObj 'getMap().values()'

$ getstatic com.unisound.ig.house.controller.TestController complexObj 'getList().iterator.{? #this == 7}'



### ognl

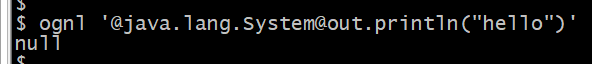
执行ognl表达式, 从3.0.5版本增加.

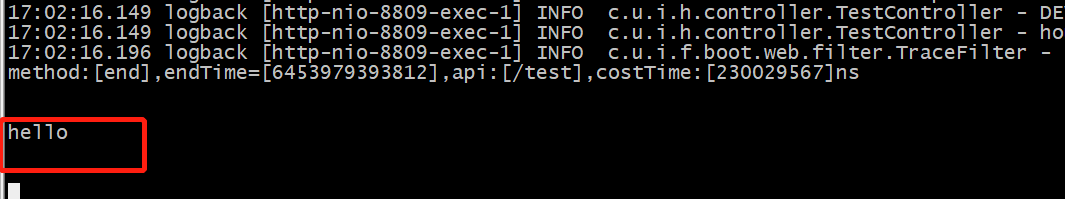
#### 参数说明

| **参数名称** | **参数说明** |
| --- | --- |
| *express* | 执行的表达式 |
| [c:] | 执行表达式的 ClassLoader 的 hashcode，默认值是SystemClassLoader |
| [x] | 结果对象的展开层次，默认值1 |

#### 调用静态函数

$ ognl '@java.lang.System@out.println("hello")'





#### 获取静态类的静态字段

* [sysenv](https://alibaba.github.io/arthas/sysenv.html)——查看JVM的环境变量
* [getstatic](https://alibaba.github.io/arthas/getstatic.html)——查看类的静态属性
* **New!** [ognl](https://alibaba.github.io/arthas/ognl.html)——执行ognl表达式
* **New!** [mbean](https://alibaba.github.io/arthas/mbean.html)——查看 Mbean 的信息

## class/classloader相关

* [sc](https://alibaba.github.io/arthas/sc.html)——查看JVM已加载的类信息
* [sm](https://alibaba.github.io/arthas/sm.html)——查看已加载类的方法信息
* [jad](https://alibaba.github.io/arthas/jad.html)——反编译指定已加载类的源码
* [mc](https://alibaba.github.io/arthas/mc.html)——内存编绎器，内存编绎.java文件为.class文件
* [redefine](https://alibaba.github.io/arthas/redefine.html)——加载外部的.class文件，redefine到JVM里
* [dump](https://alibaba.github.io/arthas/dump.html)——dump 已加载类的 byte code 到特定目录
* [classloader](https://alibaba.github.io/arthas/classloader.html)——查看classloader的继承树，urls，类加载信息，使用classloader去getResource

## monitor/watch/trace相关

请注意，这些命令，都通过字节码增强技术来实现的，会在指定类的方法中插入一些切面来实现数据统计和观测，因此在线上、预发使用时，请尽量明确需要观测的类、方法以及条件，诊断结束要执行 shutdown 或将增强过的类执行 reset 命令。

* [monitor](https://alibaba.github.io/arthas/monitor.html)——方法执行监控
* [watch](https://alibaba.github.io/arthas/watch.html)——方法执行数据观测
* [trace](https://alibaba.github.io/arthas/trace.html)——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](https://alibaba.github.io/arthas/stack.html)——输出当前方法被调用的调用路径
* [tt](https://alibaba.github.io/arthas/tt.html)——方法执行数据的时空隧道，记录下指定方法每次调用的入参和返回信息，并能对这些不同的时间下调用进行观测

## options

* [options](https://alibaba.github.io/arthas/options.html)——查看或设置Arthas全局开关

## 管道

Arthas支持使用管道对上述命令的结果进行进一步的处理，如sm java.lang.String \* | grep 'index'

* grep——搜索满足条件的结果
* plaintext——将命令的结果去除ANSI颜色
* wc——按行统计输出结果

## 后台异步任务

当线上出现偶发的问题，比如需要watch某个条件，而这个条件一天可能才会出现一次时，异步后台任务就派上用场了.