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 | 2019-08-27 | 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

## Arthas版本

当前版本 3.1.1

# 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

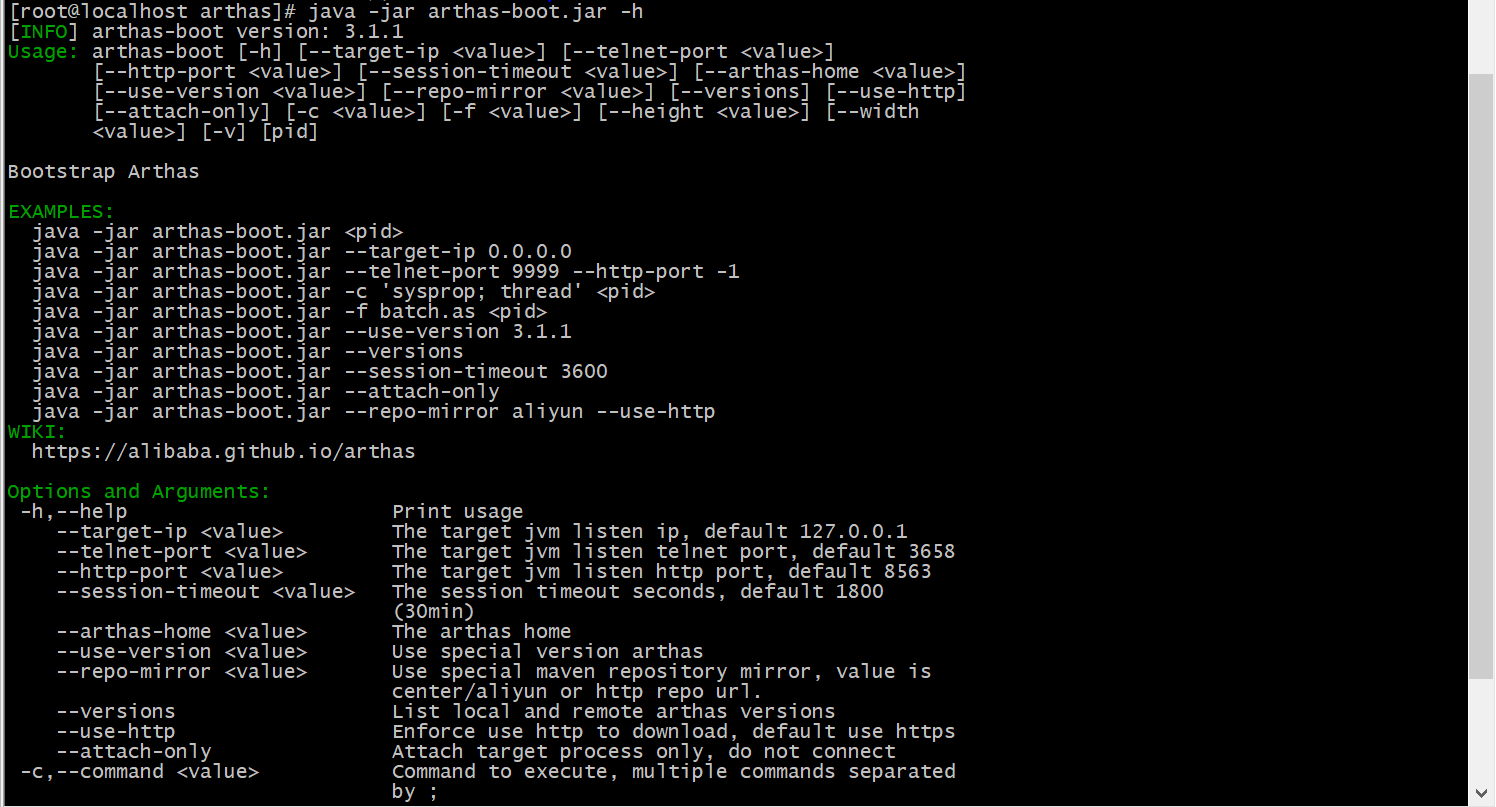
<https://alibaba.github.io/arthas/docker.html>

## 3.2 快速入门

### 3.2.1 启动

查看启动帮助

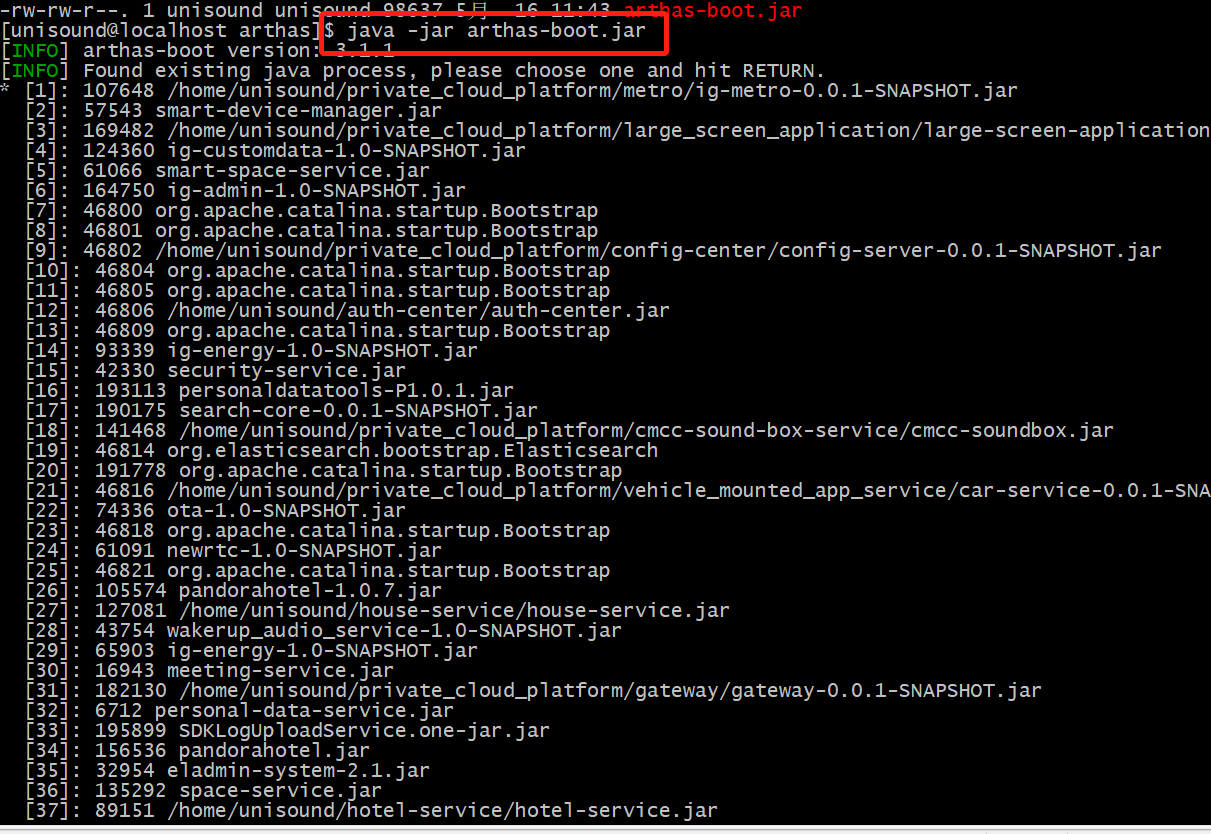
$ java -jar arthas-boot.jar –h



直接jar包启动.

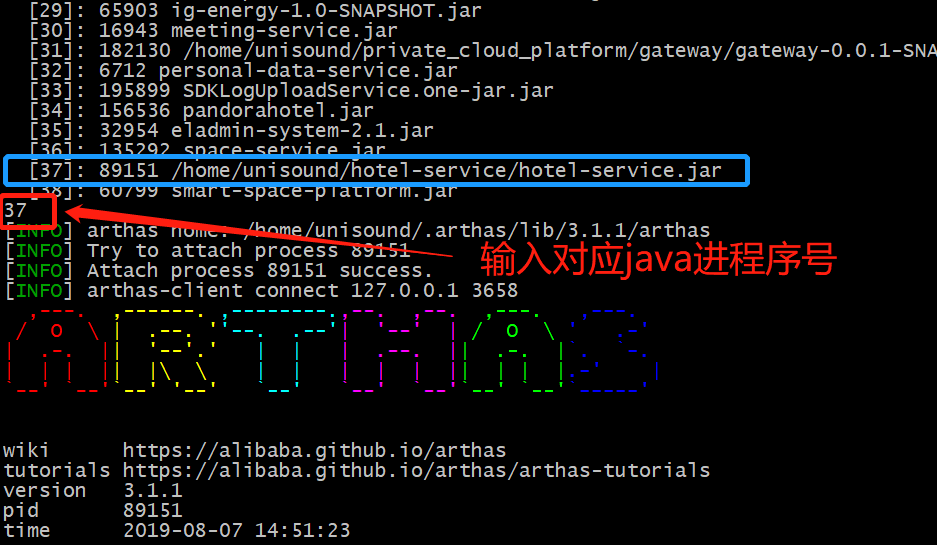
java -jar arthas-boot.jar

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

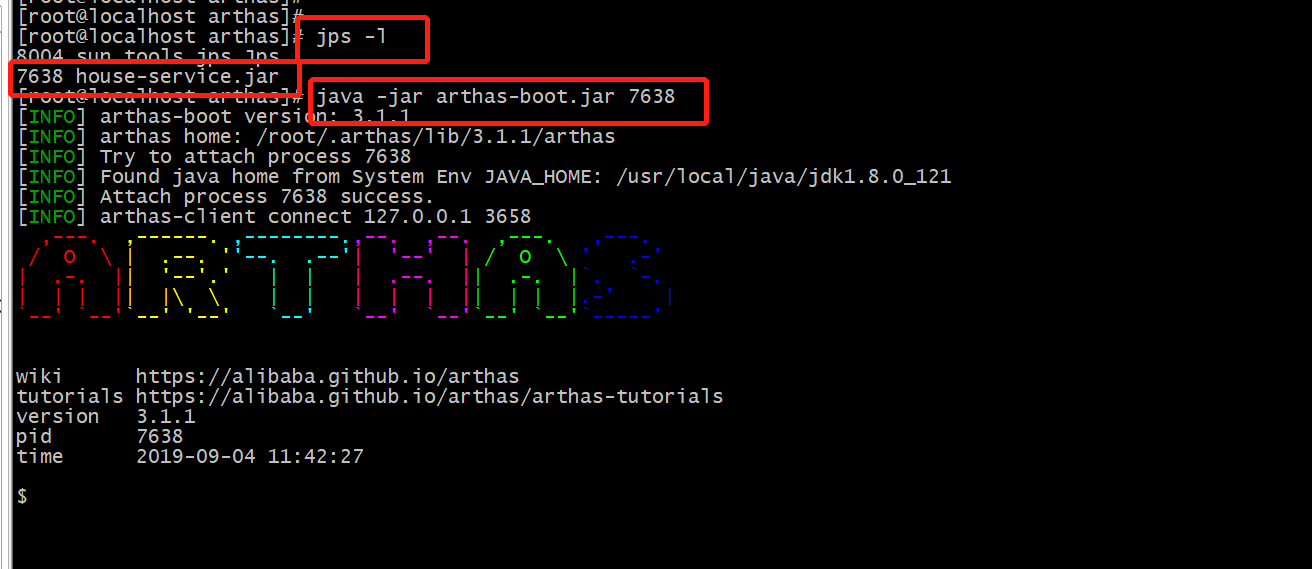
hotel-service 进程是 第37个, 输入37 按回车, Arthas 会attach到目标进程上.



### 3.2.3 直接attach到某一java进程

$ jps -l 找到对应java进程id

$ java -jar arthas-boot.jar {targetPid}



### 3.2.4 重新进入目标进程arthas命令界

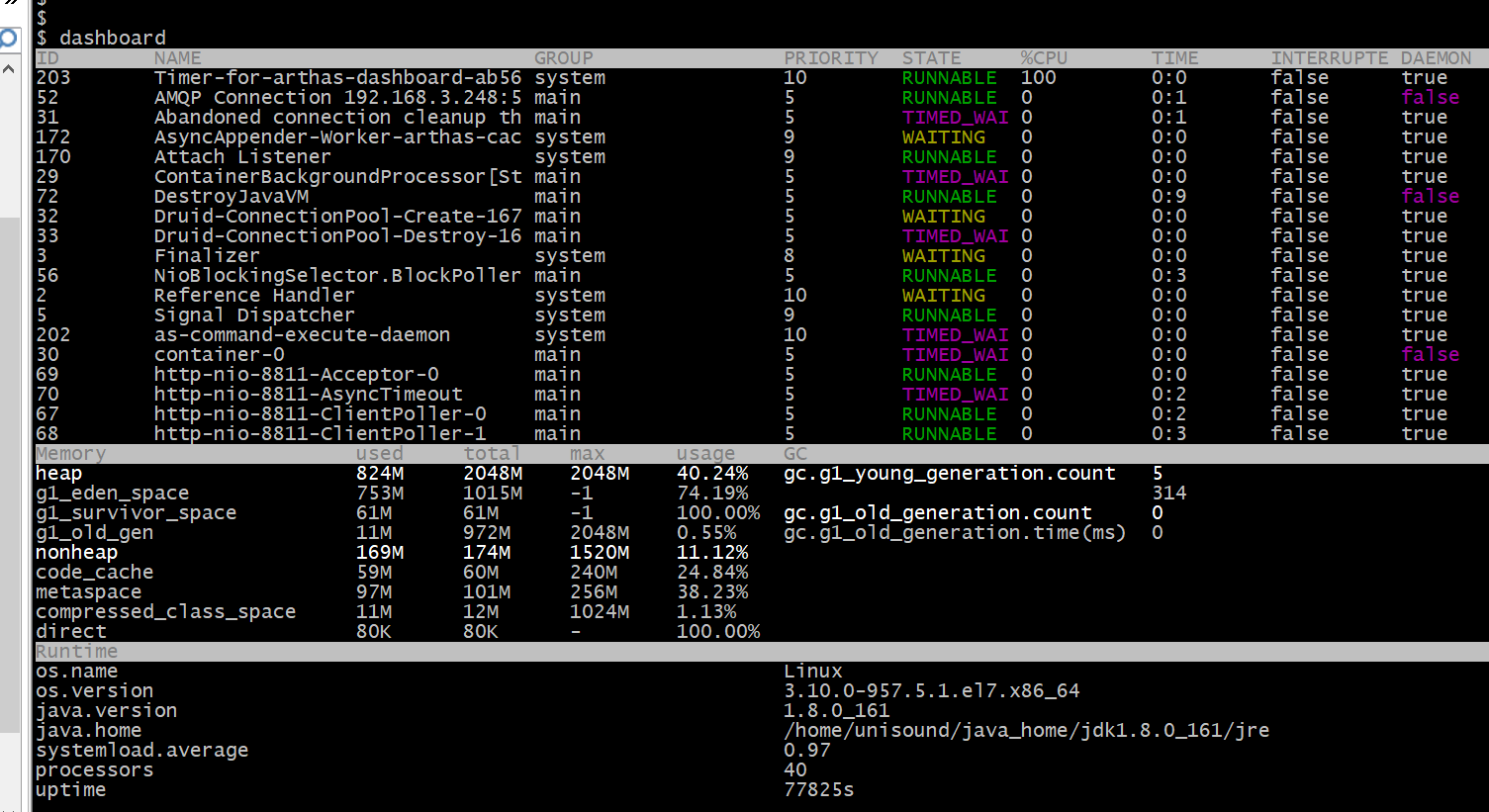
$ telnet 127.0.0.1 3658



### 3.2.5 基础命令

#### dashboard

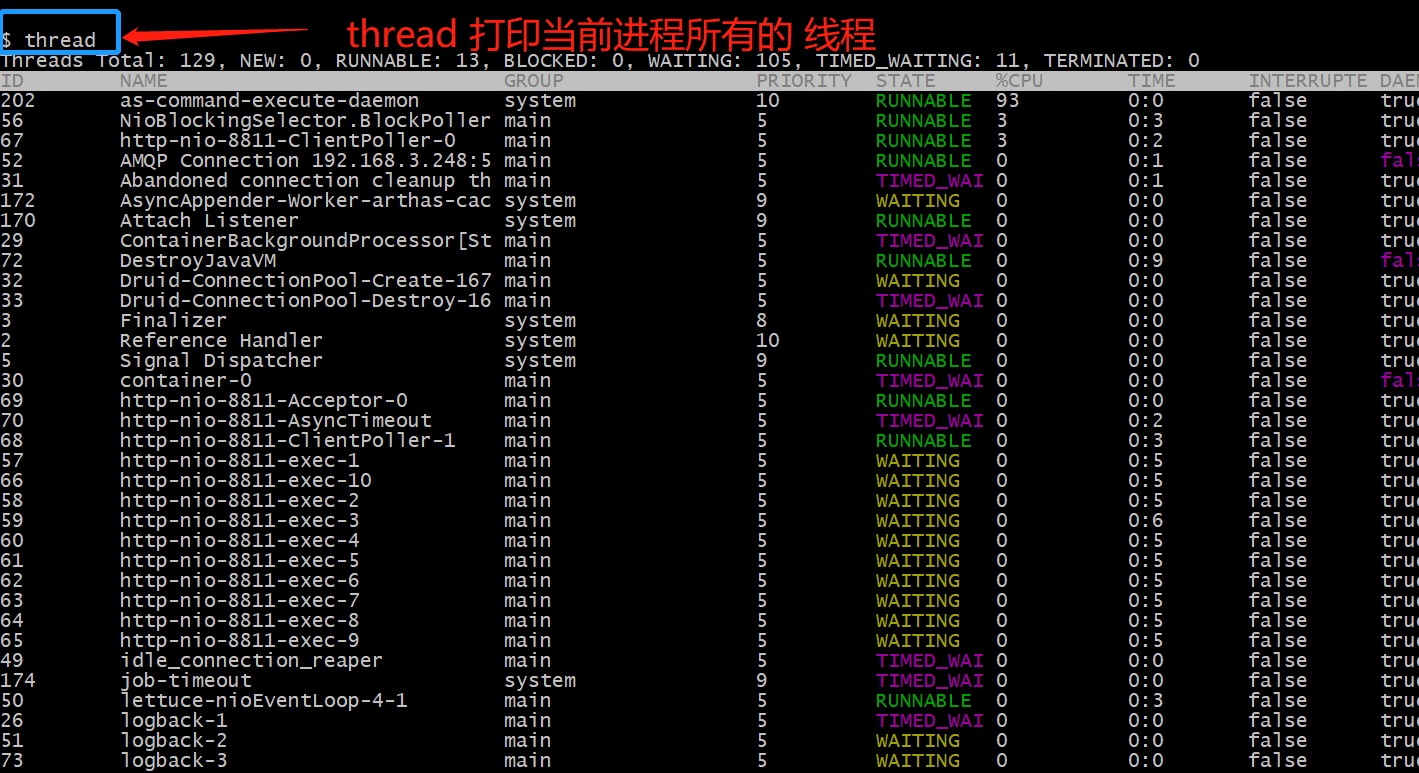
展示当前进程的信息.



#### thread

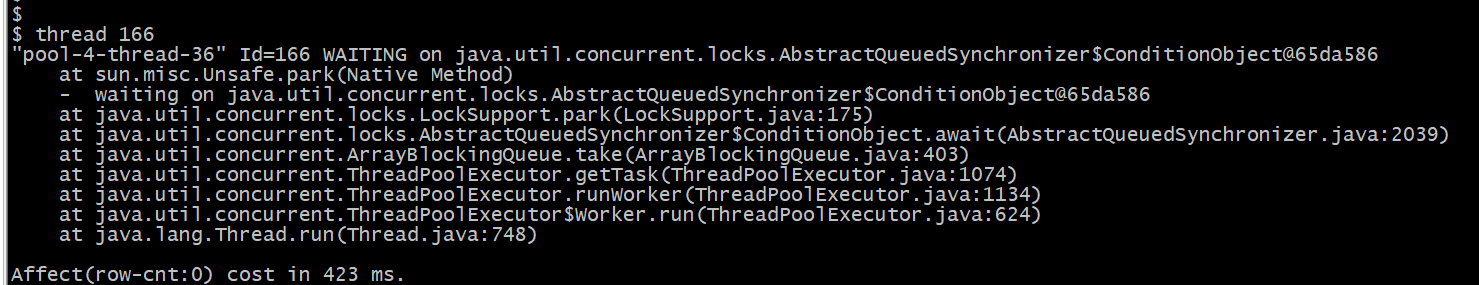
线程相关信息

thread 命令 展示所有线程信息.



thread + 线程序号

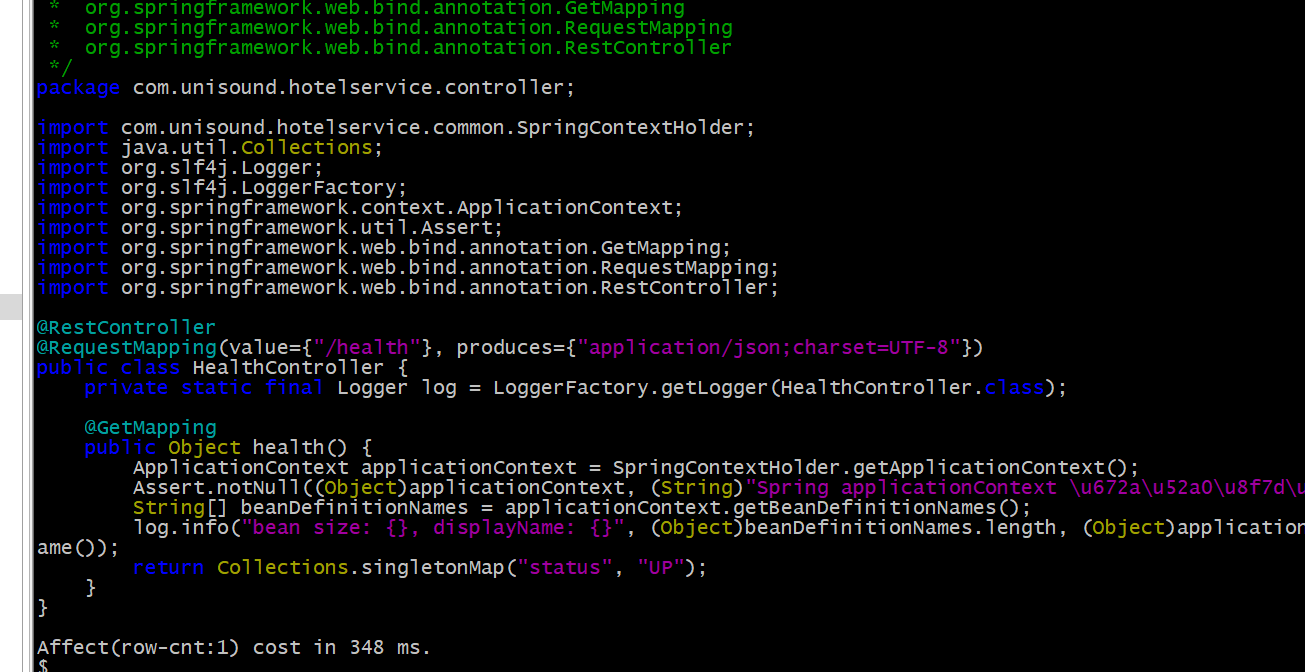
输出该线程的栈.



#### jad

反编译指令

jad com.unisound.hotelservice.controller.HealthController

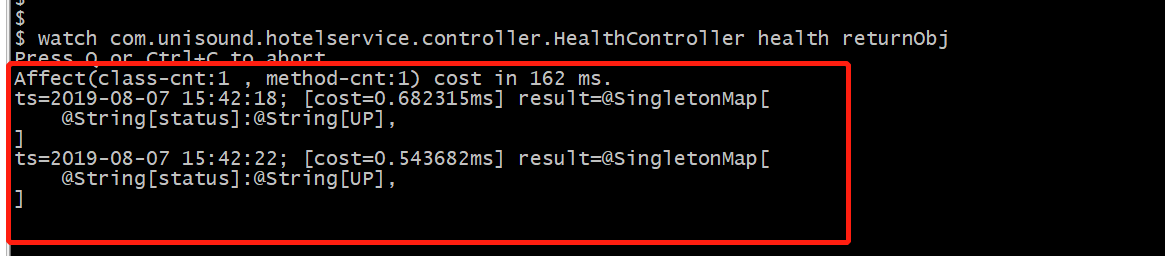


#### watch

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



### 3.2.6 退出arthas

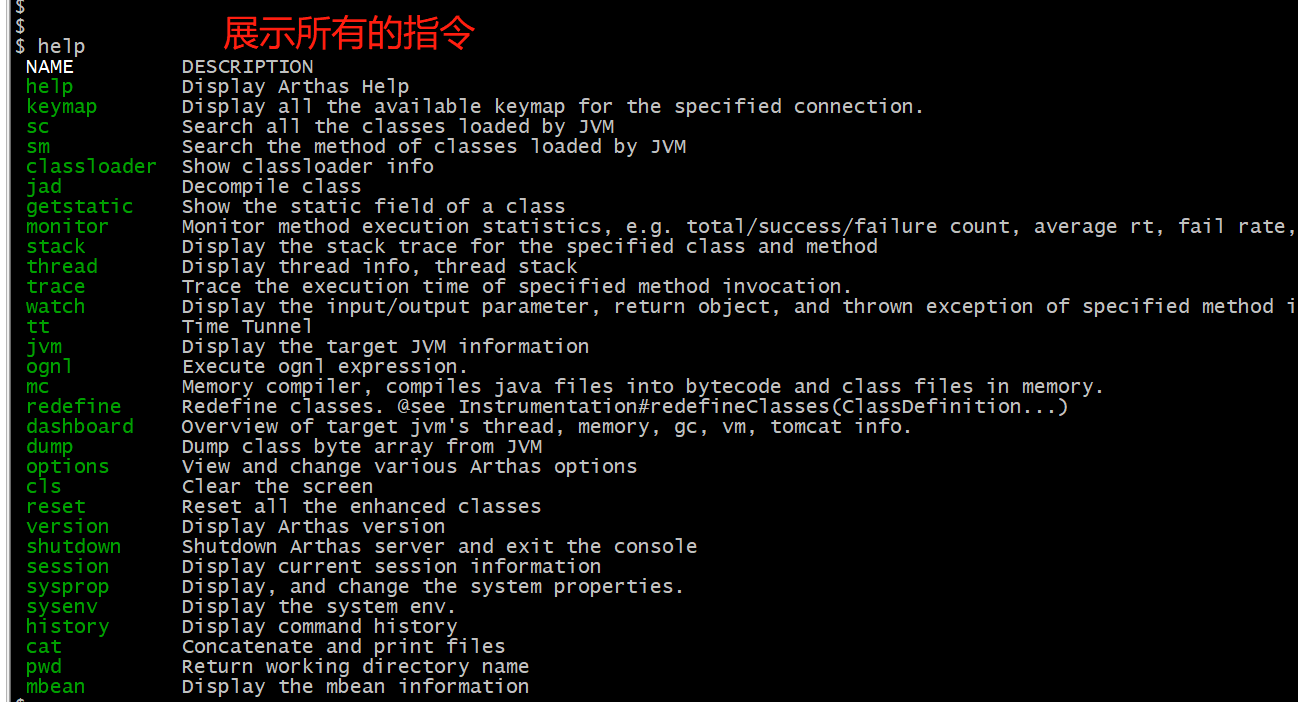
如果只是退出当前的连接，可以用quit或者exit命令。Attach到目标进程上的arthas还会继续运行，端口会保持开放，下次连接时可以直接连接上。

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

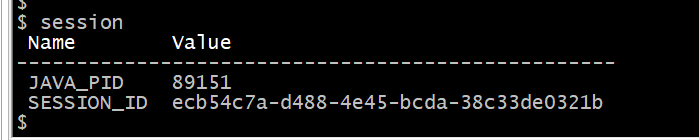
返回当前的工作目录，和linux命令类似

### cls

清空当前屏幕区域, 跟linux ctrl+l 命令类似

### session

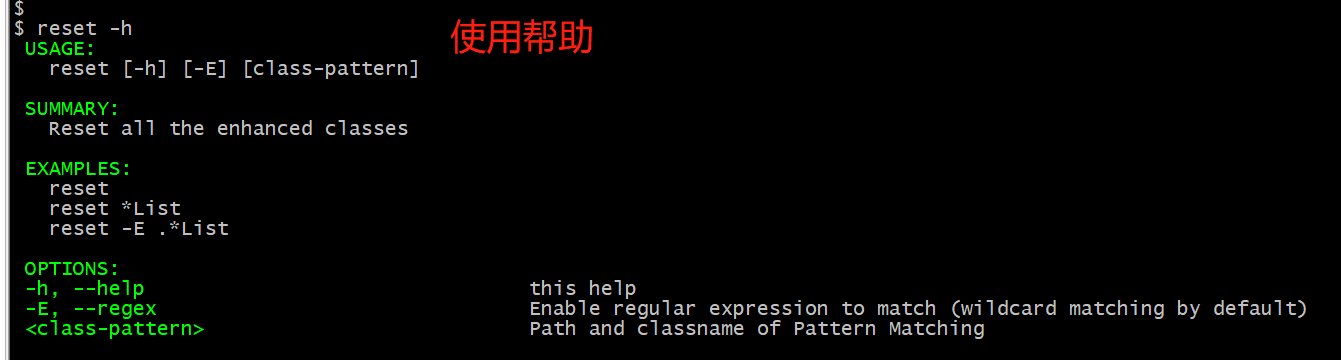
查看当前会话的信息



### 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

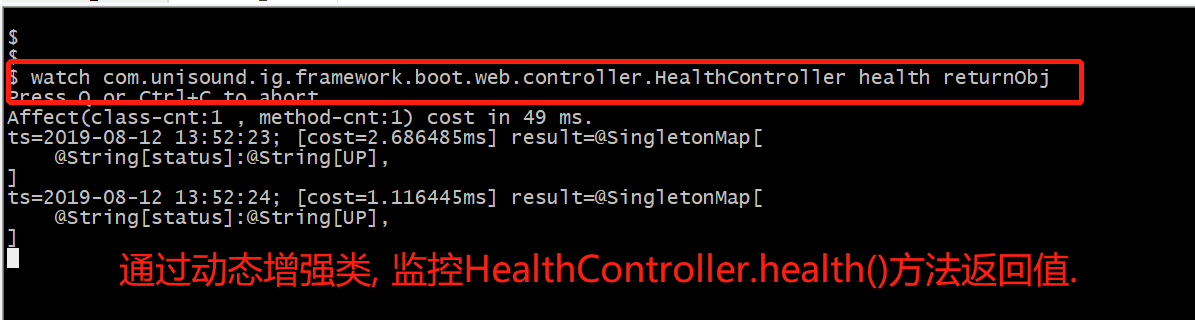


-E 开启正则表达式匹配

#### 示例

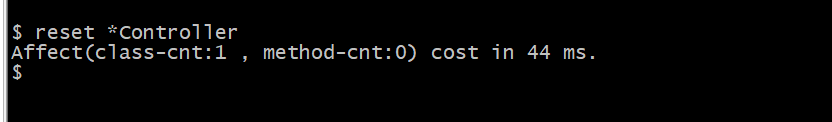
增强两个类

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

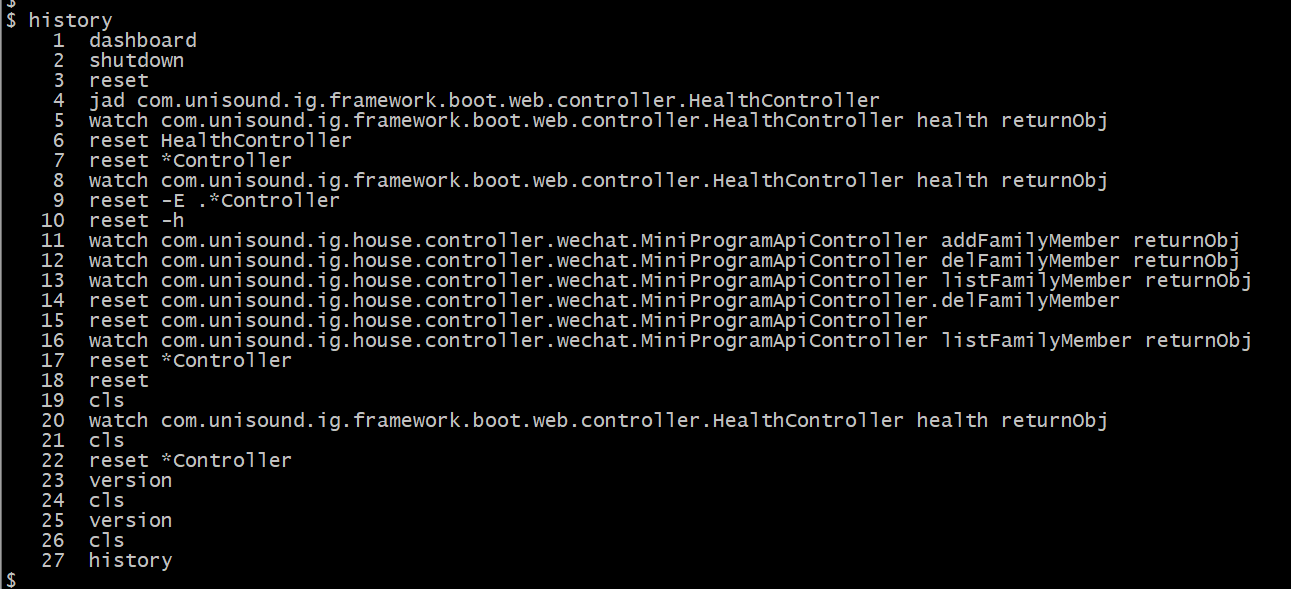


### version

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

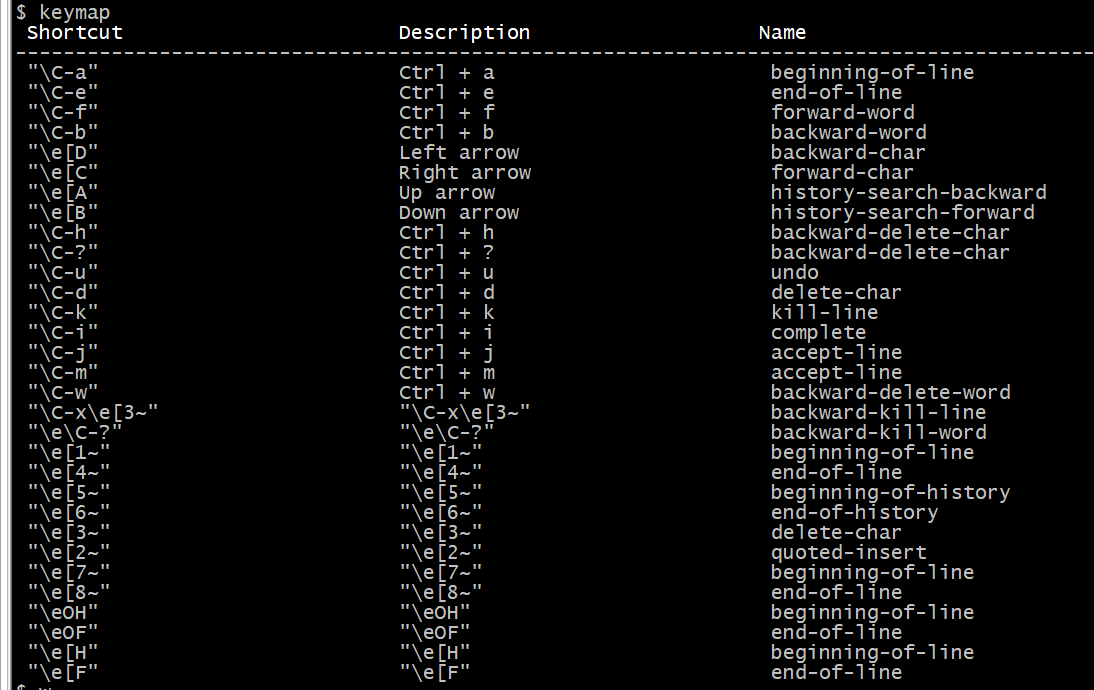
退出当前 Arthas 客户端，其他 Arthas 客户端不受影响

### 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

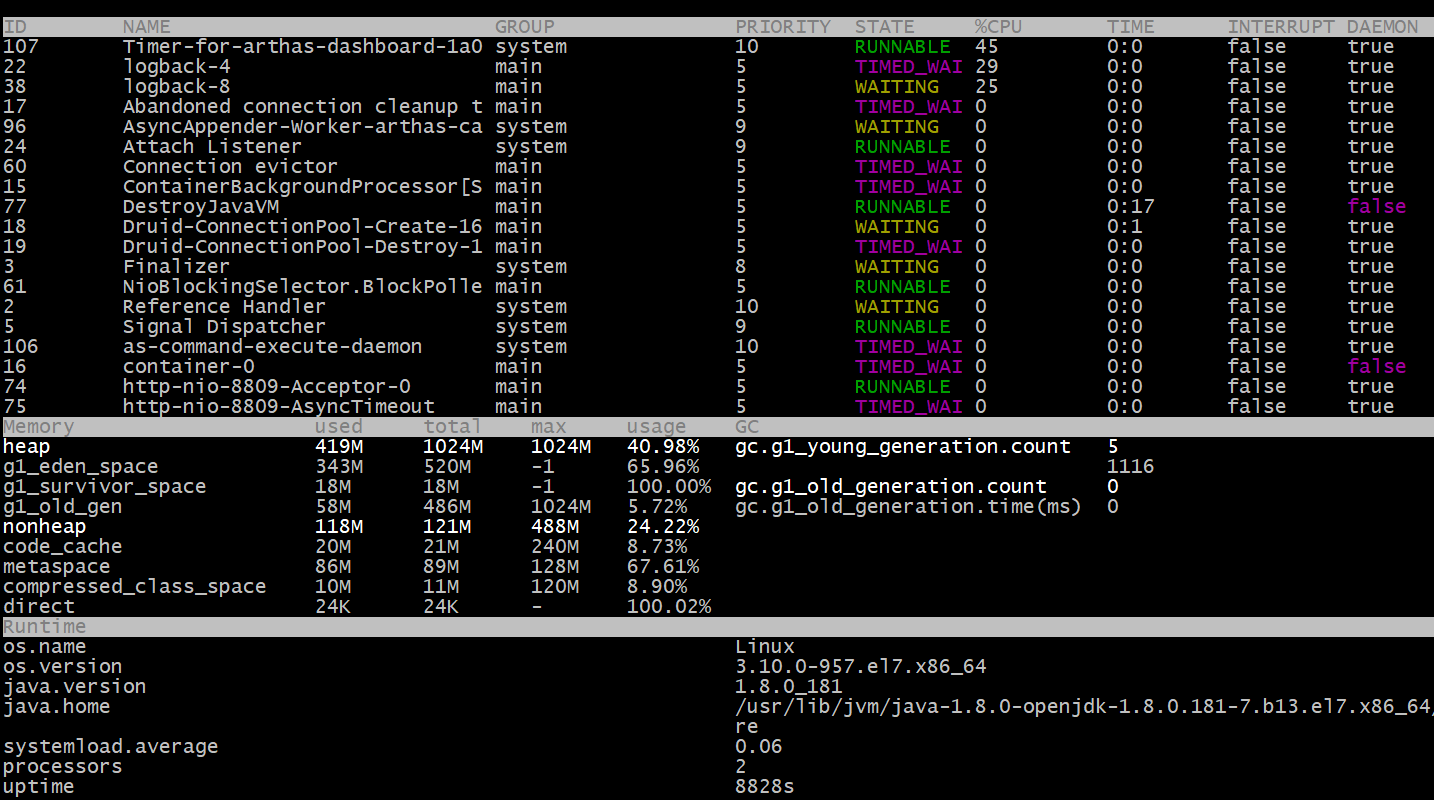
"\e\C-?": backward-kill-word

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

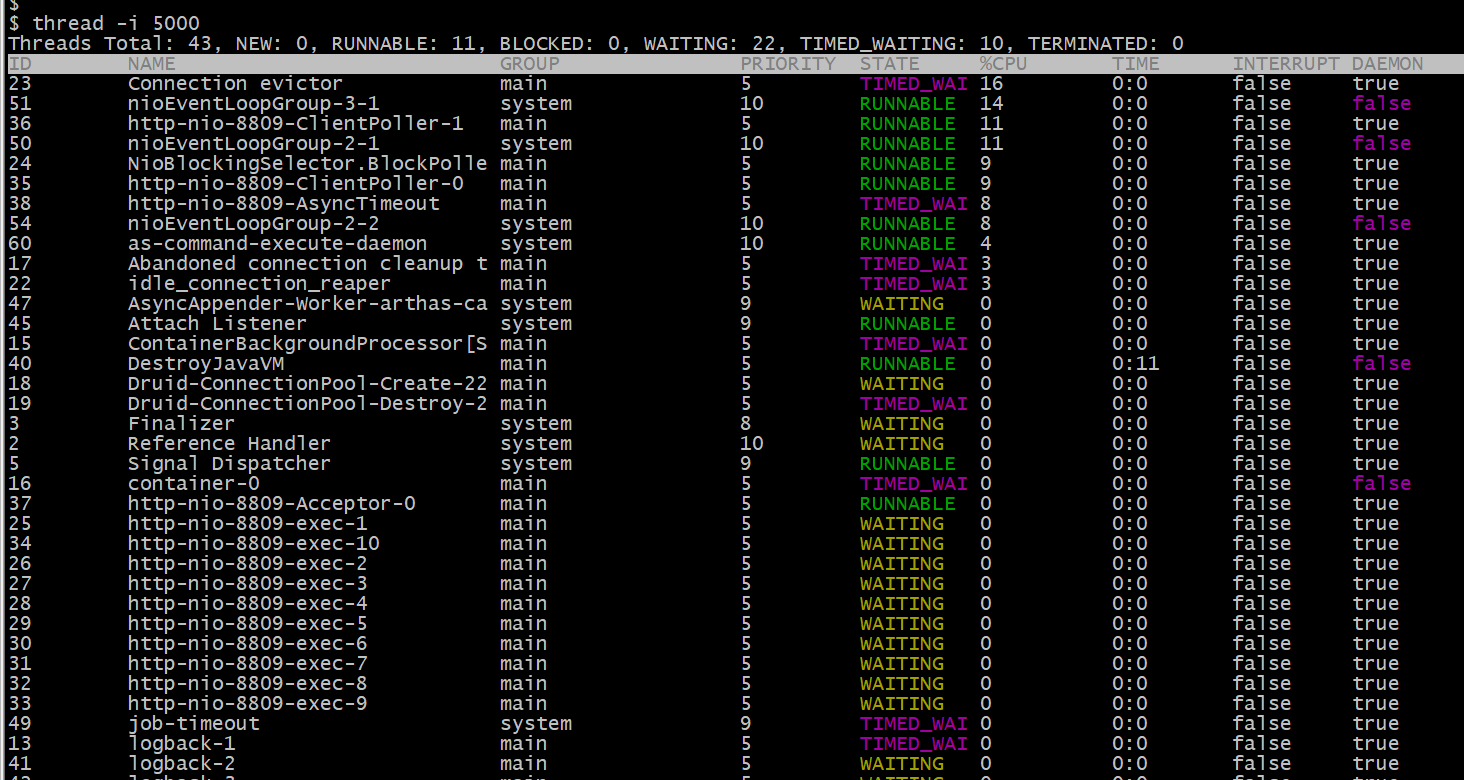


#### 数据说明

* 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。

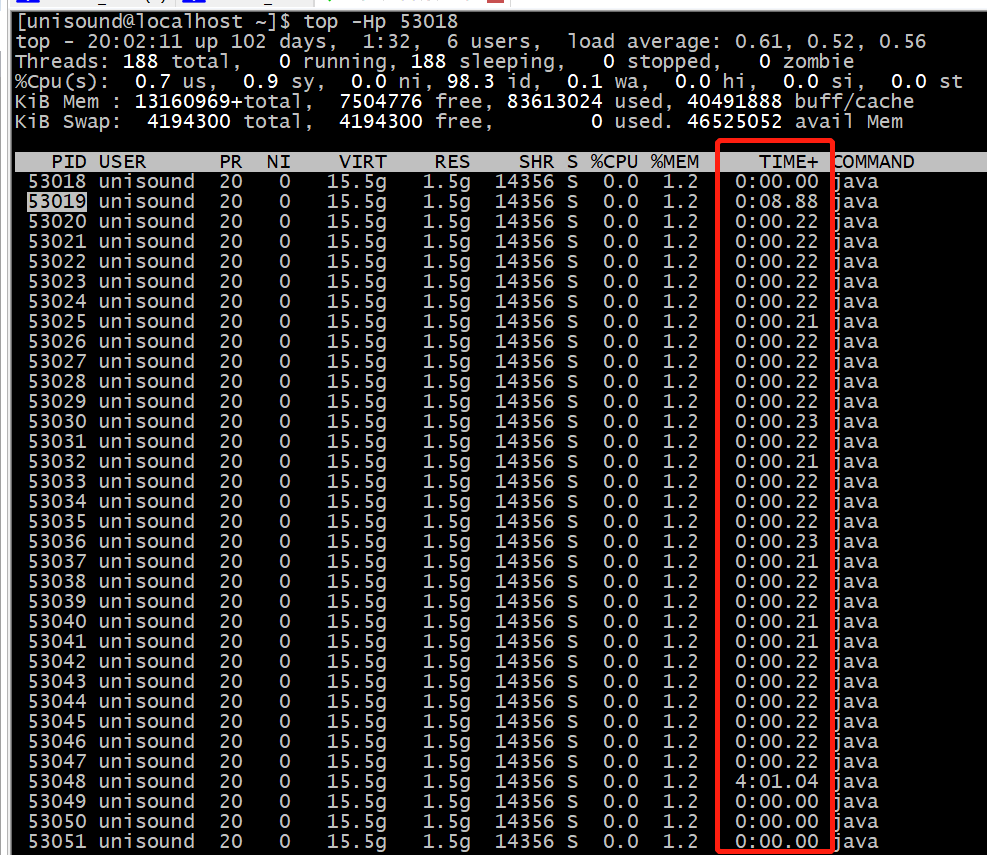
注意： 这个统计也会产生一定的开销（JDK这个接口本身开销比较大），因此会看到as的线程占用一定的百分比，为了降低统计自身的开销带来的影响，可以把采样间隔拉长一些，比如5000毫秒。

#### jvm启动到现在各线程cpu占比

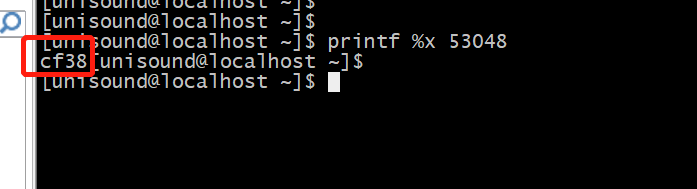
查找jvm启动到当前时间, 最耗cpu线程信息.

##### 方法一

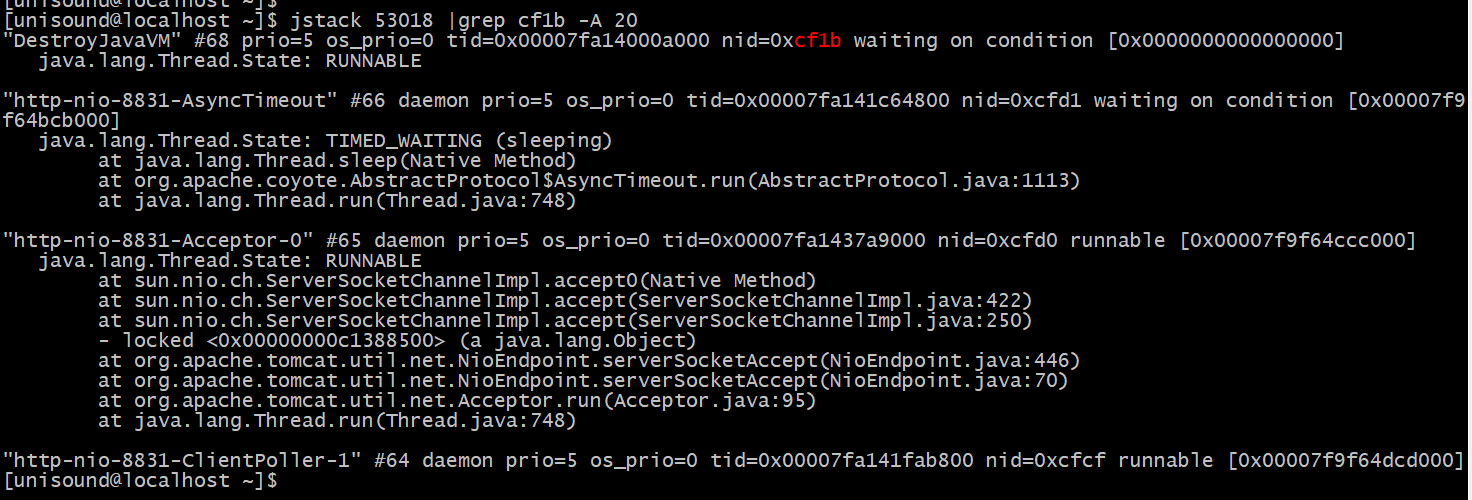
1. 查找到对应java进程pid
2. 使用top –Hp pid 获取该java进程 中所有线程占用cpu时间信息.



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

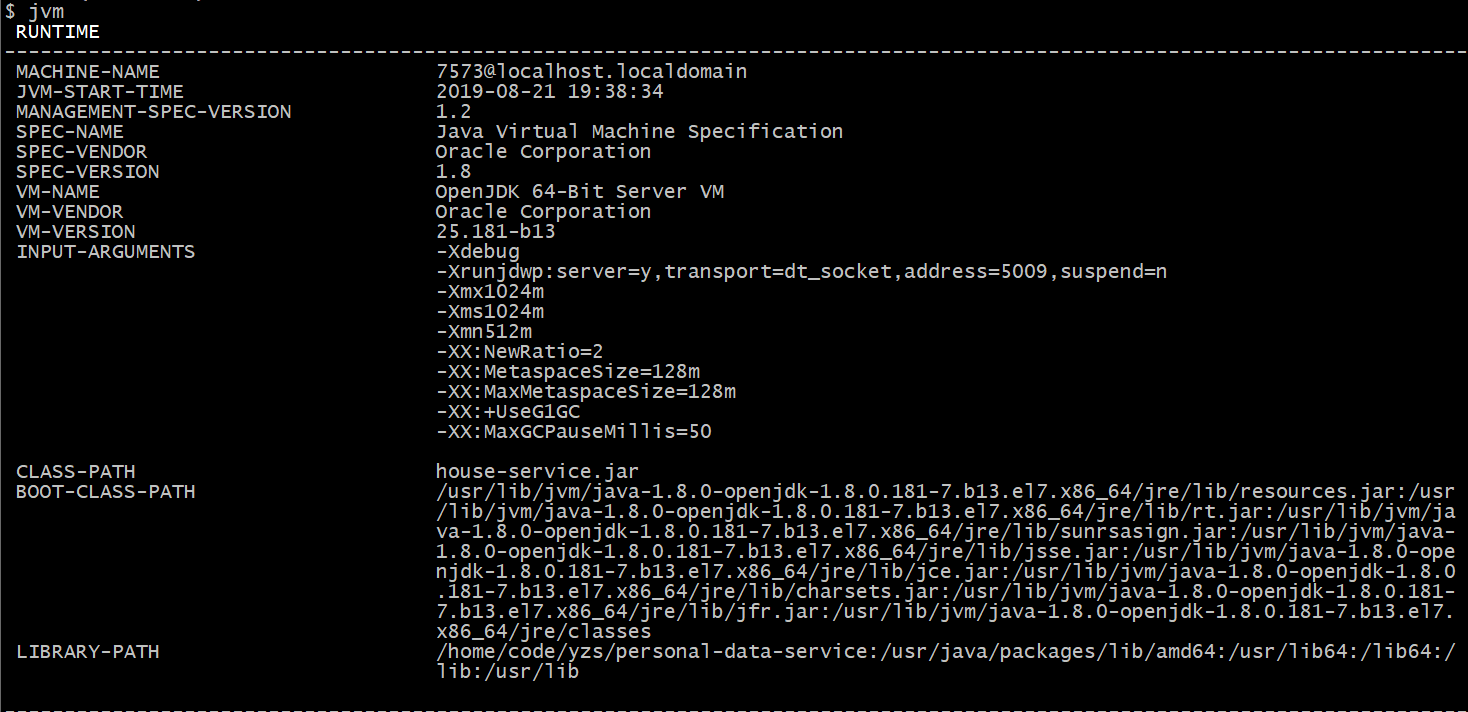
show-busy-java-threads -l

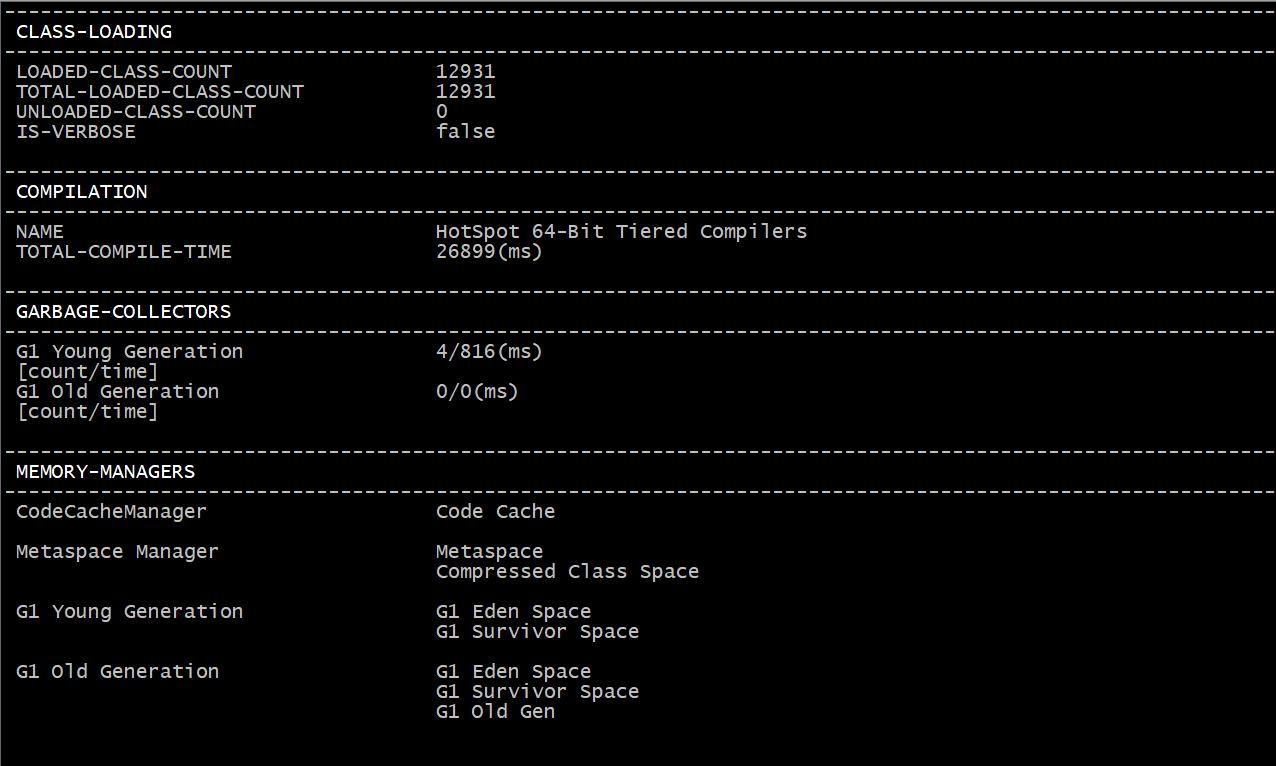
# 帮助信息

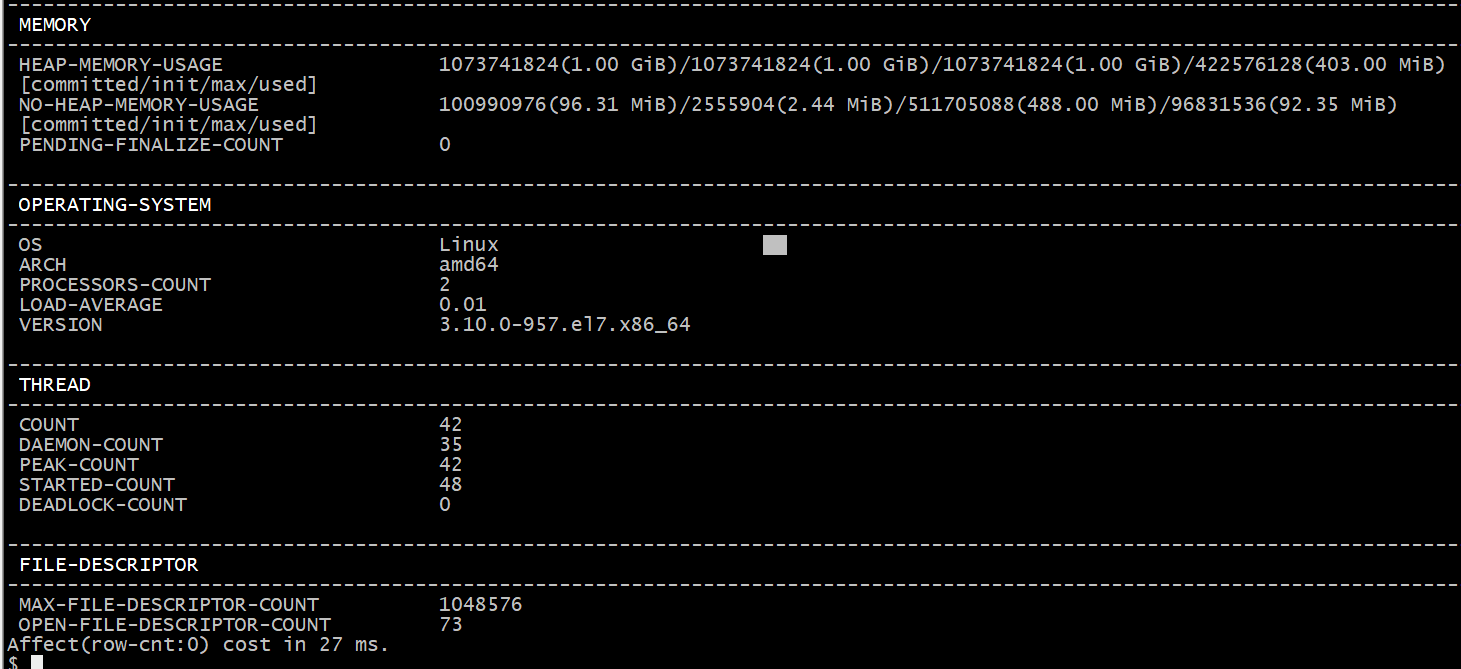
$ show-busy-java-threads -h

### 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

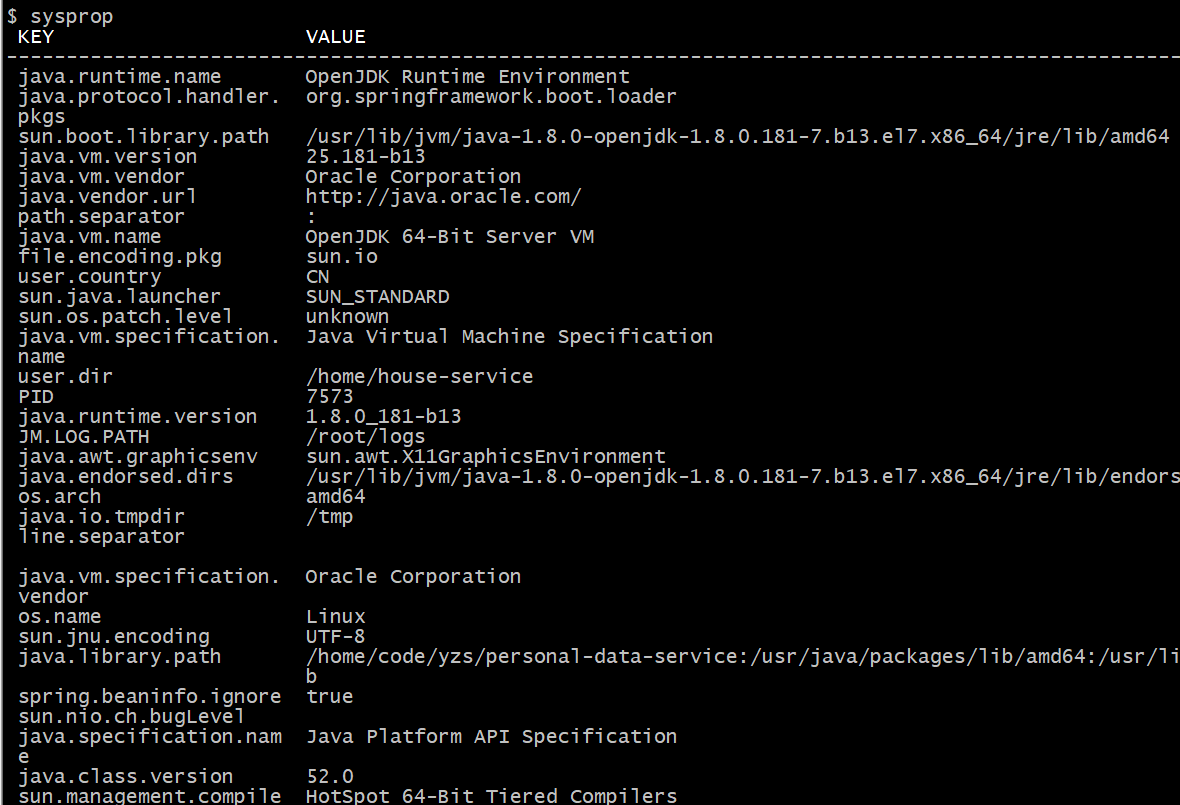
* MAX-FILE-DESCRIPTOR-COUNT：JVM进程最大可以打开的文件描述符数
* OPEN-FILE-DESCRIPTOR-COUNT：JVM当前打开的文件描述符数

### sysprop

查看和修改JVM的系统属性

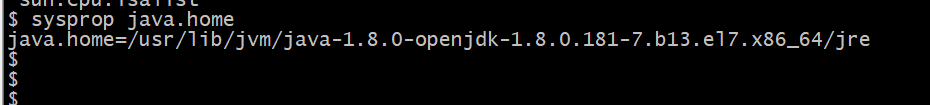
#### 查看所有属性

$ sysprop



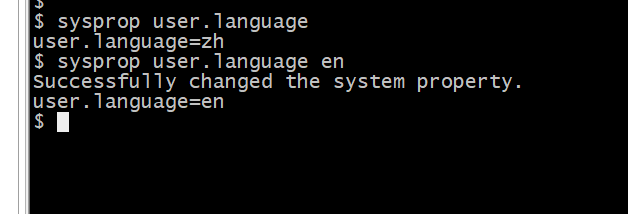
#### 查看单个属性

$ 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

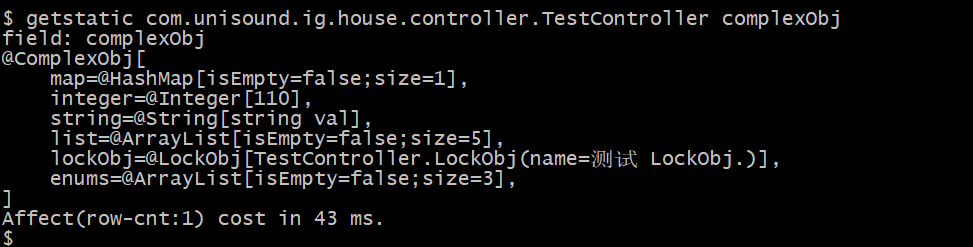
查看类的静态属性.

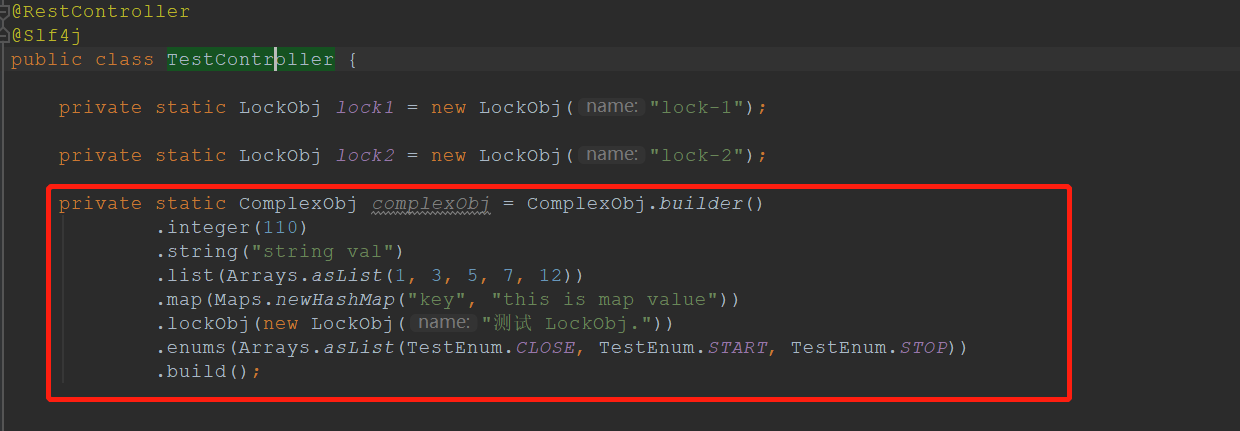
#### 使用方法

getstatic class\_name field\_name

#### 使用示例

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





支持OGNL表达式.

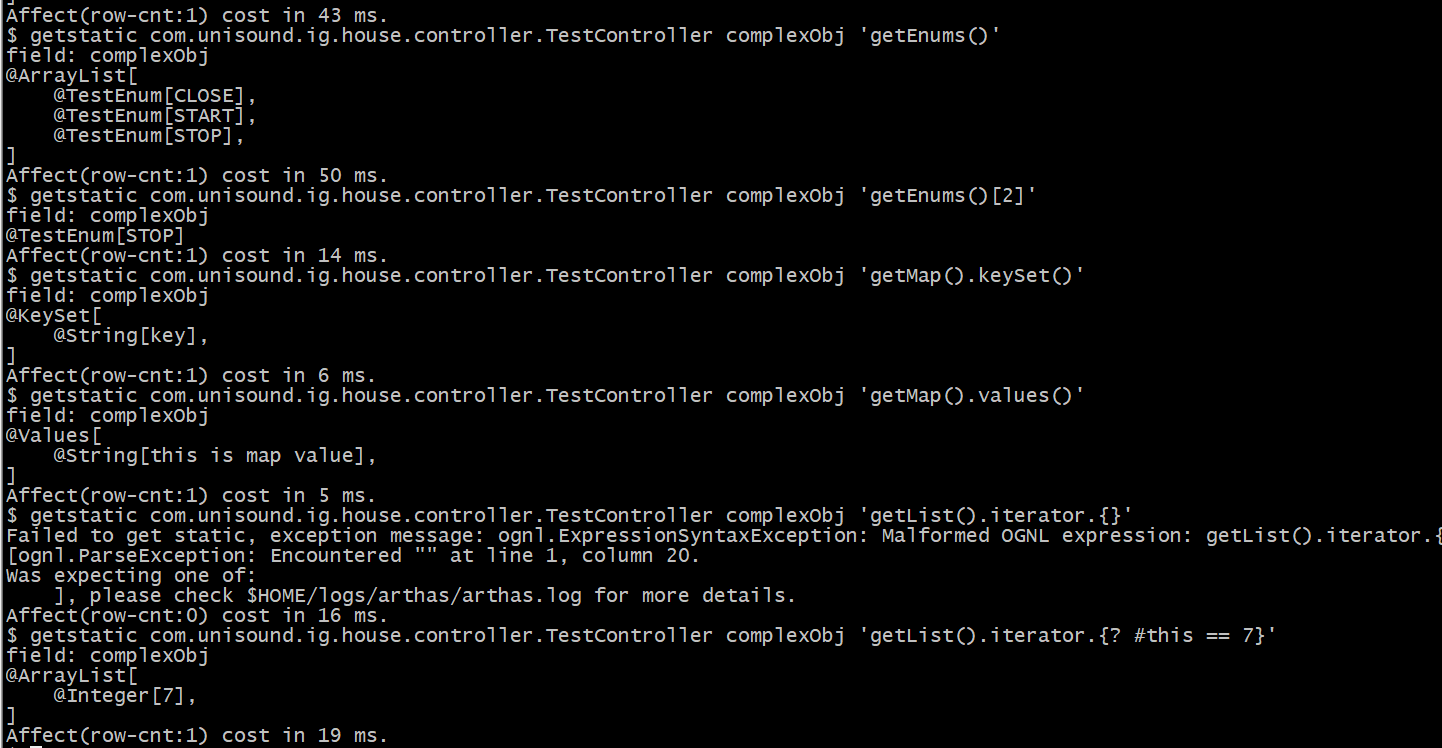
$ getstatic com.unisound.ig.house.controller.TestController complexObj 'getEnums()'

$ 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

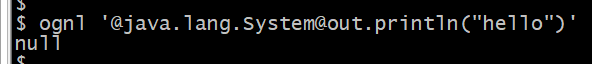
执行ognl表达式, 从3.0.5版本增加.

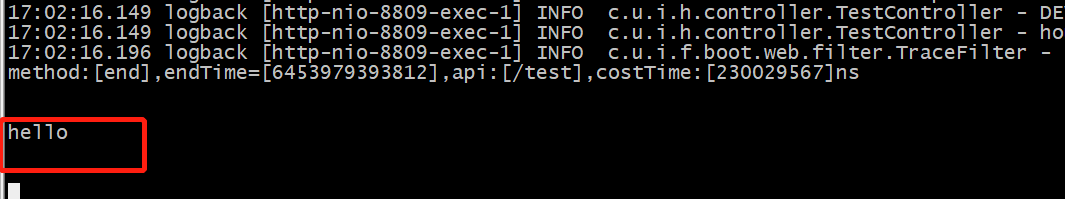
#### 参数说明

| **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

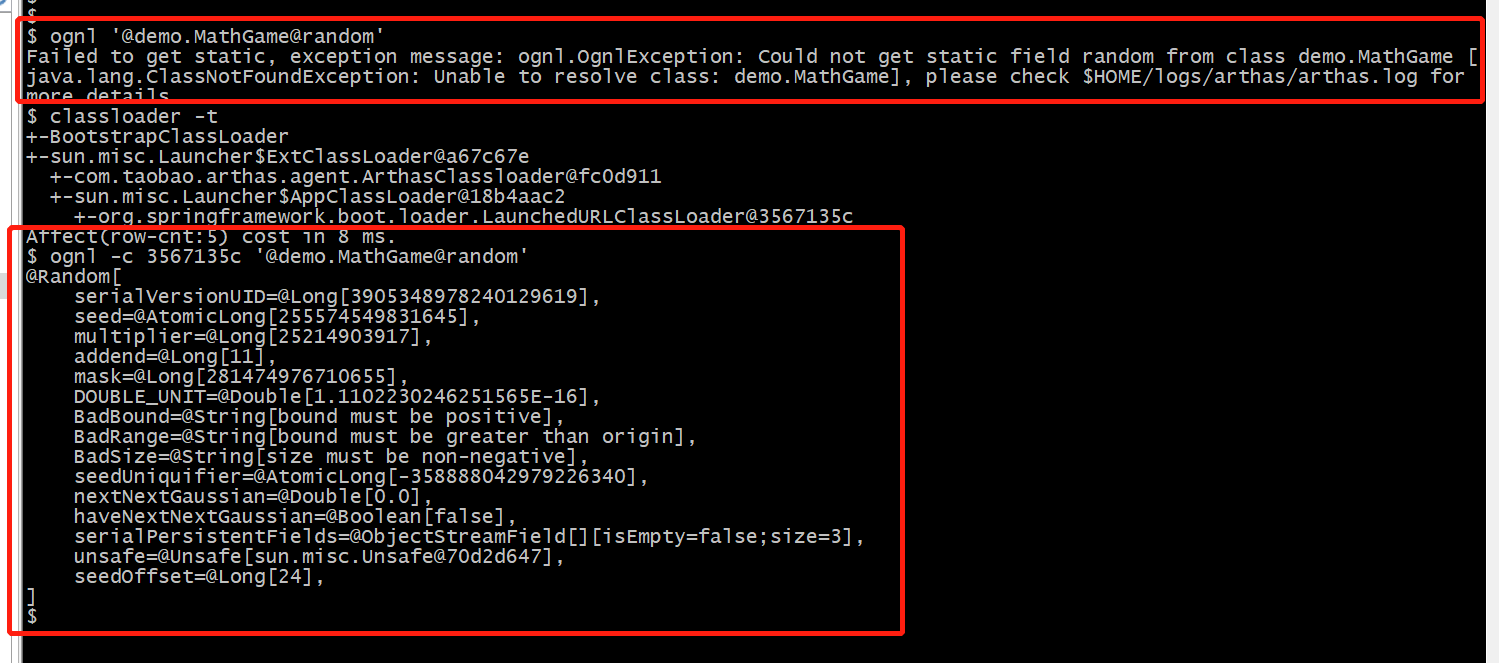
$ ognl '@java.lang.System@out.println("hello")'





#### 获取静态类的静态字段

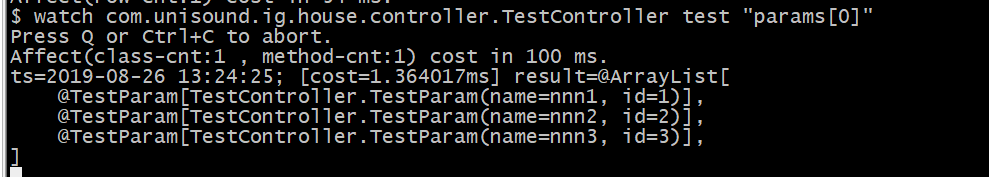
$ ognl -c 3567135c '@demo.MathGame@random' // ognl命令 需要指定 classloader



#### 其他示例

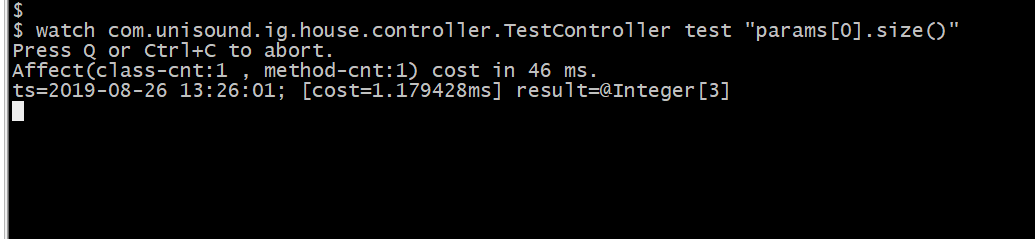
查看第一个参数

$ watch com.unisound.ig.house.controller.TestController test "params[0]"



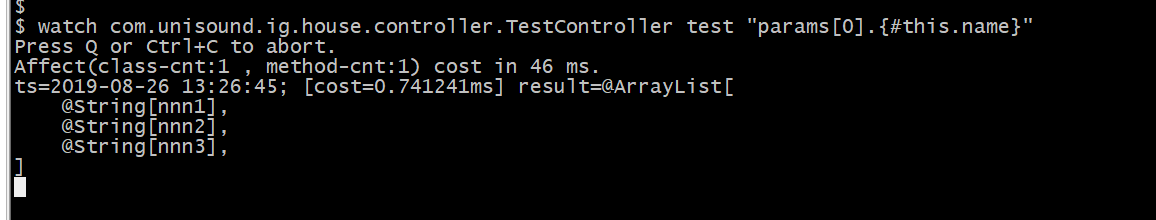
查看第一个参数的size

$ watch com.unisound.ig.house.controller.TestController test "params[0].size()"



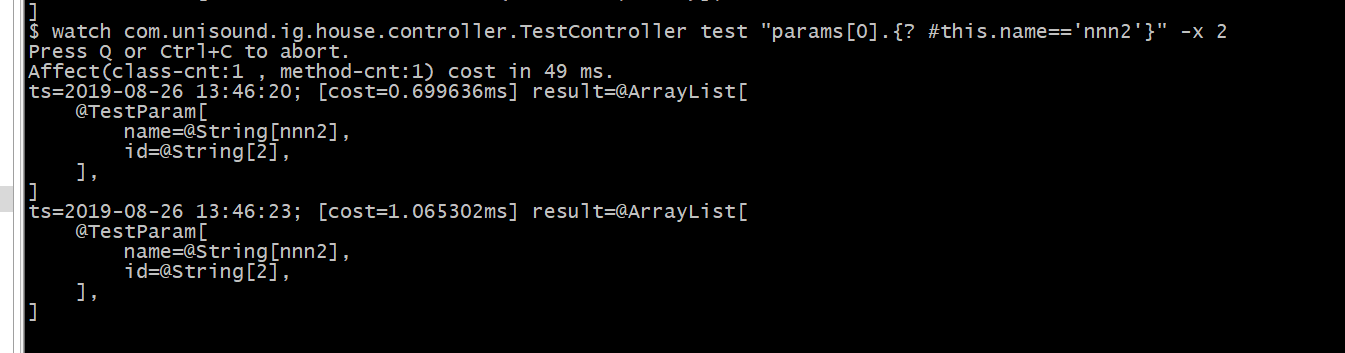
将结果按name属性投影

$ watch com.unisound.ig.house.controller.TestController test "params[0].{#this.name}"



按条件过滤

$watch com.unisound.ig.house.controller.TestController test "params[0].{? #this.name=='nnn2'}" -x 2



### mbean

查看 Mbean 的信息这个命令可以便捷的查看或监控 Mbean 的属性信息。

#### 概念解释

描述一个可管理的资源。是一个java对象，遵循以下一些规则：

* 1. 必须是公用的，非抽象的类
  2. 必须有至少一个公用的[构造器](https://baike.baidu.com/item/%E6%9E%84%E9%80%A0%E5%99%A8/9844976)
  3. 必须实现它自己的相应的MBean接口或者实现javax.management.DynamicMBean接口
  4. 可选的，一个MBean可以实现javax.management.NotificationBroadcaster接口MBean的类型

参考<https://baike.baidu.com/item/MBean/3606231?fr=aladdin>

#### 参数信息

| **参数名称** | **参数说明** |
| --- | --- |
| *name-pattern* | 名称表达式匹配 |
| *attribute-pattern* | 属性名表达式匹配 |
| [m] | 查看元信息 |
| [i:] | 刷新属性值的时间间隔 (ms) |
| [n:] | 刷新属性值的次数 |
| [E] | 开启正则表达式匹配，默认为通配符匹配。仅对属性名有效 |

#### 使用参考

列出所有Mbean的名称

$ mbean

查看Mbean的元信息

$ mbean -m java.lang:type=Threading

mbean的name支持通配符匹配

$ mbean java.lang:type=Th\*

通配符匹配特定的属性字段

$ mbean java.lang:type=Threading \*Count

使用-E命令切换为正则匹配

$ mbean -E java.lang:type=Threading PeakThreadCount|ThreadCount|DaemonThreadCount

使用-i命令实时监控：

$ mbean -i 1000 java.lang:type=Threading \*Count

## class/classloader相关

### sc

查看jvm已加载的类信息.

“Search-Class” 的简写，这个命令能搜索出所有已经加载到 JVM 中的 Class 信息，这个命令支持的参数有 [d]、[E]、[f] 和 [x:]。

#### 参数说明

| **参数名称** | **参数说明** |
| --- | --- |
| *class-pattern* | 类名表达式匹配 |
| *~~method-pattern~~* | ~~方法名表达式匹配~~ |
| [d] | 输出当前类的详细信息，包括这个类所加载的原始文件来源、类的声明、加载的ClassLoader等详细信息。 如果一个类被多个ClassLoader所加载，则会出现多次 |
| [E] | 开启正则表达式匹配，默认为通配符匹配 |
| [f] | 输出当前类的成员变量信息（需要配合参数-d一起使用） |
| [x:] | 指定输出静态变量时属性的遍历深度，默认为 0，即直接使用 toString 输出 |

class-pattern支持全限定名，如com.taobao.test.AAA，也支持com/taobao/test/AAA这样的格式，这样，我们从异常堆栈里面把类名拷贝过来的时候，不需要在手动把/替换为.啦。

sc 默认开启了子类匹配功能，也就是说所有当前类的子类也会被搜索出来，想要精确的匹配，请打开options disable-sub-class true开关

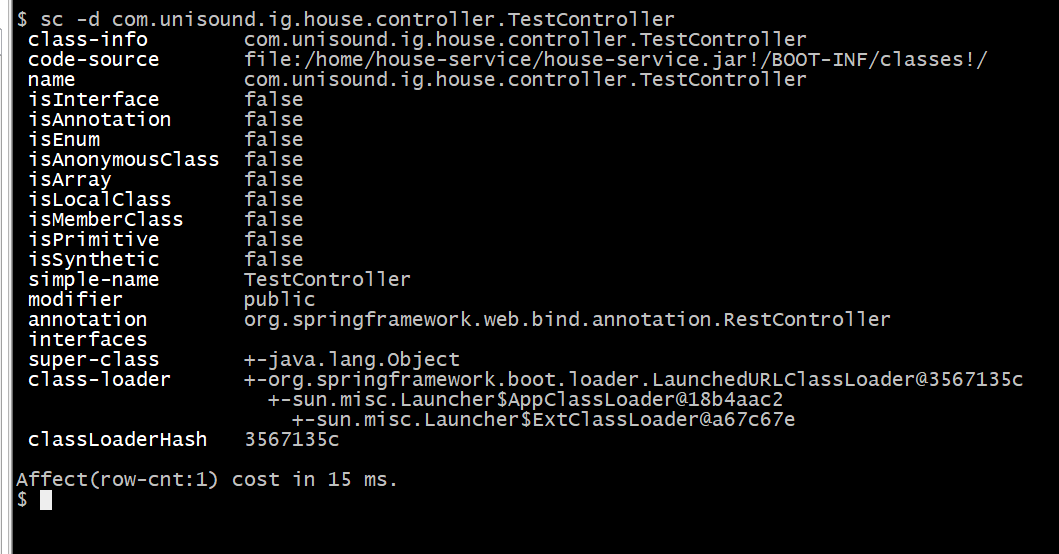
#### 使用参考

##### 模糊搜索

$ sc com.unisound.\*

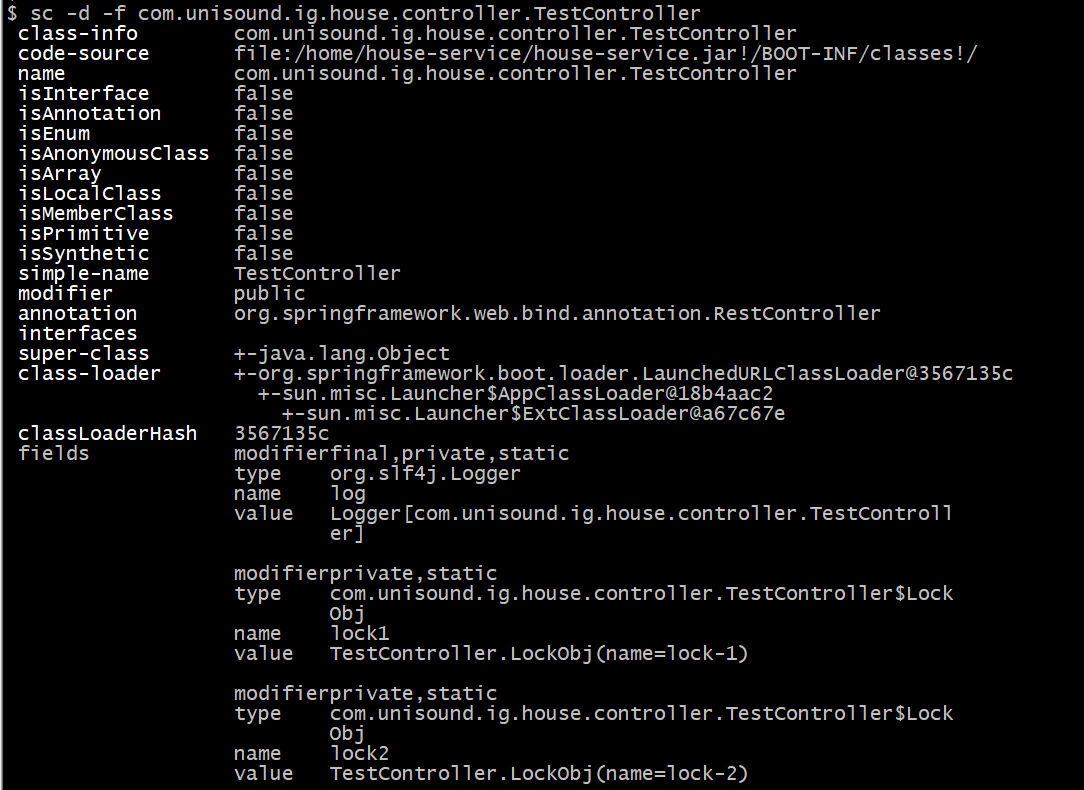
##### 打印类的详细信息

$ sc -d com.unisound.ig.house.controller.TestController



##### 打印出类的Field信息

$ sc -d -f com.unisound.ig.house.controller.TestController



### sm

查看已加载的类的方法信息.

“Search-Method” 的简写，这个命令能搜索出所有已经加载了 Class 信息的方法信息。sm 命令只能看到由当前类所声明 (declaring) 的方法，父类则无法看到。

#### 参数说明

| **参数名称** | **参数说明** |
| --- | --- |
| *class-pattern* | 类名表达式匹配 |
| *method-pattern* | 方法名表达式匹配 |
| [d] | 展示每个方法的详细信息 |
| [E] | 开启正则表达式匹配，默认为通配符匹配 |

#### 使用参考

查看类 所有的方法信息.

$ sm com.unisound.ig.house.controller.TestController

展示每个方法的详细信息

$ sm -d com.unisound.ig.house.controller.TestController

展示一个方法(costCpu)的详细信息

$ sm -d com.unisound.ig.house.controller.TestController costCpu

### jad

反编译指定已加载类的源码

jad 命令将 JVM 中实际运行的 class 的 byte code 反编译成 java 代码，便于你理解业务逻辑；

在 Arthas Console 上，反编译出来的源码是带语法高亮的，阅读更方便当然，反编译出来的 java 代码可能会存在语法错误，但不影响你进行阅读理解

#### 参数说明

| **参数名称** | **参数说明** |
| --- | --- |
| *class-pattern* | 类名表达式匹配 |
| [c:] | 类所属 ClassLoader 的 hashcode |
| [E] | 开启正则表达式匹配，默认为通配符匹配 |

#### 使用参考

##### 反编译一个类

$ jad com.unisound.ig.house.controller.TestController

##### 反编译只显示源代码

默认情况下，反编译结果里会带有ClassLoader信息，通过--source-only选项，可以只打印源代码。方便和[mc](https://alibaba.github.io/arthas/mc.html)/[redefine](https://alibaba.github.io/arthas/redefine.html)命令结合使用。

$ jad --source-only com.unisound.ig.house.controller.TestController

##### 反编译指定的方法

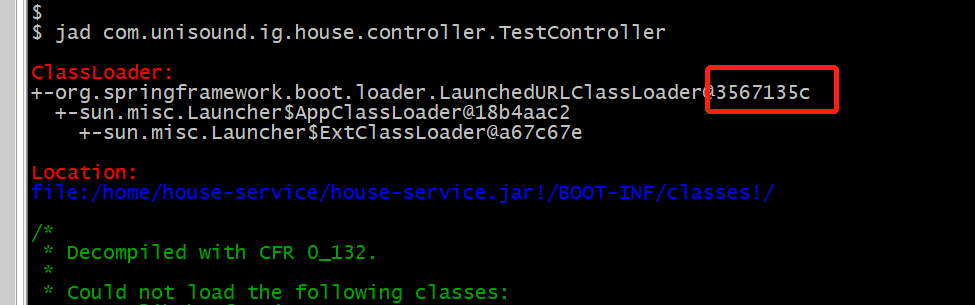
$ jad com.unisound.ig.house.controller.TestController init



##### 反编译时指定ClassLoader

当有多个 ClassLoader 都加载了这个类时，jad 命令会输出对应 ClassLoader 实例的 hashcode，然后你只需要重新执行 jad 命令，并使用参数 -c 就可以反编译指定 ClassLoader 加载的那个类了；

$ jad -c 3567135c com.unisound.ig.house.controller.TestController



### mc

Memory Compiler/内存编译器，编译.java文件生成.class。

编译一个java文件

$ mc /tmp/ArthasDemo.java

可以通过-c参数指定classloader：

$ mc -c 327a647b /tmp/ArthasDemo.java

可以通过-d命令指定输出目录

$ mc -d /tmp/output /tmp/TestA.java /tmp/TestB.java

编译生成.class文件之后，可以结合redefine命令实现热更新代码。注意，mc命令有可能失败。如果编译失败可以在本地编译好.class文件，再上传到服务器。具体参考redefine命令说明。

### redefine

加载外部的.class文件，redefine jvm已加载的类。

参考: [Instrumentation#redefineClasses](https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/Instrumentation.html#redefineClasses-java.lang.instrument.ClassDefinition...-)

注意: redefine后的原来的类不能恢复，redefine有可能失败（比如增加了新的field），参考jdk本身的文档。

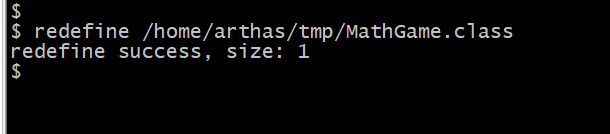
#### 参数说明

| **参数名称** | **参数说明** |
| --- | --- |
| [c:] | ClassLoader的hashcode |
| [p:] | 外部的.class文件的完整路径，支持多个 |

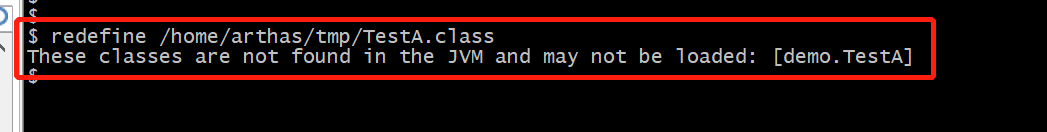
#### 使用参考

##### redefine一个class

$ redefine /home/arthas/tmp/MathGame.class



注意: 只能redefine当前jvm存在并且已加载的类.



##### 指定classLoader

$ redefine -c 3567135c /home/arthas/tmp/MathGame.class

#### 结合jad/mc命令使用

jad --source-only com.example.demo.arthas.user.UserController > /tmp/UserController.java

mc /tmp/UserController.java -d /tmp

redefine /tmp/com/example/demo/arthas/user/UserController.class

* jad命令反编译，然后可以用其它编译器，比如vim来修改源码
* mc命令来内存编译修改过的代码
* 用redefine命令加载新的字节码

#### 上传 .class 文件到服务器的技巧

使用mc命令来编译jad的反编译的代码有可能失败。可以在本地修改代码，编译好后再上传到服务器上。有的服务器不允许直接上传文件，可以使用base64命令来绕过。[mc](https://alibaba.github.io/arthas/mc.html)——内存编绎器，内存编绎.java文件为.class文件

1. 在本地先转换.class文件为base64，再保存为result.txt

base64 < Test.class > result.txt

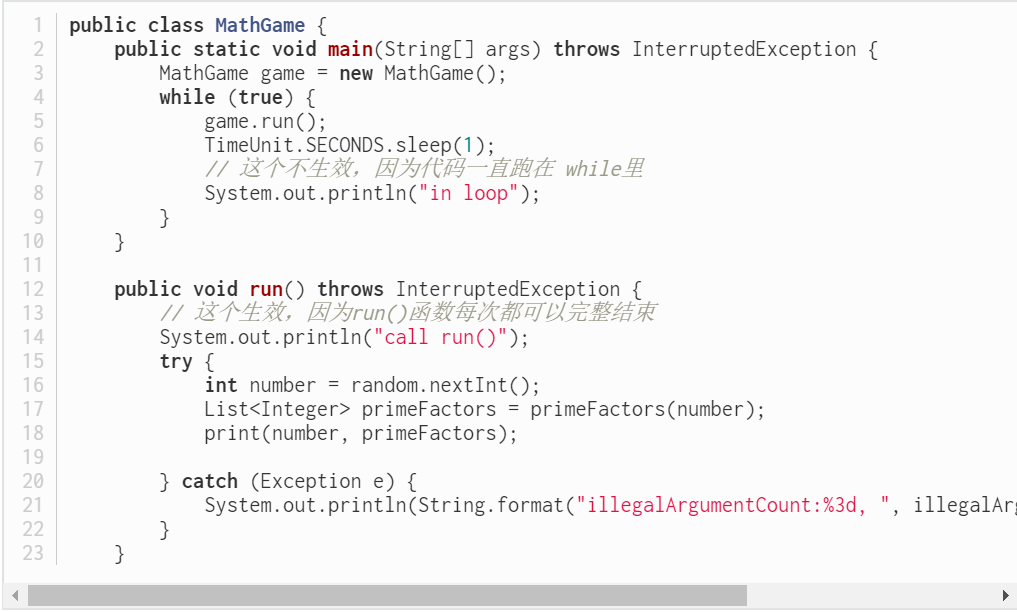
1. 到服务器上，新建并编辑result.txt，复制本地的内容，粘贴再保存
2. 把服务器上的 result.txt还原为.class

base64 -d < result.txt > Test.class

1. 用md5命令计算哈希值，校验是否一致

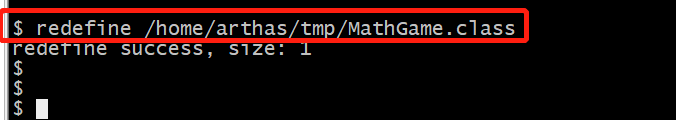
#### redefine的限制

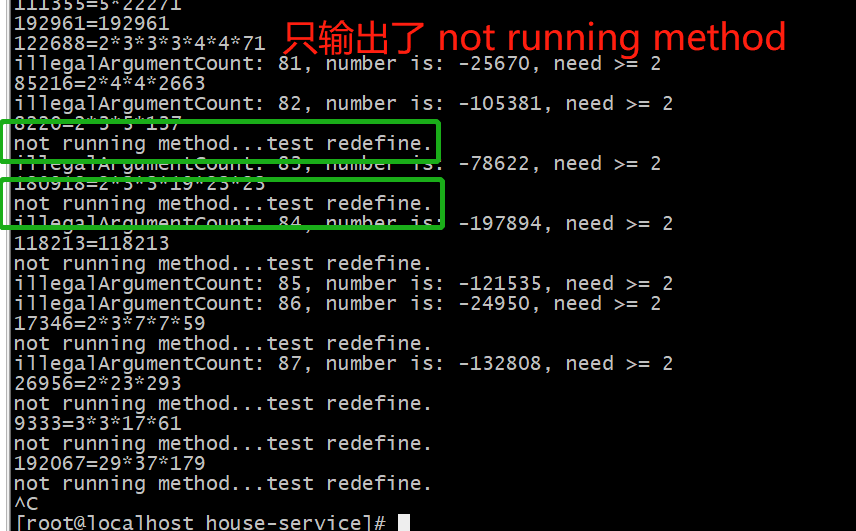
1. 不允许新增field / method
2. 正在跑的函数, 没有退出不能生效. 比如下面新增加的System.out.println，只有run()函数里的会生效



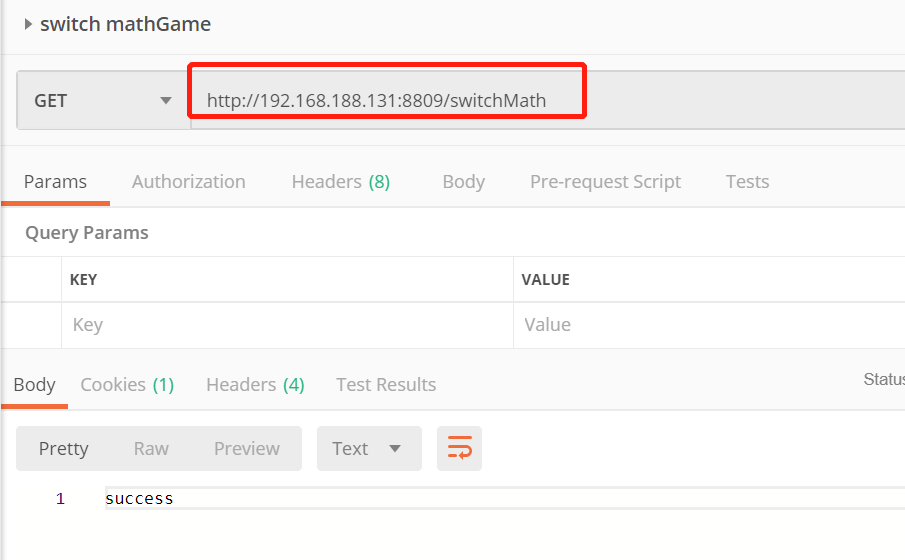


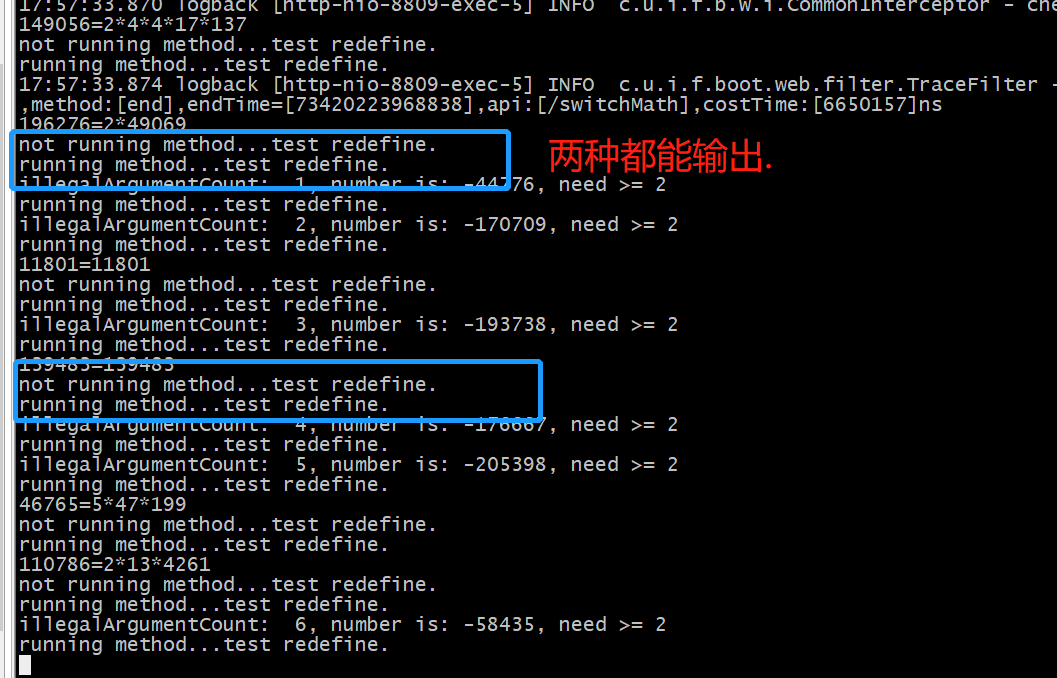
$ redefine /home/arthas/tmp/MathGame.class





这里 重新触发 mathGame方法. 看效果.





### dump

dump 已加载类的 bytecode 到特定目录

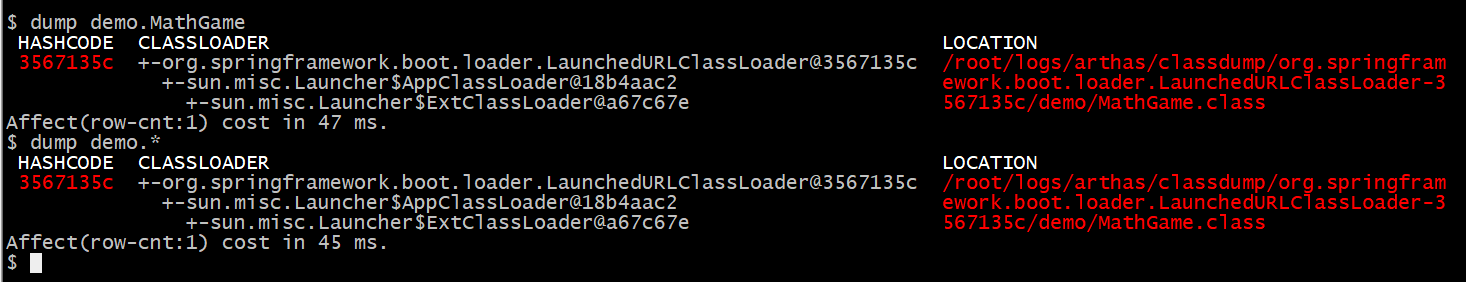
#### 参数说明

| **参数名称** | **参数说明** |
| --- | --- |
| *class-pattern* | 类名表达式匹配 |
| [c:] | 类所属 ClassLoader 的 hashcode |
| [E] | 开启正则表达式匹配，默认为通配符匹配 |

#### 使用参考

$ dump demo.MathGame

$ dump demo.\*



### classloader

查看classloader的继承树，urls，类加载信息

classloader 命令将 JVM 中所有的classloader的信息统计出来，并可以展示继承树，urls等。可以让指定的classloader去getResources，打印出所有查找到的resources的url。对于ResourceNotFoundException比较有用。

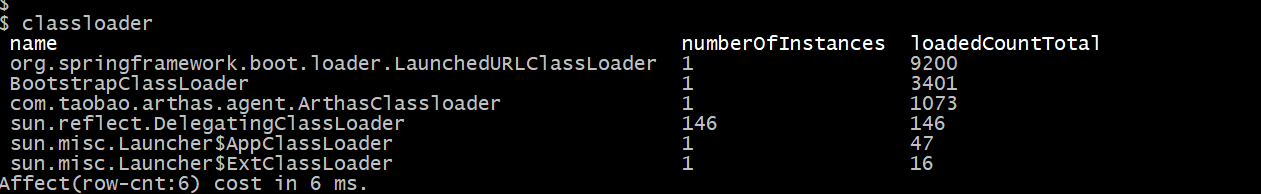
#### 参数说明

| **参数名称** | **参数说明** |
| --- | --- |
| [l] | 按类加载实例进行统计 |
| [t] | 打印所有ClassLoader的继承树 |
| [a] | 列出所有ClassLoader加载的类，请谨慎使用 |
| [c:] | ClassLoader的hashcode |
| [c: r:] | 用ClassLoader去查找resource |
| [c: load:] | 用ClassLoader去加载指定的类 |

#### 使用参考

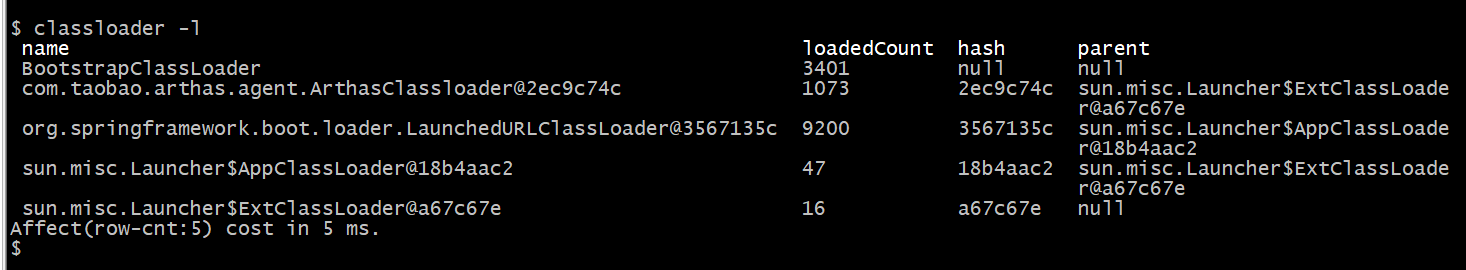
##### 按类加载类型查看统计信息

$ classloader



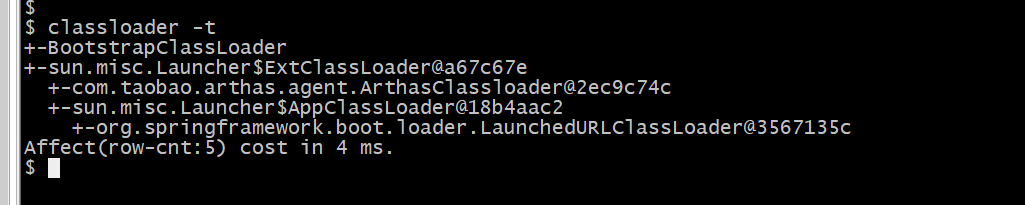
##### 按类加载实例查看统计信息

$ classloader -l



##### 查看ClassLoader的继承树

$ classloader –t



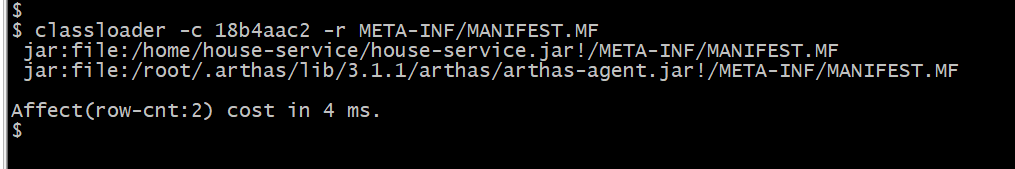
##### 查看URLClassLoader实际的urls

$ classloader -c 2ec9c74c



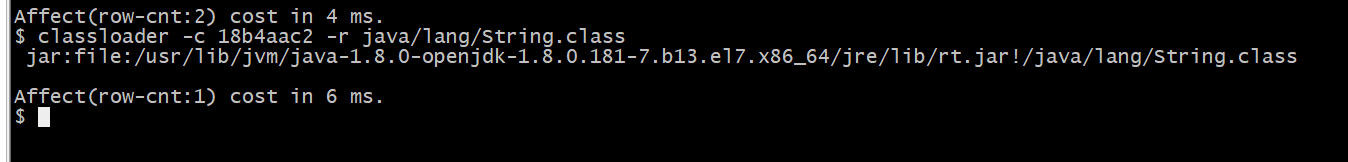
##### 使用classLoader去查找resource

$ classloader -c 18b4aac2 -r META-INF/MANIFEST.MF



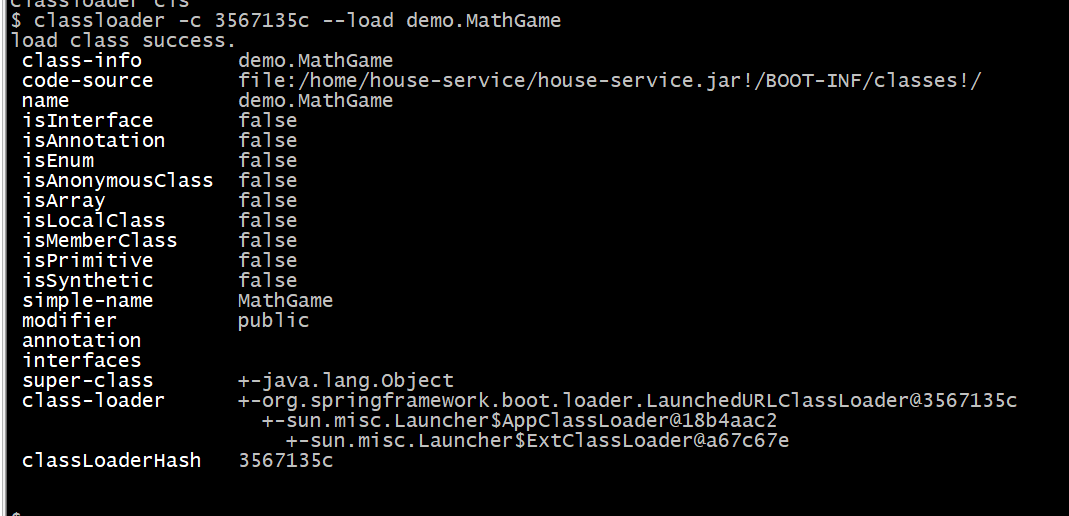
##### 尝试查找类的class文件

$ classloader -c 18b4aac2 -r java/lang/String.class



##### 使用ClassLoader去加载类

$ classloader -c 3567135c --load demo.MathGame



## 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

方法执行监控.

对匹配 class-pattern／method-pattern的类、方法的调用进行监控。

monitor 命令是一个非实时返回命令.

实时返回命令是输入之后立即返回，而非实时返回的命令，则是不断的等待目标 Java 进程返回信息，直到用户输入 Ctrl+C 为止。

#### 监控的维度说明

| **监控项** | **说明** |
| --- | --- |
| timestamp | 时间戳 |
| class | Java类 |
| method | 方法（构造方法、普通方法） |
| total | 调用次数 |
| success | 成功次数 |
| fail | 失败次数 |
| rt | 平均RT |
| fail-rate | 失败率 |

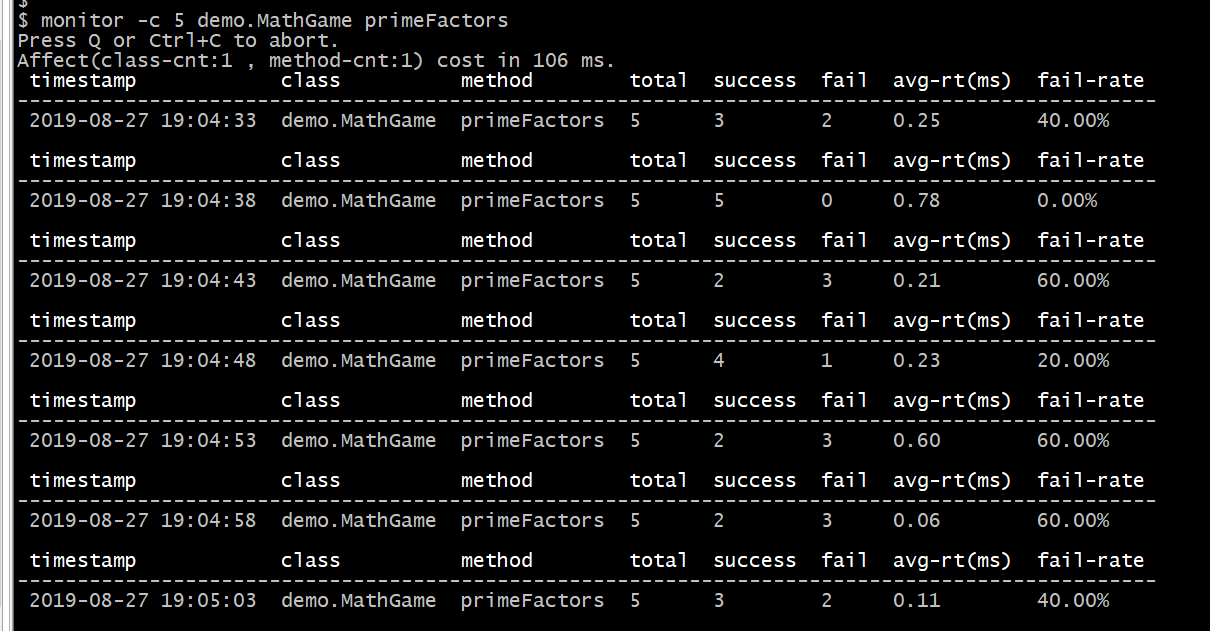
#### 参数说明

方法拥有一个命名参数 [c:]，意思是统计周期（cycle of output），拥有一个整型的参数值

| **参数名称** | **参数说明** |
| --- | --- |
| *class-pattern* | 类名表达式匹配 |
| *method-pattern* | 方法名表达式匹配 |
| [E] | 开启正则表达式匹配，默认为通配符匹配 |
| [c:] | 统计周期，默认值为120秒 |

#### 使用参考

$ monitor -c 5 demo.MathGame primeFactors



### watch

方法执行数据观测

让你能方便的观察到指定方法的调用情况。能观察到的范围为：返回值、抛出异常、入参，通过编写OGNL 表达式进行对应变量的查看。

#### 参数说明

watch 的参数比较多，主要是因为它能在 4 个不同的场景观察对象

| **参数名称** | **参数说明** |
| --- | --- |
| *class-pattern* | 类名表达式匹配 |
| *method-pattern* | 方法名表达式匹配 |
| *express* | 观察表达式 |
| *condition-express* | 条件表达式 |
| [b] | 在**方法调用之前**观察 |
| [e] | 在**方法异常之后**观察 |
| [s] | 在**方法返回之后**观察 |
| [f] | 在**方法结束之后**(正常返回和异常返回)观察 |
| [E] | 开启正则表达式匹配，默认为通配符匹配 |
| [x:] | 指定输出结果的属性遍历深度，默认为 1 |

这里重点要说明的是观察表达式，观察表达式的构成主要由ognl表达式组成，所以你可以这样写"{params,returnObj}"，只要是一个合法的ognl表达式，都能被正常支持。

观察的维度也比较多，主要体现在参数 advice 的数据结构上。Advice 参数最主要是封装了通知节点的所有信息。请参考[表达式核心变量](https://alibaba.github.io/arthas/advice-class.html)中关于该节点的描述。

特殊用法请参考:

<https://github.com/alibaba/arthas/issues/71>

OGNL表达式官网:

<https://commons.apache.org/proper/commons-ognl/language-guide.html>

#### 特别说明

1. watch 命令定义了4个观察事件点，即 -b 方法调用前，-e 方法异常后，-s 方法返回后，-f 方法结束后
2. 4个观察事件点 -b、-e、-s 默认关闭，-f 默认打开，当指定观察点被打开后，在相应事件点会对观察表达式进行求值并输出.
3. 这里要注意方法入参和方法出参的区别，有可能在中间被修改导致前后不一致，除了 -b 事件点 params 代表方法入参外，其余事件都代表方法出参.
4. 当使用 -b 时，由于观察事件点是在方法调用前，此时返回值或异常均不存在

#### 使用说明

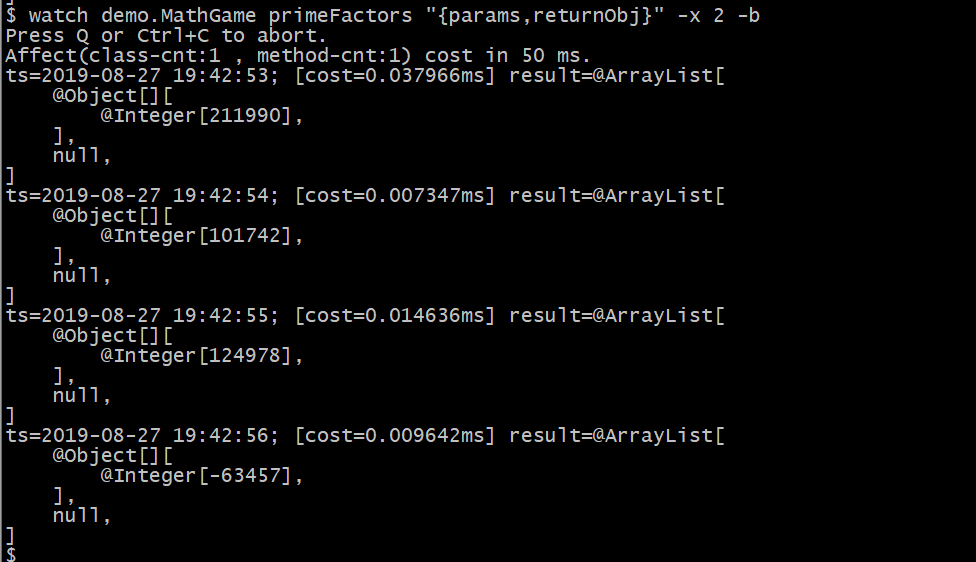
##### 观察方法出参和返回值

$ watch demo.MathGame primeFactors "{params,returnObj}" -x 2



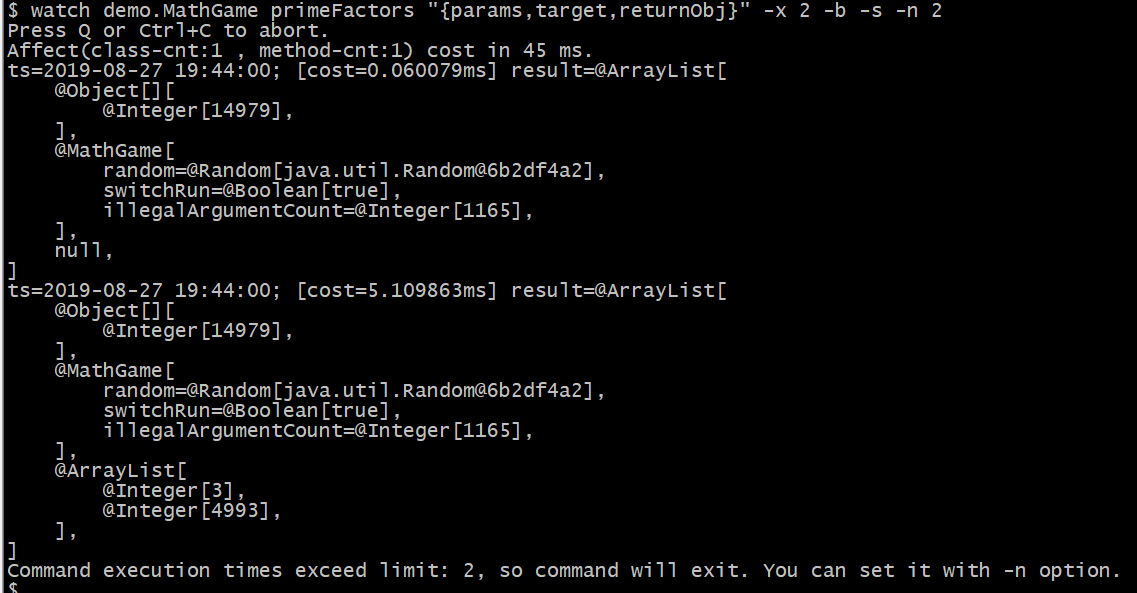
##### 观察方法入参

$ watch demo.MathGame primeFactors "{params,returnObj}" -x 2 –b

对比前一个例子，返回值为空（事件点为方法执行前，因此获取不到返回值）

##### 同时观察方法调用前和方法返回后

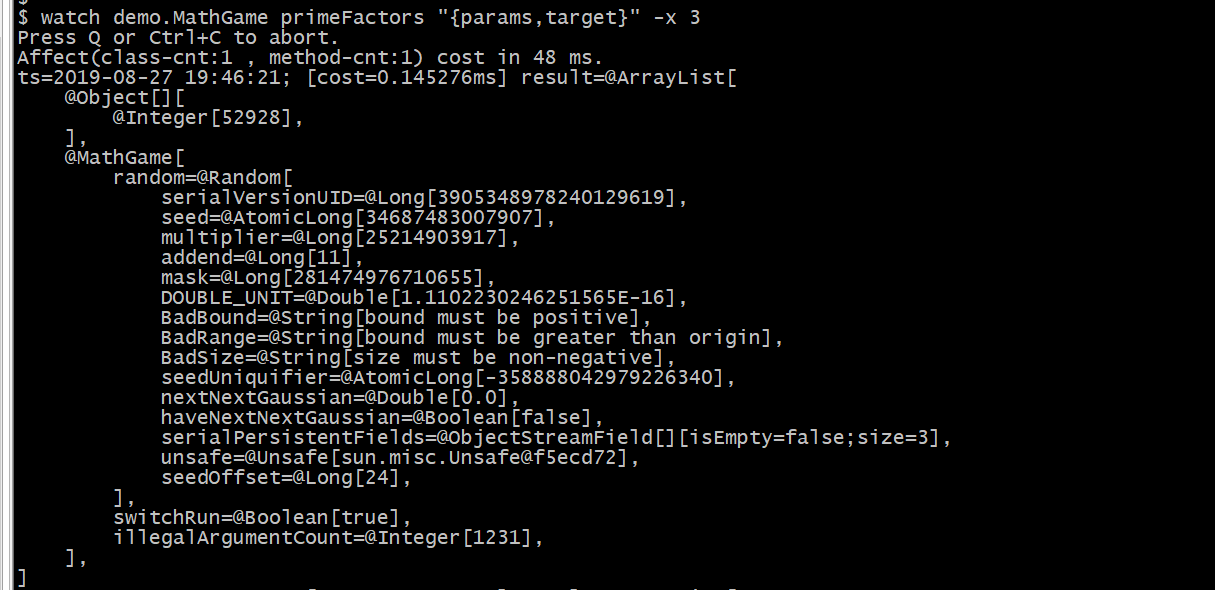
$ watch demo.MathGame primeFactors "{params,target,returnObj}" -x 2 -b -s -n 2



1. 参数里-n 2，表示只执行两次
2. 这里输出结果中，第一次输出的是方法调用前的观察表达式的结果，第二次输出的是方法返回后的表达式的结果
3. 结果的输出顺序和事件发生的先后顺序一致，和命令中 -s -b 的顺序无关

##### 调整-x的值，观察具体的方法参数值

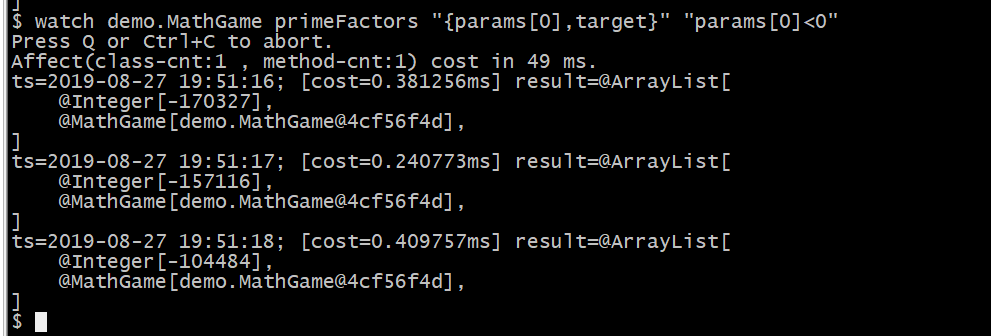
$ watch demo.MathGame primeFactors "{params,target}" -x 3



-x表示遍历深度，可以调整来打印具体的参数和结果内容，默认值是1。

##### 条件表达式的例子

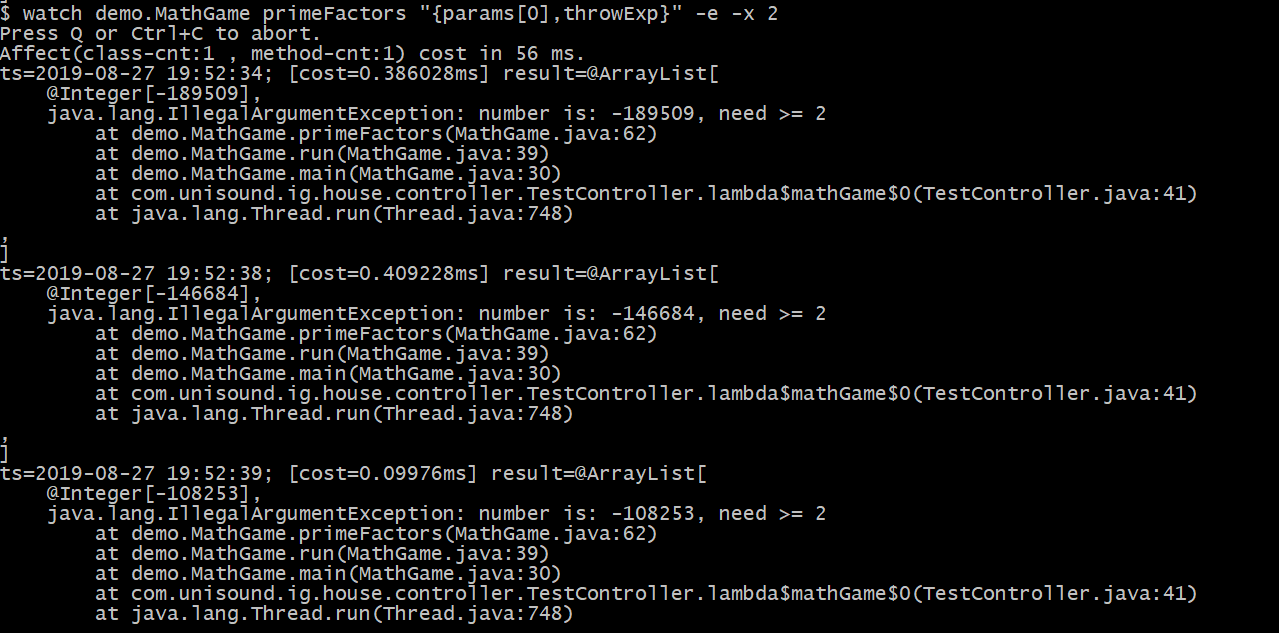
$ watch demo.MathGame primeFactors "{params[0],target}" "params[0]<0"



只有满足条件的调用，才会有响应

##### 观察异常信息的例子

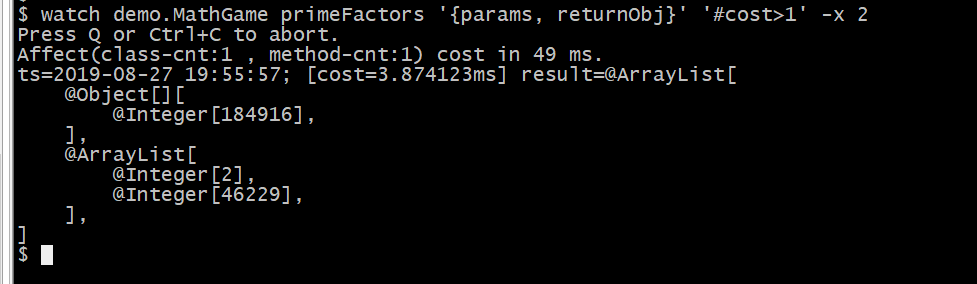
$ watch demo.MathGame primeFactors "{params[0],throwExp}" -e -x 2



1. -e表示抛出异常时才触发
2. express中，表示异常信息的变量是throwExp

##### 按照耗时进行过滤

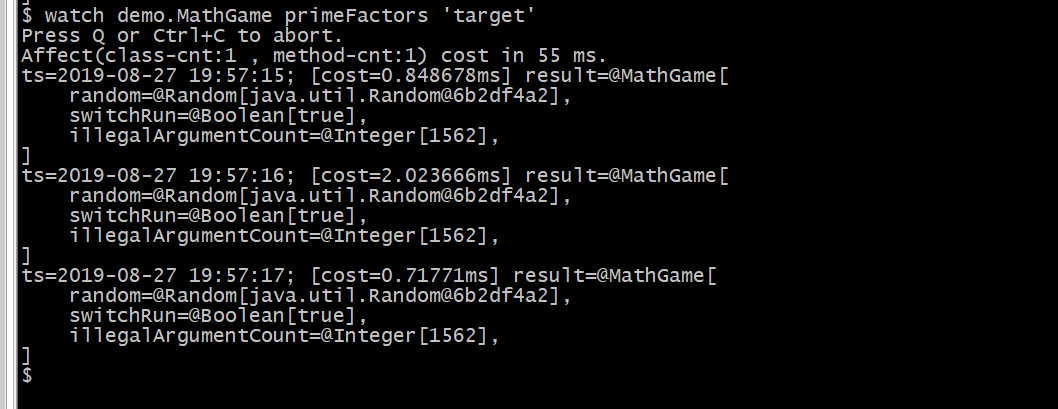
$ watch demo.MathGame primeFactors '{params, returnObj}' '#cost>200' -x 2



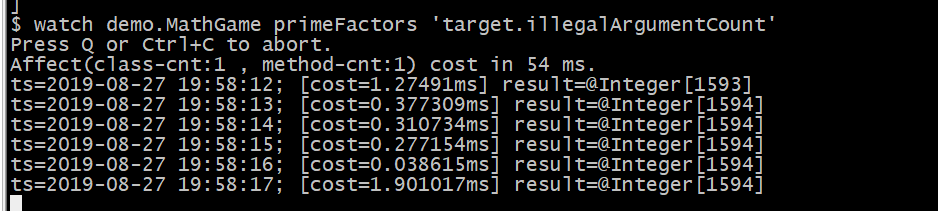
#cost>200(单位是ms)表示只有当耗时大于200ms时才会输出，过滤掉执行时间小于200ms的调用

##### 观察当前对象中的属性

$ watch demo.MathGame primeFactors 'target'



##### 使用target.field\_name访问当前对象的某个属性



### 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.

trace 命令能主动搜索 class-pattern／method-pattern 对应的方法调用路径，渲染和统计整个调用链路上的所有性能开销和追踪调用链路。

#### 参数说明

| **参数名称** | **参数说明** |
| --- | --- |
| *class-pattern* | 类名表达式匹配 |
| *method-pattern* | 方法名表达式匹配 |
| *condition-express* | 条件表达式 |
| [E] | 开启正则表达式匹配，默认为通配符匹配 |
| [n:] | 命令执行次数 |
| #cost | 方法执行耗时 |

这里重点要说明的是观察表达式，观察表达式的构成主要由ognl表达式组成，所以你可以这样写"{params,returnObj}"，只要是一个合法的ognl表达式，都能被正常支持。

很多时候我们只想看到某个方法的rt大于某个时间之后的trace结果，现在Arthas可以按照方法执行的耗时来进行过滤了，例如trace \*StringUtils isBlank '#cost>100'表示当执行时间超过100ms的时候，才会输出trace的结果。

watch/stack/trace这个三个命令都支持 #cost

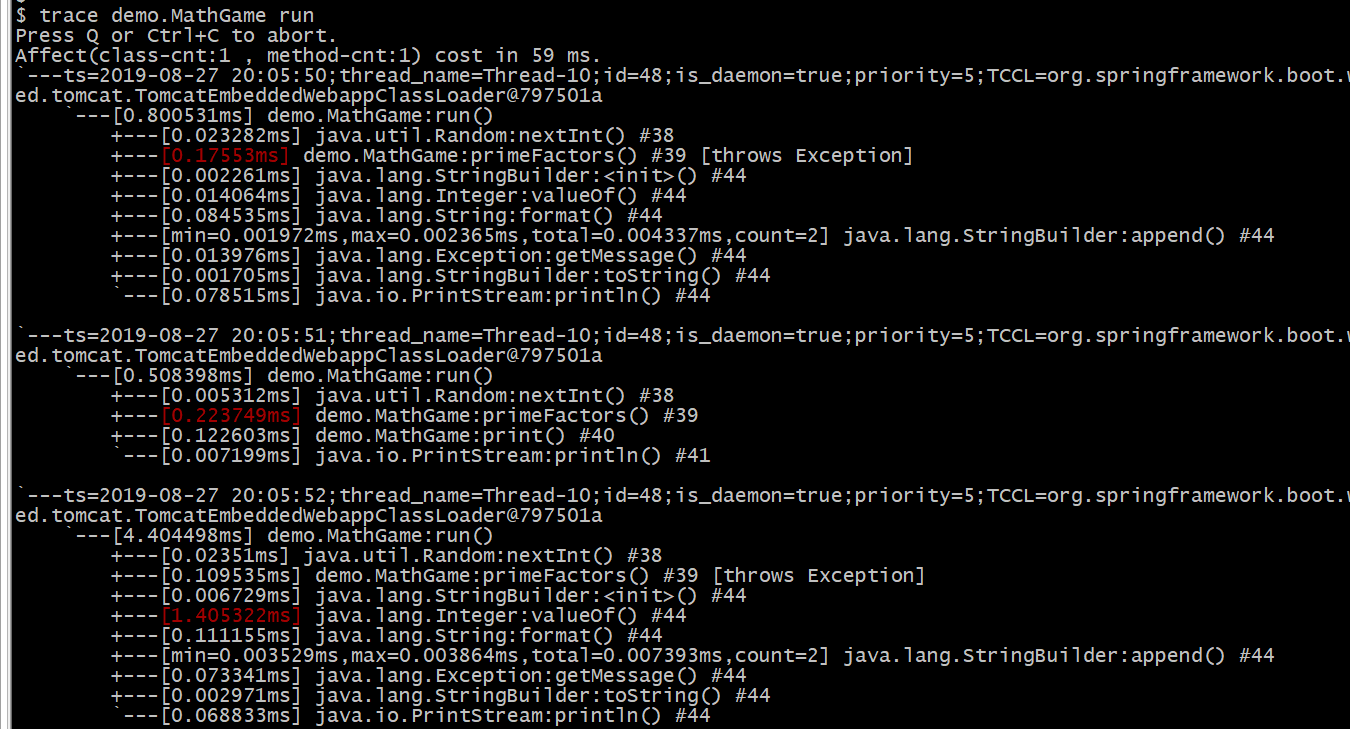
#### 注意事项

trace 能方便的帮助你定位和发现因 RT 高而导致的性能问题缺陷，但其每次只能跟踪一级方法的调用链路。

#### 使用参考

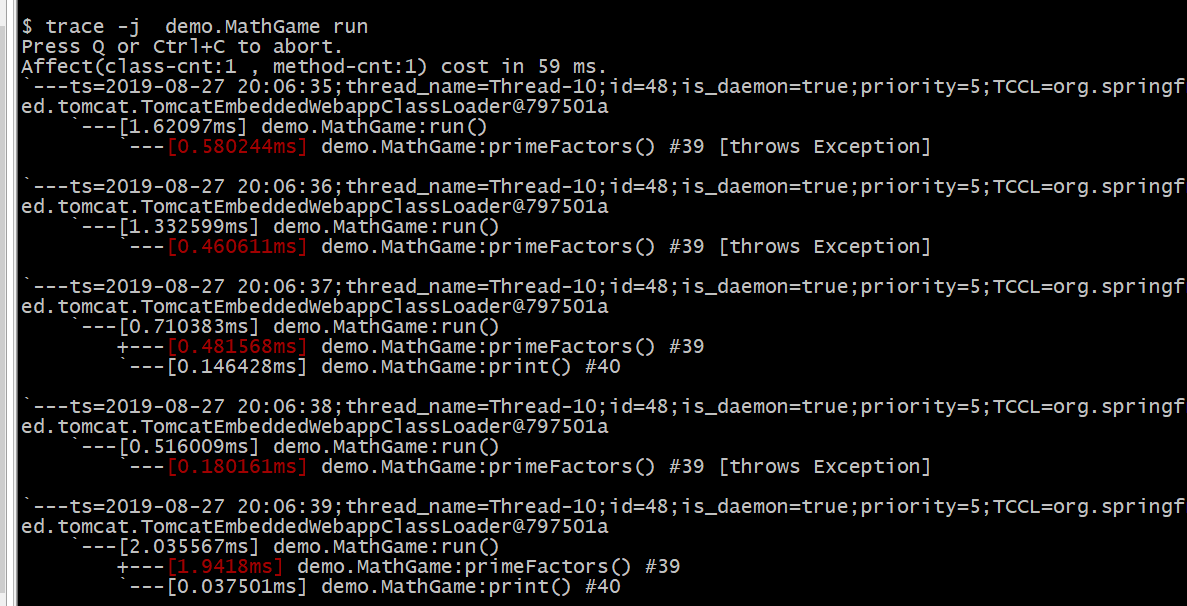
##### trace函数

$ trace demo.MathGame run



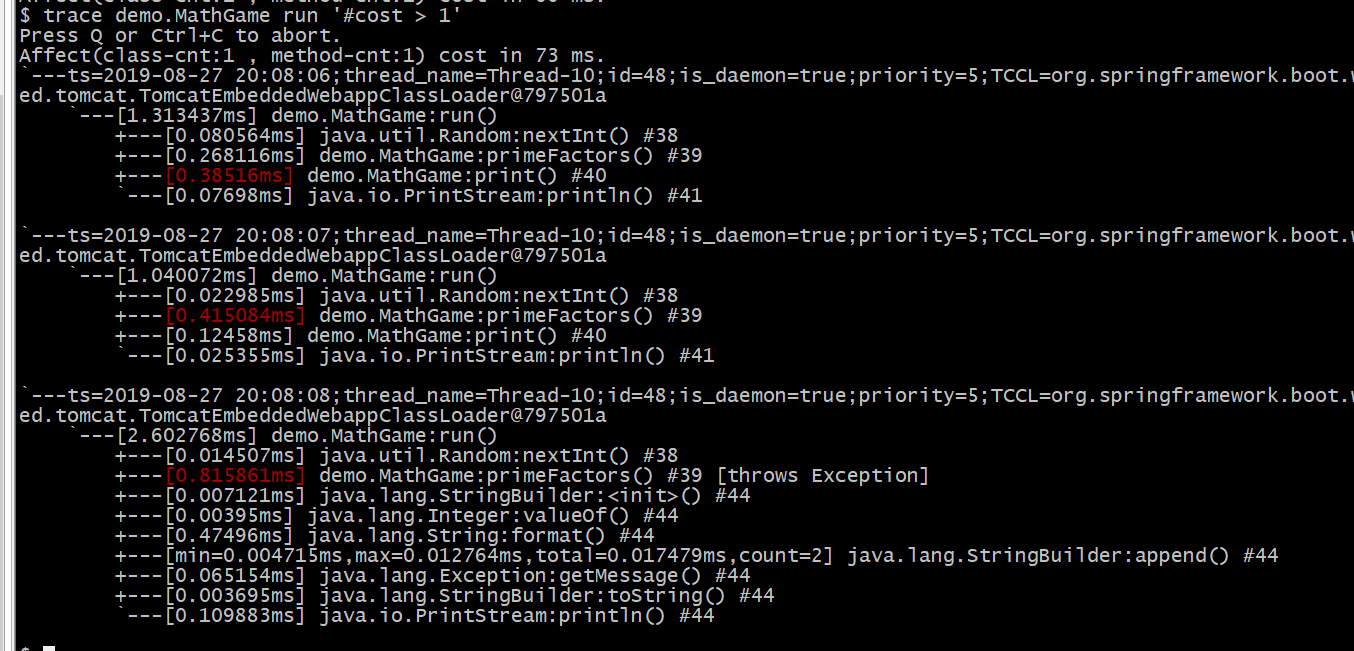
##### 过滤掉jdk的函数

$ trace -j demo.MathGame run



##### 据调用耗时过滤

$ trace demo.MathGame run '#cost > 1'



只会展示耗时大于1ms的调用路径，有助于在排查问题的时候，只关注异常情况

$ trace demo.MathGame run '#cost > 10'

Press Ctrl+C to abort.

Affect(class-cnt:1 , method-cnt:1) cost in 41 ms.

`---ts=2018-12-04 01:12:02;thread\_name=main;id=1;is\_daemon=false;priority=5;TCCL=sun.misc.Launcher$AppClassLoader@3d4eac69

`---[12.033735ms] demo.MathGame:run()

+---[0.006783ms] java.util.Random:nextInt()

+---[11.852594ms] demo.MathGame:primeFactors()

`---[0.05447ms] demo.MathGame:print()

1. 是不是很眼熟，没错，在 JProfiler 等收费软件中你曾经见识类似的功能，这里你将可以通过命令就能打印出指定调用路径。 友情提醒下，trace 在执行的过程中本身是会有一定的性能开销，在统计的报告中并未像 JProfiler 一样预先减去其自身的统计开销。所以这统计出来有些许的不准，渲染路径上调用的类、方法越多，性能偏差越大。但还是能让你看清一些事情的。
2. [12.033735ms] 的含义，12.033735 的含义是：当前节点在当前步骤的耗时，单位为毫秒
3. [0,0,0ms,11]xxx:yyy() [throws Exception]，对该方法中相同的方法调用进行了合并，0,0,0ms,11 表示方法调用耗时，min,max,total,count；throws Exception 表明该方法调用中存在异常返回
4. 这里存在一个统计不准确的问题， 就是所有方法耗时加起来可能会小于该监测方法的总耗时，这个是由于 Arthas 本身的逻辑会有一定的耗时

##### trace多个类或者多个函数

trace命令只会trace匹配到的函数里的子调用，并不会向下trace多层。因为trace是代价比较贵的，多层trace可能会导致最终要trace的类和函数非常多。

可以用正则表匹配路径上的多个类和函数，一定程度上达到多层trace的效果。$ trace -E com.test.ClassA|org.test.ClassB method1|method2|method3

### stack

输出当前方法被调用的调用路径.

很多时候我们都知道一个方法被执行，但这个方法被执行的路径非常多，或者你根本就不知道这个方法是从那里被执行了，此时你需要的是 stack 命令。

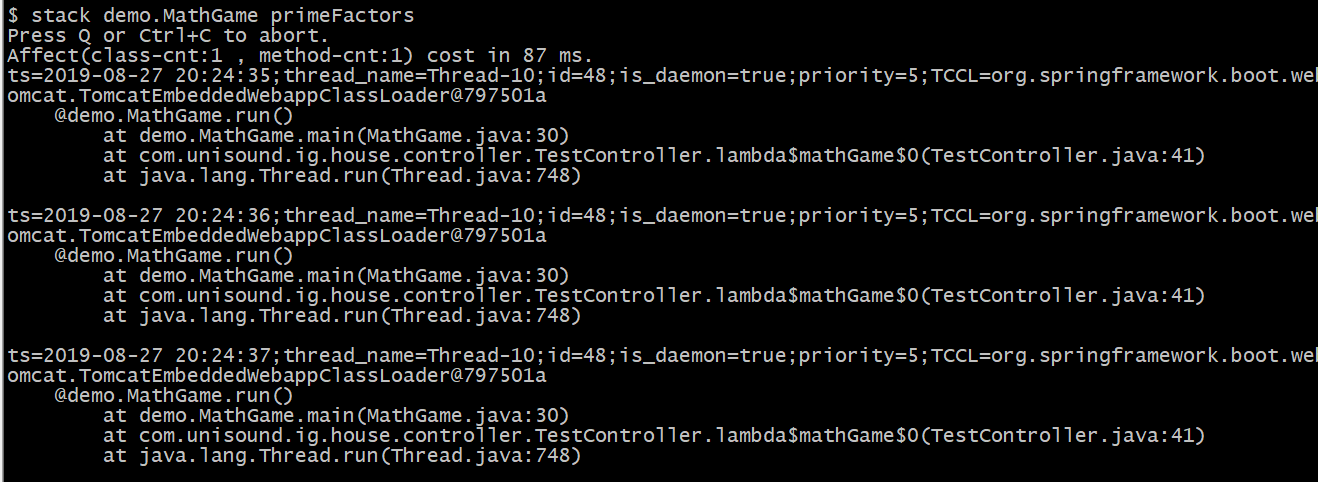
#### 参数说明

| **参数名称** | **参数说明** |
| --- | --- |
| *class-pattern* | 类名表达式匹配 |
| *method-pattern* | 方法名表达式匹配 |
| *condition-express* | 条件表达式 |
| [E] | 开启正则表达式匹配，默认为通配符匹配 |
| [n:] | 执行次数限制 |

#### 使用参考

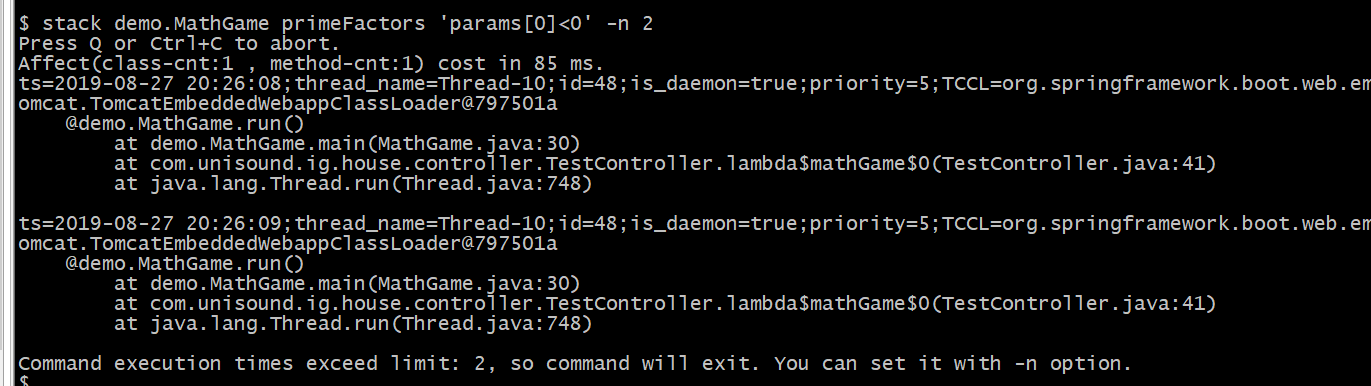
##### stack一个方法

$ stack demo.MathGame primeFactors



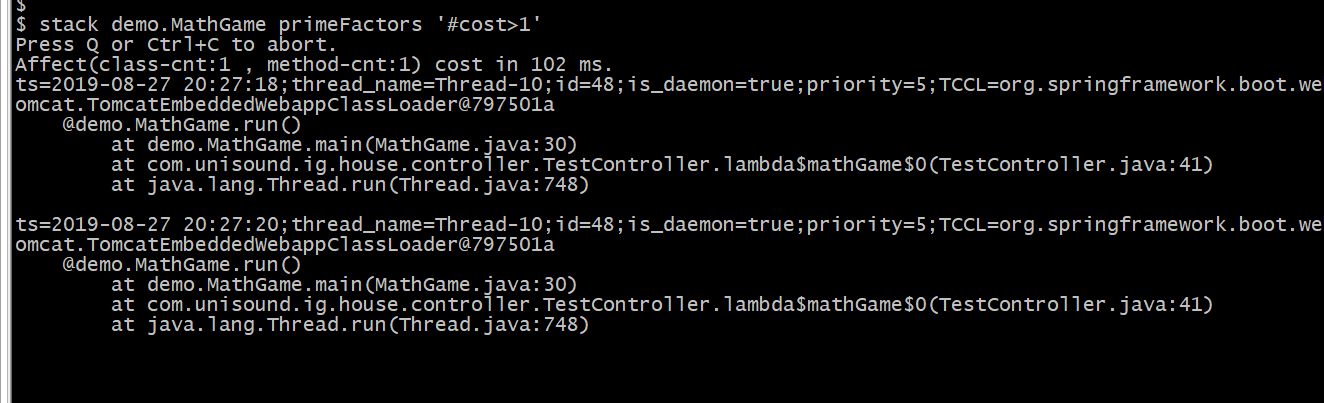
##### 据条件表达式来过滤

$ stack demo.MathGame primeFactors 'params[0]<0' -n 2



##### 据执行时间来过滤

$ stack demo.MathGame primeFactors '#cost>1'



### tt

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的调用进行观测

watch 虽然很方便和灵活，但需要提前想清楚观察表达式的拼写，这对排查问题而言要求太高，因为很多时候我们并不清楚问题出自于何方，只能靠蛛丝马迹进行猜测。

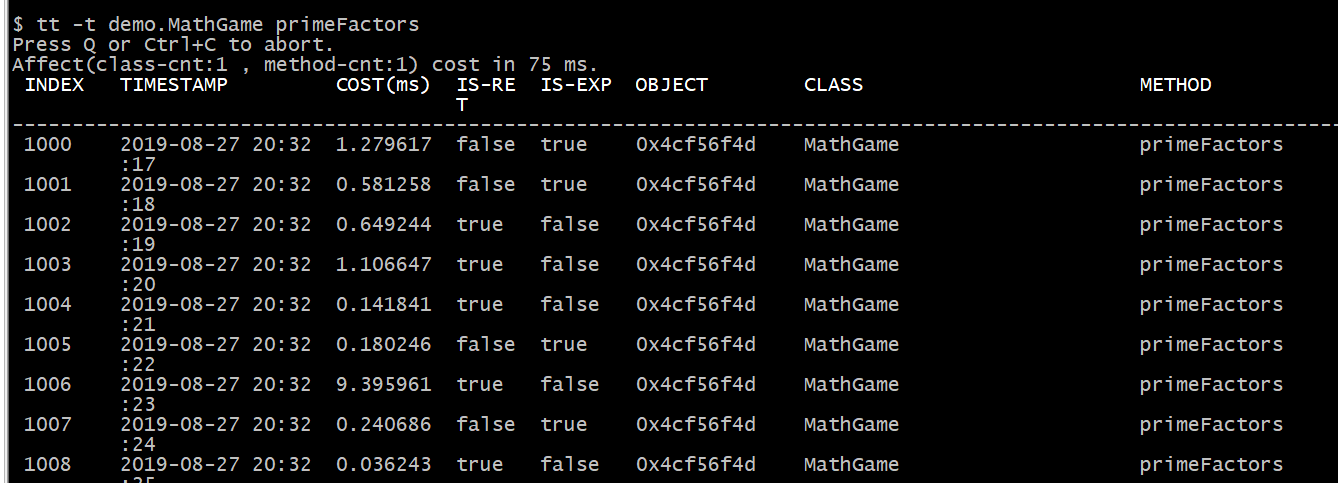
这个时候如果能记录下当时方法调用的所有入参和返回值、抛出的异常会对整个问题的思考与判断非常有帮助。

于是乎，TimeTunnel 命令就诞生了。

#### 使用参考

##### 记录调用

$ tt -t demo.MathGame primeFactors



##### 命令参数解析

###### -t

tt 命令有很多个主参数，-t 就是其中之一。这个参数的表明希望记录下指定类的方法的每次执行情况。

###### -n 3

当你执行一个调用量不高的方法时可能你还能有足够的时间用 CTRL+C 中断 tt 命令记录的过程，但如果遇到调用量非常大的方法，瞬间就能将你的 JVM 内存撑爆。

此时你可以通过 -n 参数指定你需要记录的次数，当达到记录次数时 Arthas 会主动中断tt命令的记录过程，避免人工操作无法停止的情况。

##### 表格字段说明

| **表格字段** | **字段解释** |
| --- | --- |
| INDEX | 时间片段记录编号，每一个编号代表着一次调用，后续tt还有很多命令都是基于此编号指定记录操作，非常重要。 |
| TIMESTAMP | 方法执行的本机时间，记录了这个时间片段所发生的本机时间 |
| COST(ms) | 方法执行的耗时 |
| IS-RET | 方法是否以正常返回的形式结束 |
| IS-EXP | 方法是否以抛异常的形式结束 |
| OBJECT | 执行对象的hashCode()，注意，曾经有人误认为是对象在JVM中的内存地址，  但很遗憾他不是。但他能帮助你简单的标记当前执行方法的类实体 |
| CLASS | 执行的类名 |
| METHOD | 执行的方法名 |

##### 条件表达式

* Arthas 似乎很难区分出重载的方法
* 我只需要观察特定参数，但是 tt 却全部都给我记录了下来

条件表达式也是用 OGNL来编写，核心的判断对象依然是 Advice 对象。除了 tt 命令之外，watch、trace、stack 命令也都支持条件表达式。

##### 解决方法重载

$ tt -t \*Test print params.length==1

通过指定参数个数的形式解决不同的方法签名，如果参数个数一样，你还可以这样写

$ tt -t \*Test print 'params[1] instanceof Integer'

##### 解决指定参数

$ tt -t \*Test print params[0].mobile=="13989838402"

##### 构成条件表达式的 Advice 对象

前边看到了很多条件表达式中，都使用了 params[0]，有关这个变量的介绍，请参考[表达式核心变量](https://alibaba.github.io/arthas/advice-class.html)

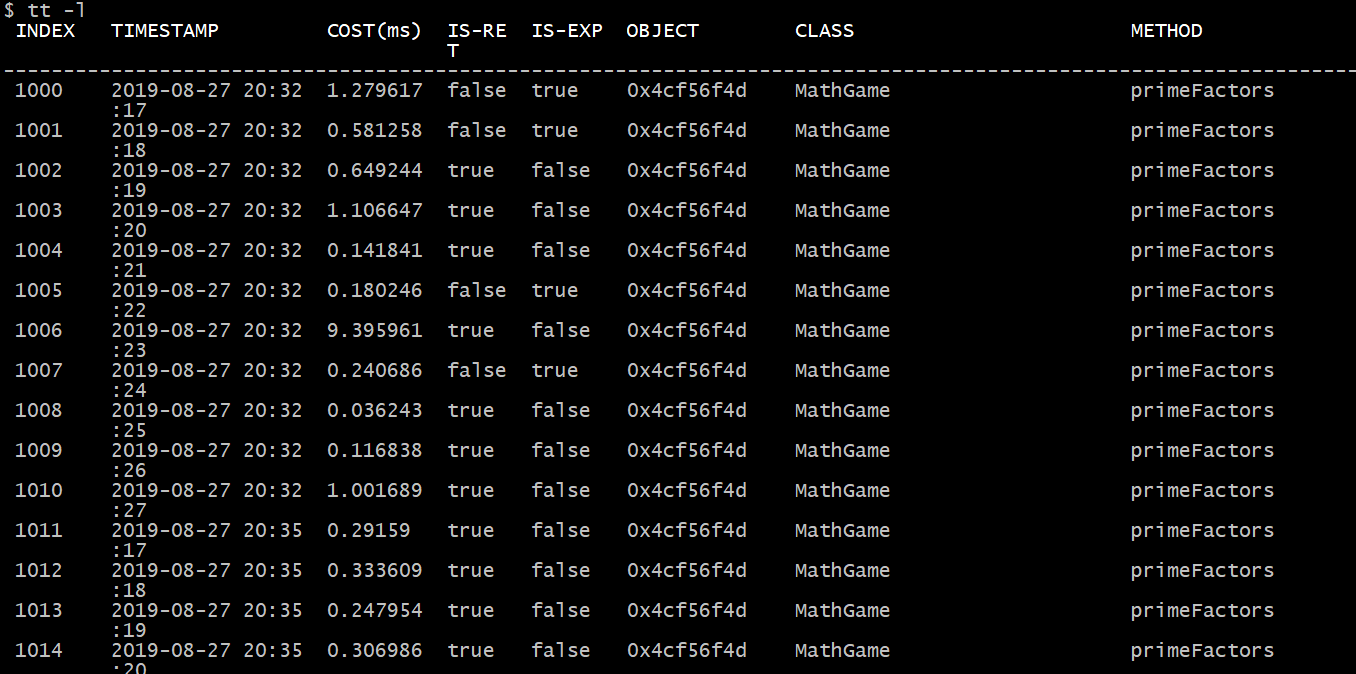
#### 检索调用记录

当你用 tt 记录了一大片的时间片段之后，你希望能从中筛选出自己需要的时间片段，这个时候你就需要对现有记录进行检索。

假设我们有这些记录

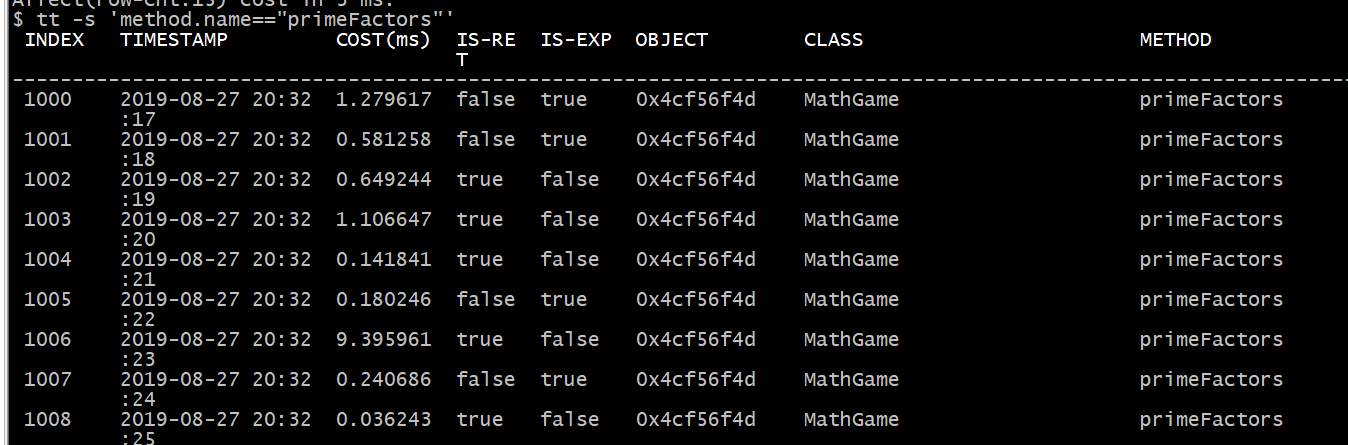
查看 已有 tt记录

$ tt -l



我需要筛选出 primeFactors 方法的调用信息

$ tt -s 'method.name=="primeFactors"'

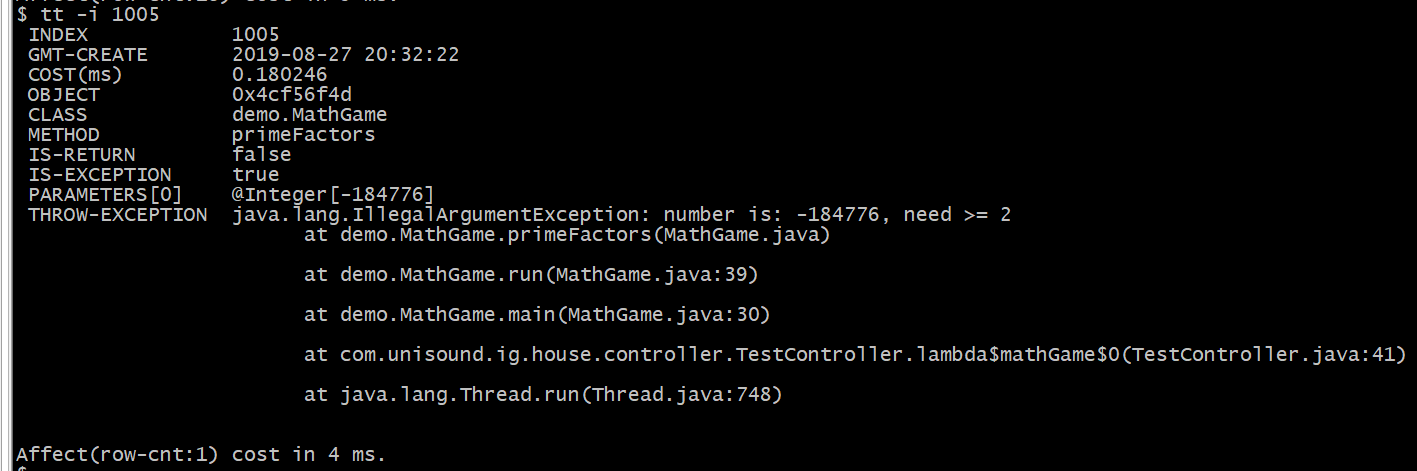


你需要一个 -s 参数。同样的，搜索表达式的核心对象依旧是 Advice 对象。

#### 查看调用信息

对于具体一个时间片的信息而言，你可以通过 -i 参数后边跟着对应的 INDEX 编号查看到他的详细信息。

$ tt -i 1005

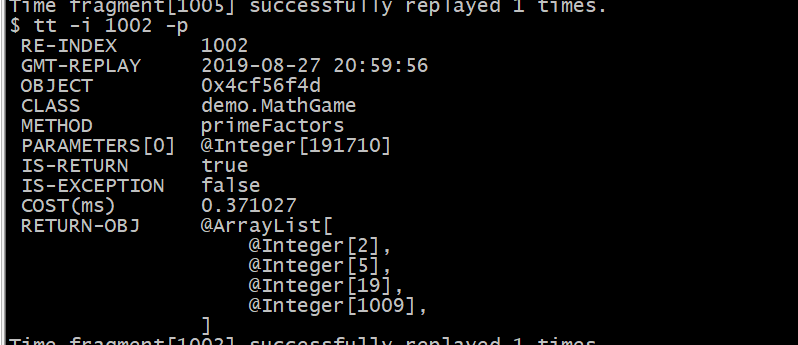


#### 重做一次调用

当你稍稍做了一些调整之后，你可能需要前端系统重新触发一次你的调用，此时得求爷爷告奶奶的需要前端配合联调的同学再次发起一次调用。而有些场景下，这个调用不是这么好触发的。

tt 命令由于保存了当时调用的所有现场信息，所以我们可以自己主动对一个 INDEX 编号的时间片自主发起一次调用，从而解放你的沟通成本。此时你需要 -p 参数。通过 --replay-times 指定 调用次数，通过 --replay-interval 指定多次调用间隔(单位ms, 默认1000ms)

$ tt -i 1002 –p



你会发现结果虽然一样，但调用的路径发生了变化，有原来的程序发起变成了 Arthas 自己的内部线程发起的调用了。

#### 获取spring context中的bean

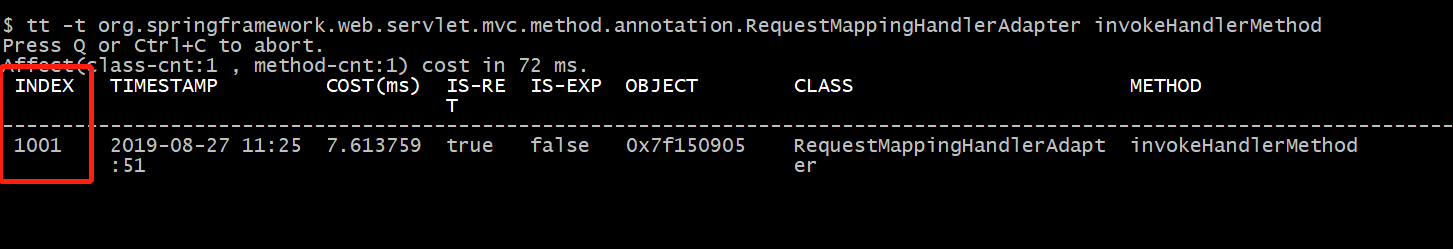
##### 启动spring-boot项目

##### 执行以下命令

$ tt -t org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter invokeHandlerMethod

然后访问一个服务一个接口. <http://192.168.188.131:8809/health>

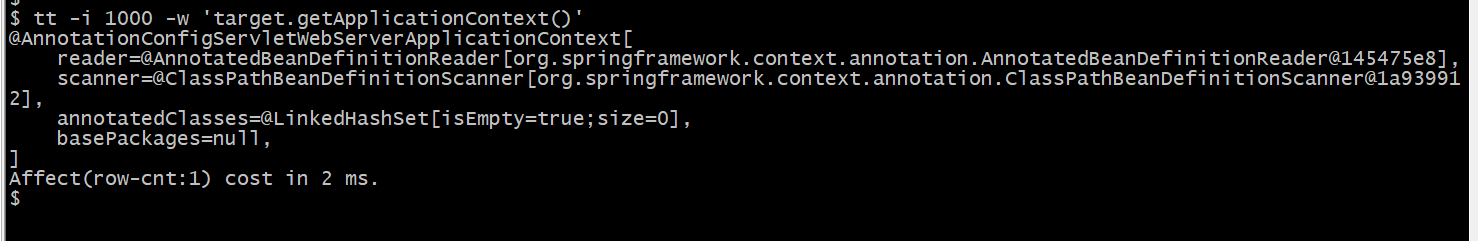
可以看到Arthas会拦截到这个调用，index是1000，并且打印出：



##### 获取spring context

可以用tt命令的-i参数来指定index，并且用-w参数来执行ognl表达式来获取spring context：

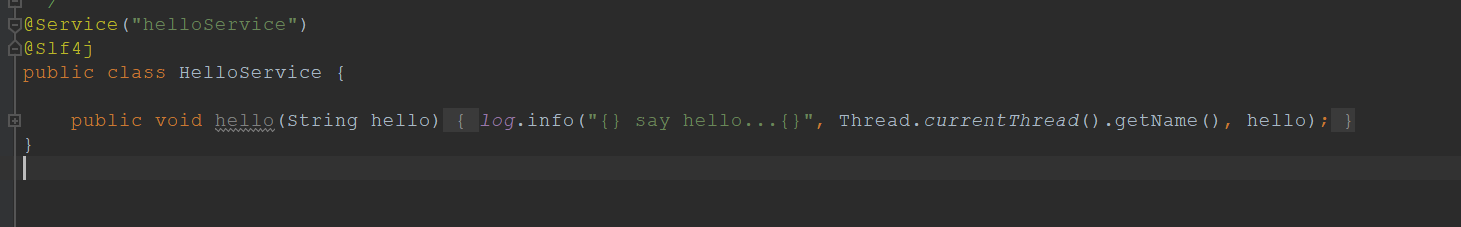
$ tt -i 1000 -w 'target.getApplicationContext()'

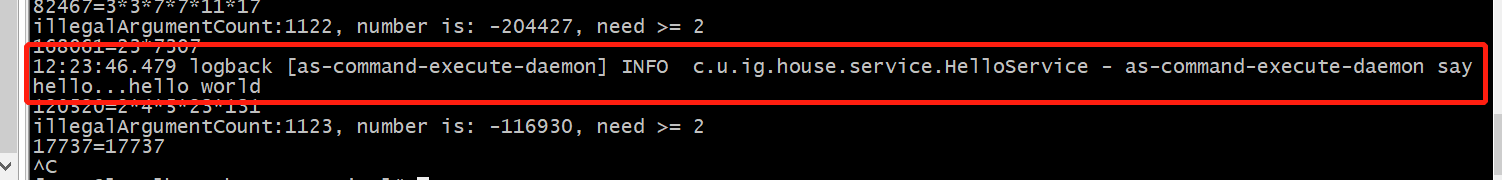


##### 从spring context中获取任意bean

##获取helloService bean , 执行 hello() 方法.

$ tt -i 1000 -w 'target.getApplicationContext().getBean("helloService").hello("hello world")'





#### 需要强调的点

##### ThreadLocal 信息丢失

很多框架偷偷的将一些环境变量信息塞到了发起调用线程的 ThreadLocal 中，由于调用线程发生了变化，这些 ThreadLocal 线程信息无法通过 Arthas 保存，所以这些信息将会丢失。一些常见的 CASE 比如：鹰眼的 TraceId 等。

##### 引用的对象

需要强调的是，tt 命令是将当前环境的对象引用保存起来，但仅仅也只能保存一个引用而已。如果方法内部对入参进行了变更，或者返回的对象经过了后续的处理，那么在 tt 查看的时候将无法看到当时最准确的值。这也是为什么 watch 命令存在的意义。

## options

查看或设置Arthas全局开关

#### 全局开关

| **名称** | **默认值** | **描述** |
| --- | --- | --- |
| unsafe | false | 是否支持对系统级别的类进行增强，打开该开关可能导致把JVM搞挂，请慎重选择！ |
| dump | false | 是否支持被增强了的类dump到外部文件中，如果打开开关，class文件会被dump到/${application dir}/arthas-class-dump/目录下，具体位置详见控制台输出 |
| batch-re-transform | true | 是否支持批量对匹配到的类执行retransform操作 |
| json-format | false | 是否支持json化的输出 |
| disable-sub-class | false | 是否禁用子类匹配，默认在匹配目标类的时候会默认匹配到其子类，如果想精确匹配，可以关闭此开关 |
| debug-for-asm | false | 打印ASM相关的调试信息 |
| save-result | false | 是否打开执行结果存日志功能，打开之后所有命令的运行结果都将保存到/home/admin/logs/arthas/arthas.log中 |
| job-timeout | 1d | 异步后台任务的默认超时时间，超过这个时间，任务自动停止；比如设置 1d, 2h, 3m, 25s，分别代表天、小时、分、秒 |

#### 使用说明

查看所有开关

$ options

查看单个开关

$ options save-result

修改单个开关

$ options save-result true

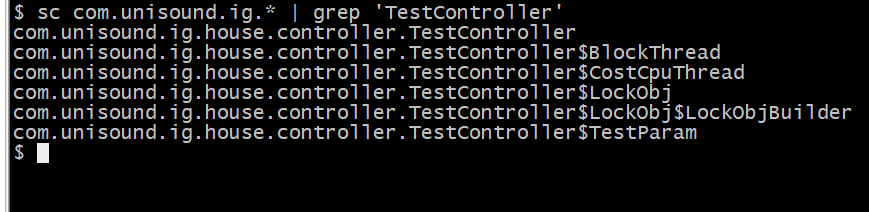
## 管道

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

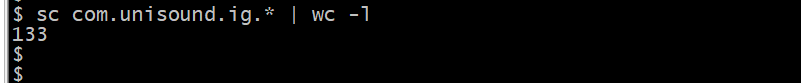
示例

$ sc com.unisound.ig.\* | grep 'TestController'



$ sc com.unisound.ig.\* | plaintext //不知道有啥用..

$ sc com.unisound.ig.\* | wc –l



## 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.

arthas中的异步调用，使用了仿linux系统任务相关的命令。[linux任务相关介绍](https://ehlxr.me/2017/01/18/Linux-%E4%B8%AD-fg%E3%80%81bg%E3%80%81jobs%E3%80%81-%E6%8C%87%E4%BB%A4/)。

### 使用&在后台执行任务

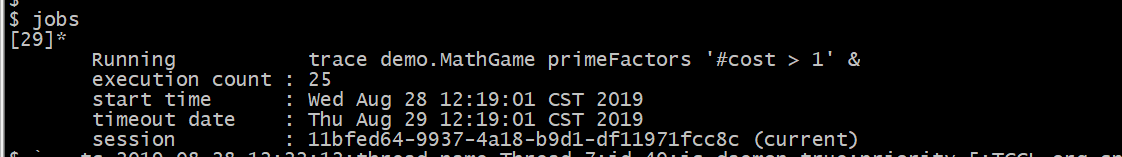
$ trace demo.MathGame primeFactors '#cost > 1' &

这时命令在后台执行，可以在console中继续执行其他命令。

### 通过jobs查看任务

如果希望查看当前有哪些arthas任务在执行，可以执行jobs命令，执行结果如下

$ jobs



可以看到目前有一个后台任务在执行。

* job id是10, \* 表示此job是当前session创建
* 状态是Running
* execution count是执行次数，从启动开始已经执行了19次
* timeout date是超时的时间，到这个时间，任务将会自动超时退出

### 任务的暂停和取消

当任务正在前台执行，比如直接调用命令trace Test t或者调用后台执行命令trace Test t &后又通过fg命令将任务转到前台。这时console中无法继续执行命令，但是可以接收并处理以下事件：

* ‘ctrl + z’：将任务暂停。通过jbos查看任务状态将会变为Stopped，通过bg 或者fg 可让任务重新开始执行
* ‘ctrl + c’：停止任务
* ‘ctrl + d’：按照linux语义应当是退出终端，目前arthas中是空实现，不处理

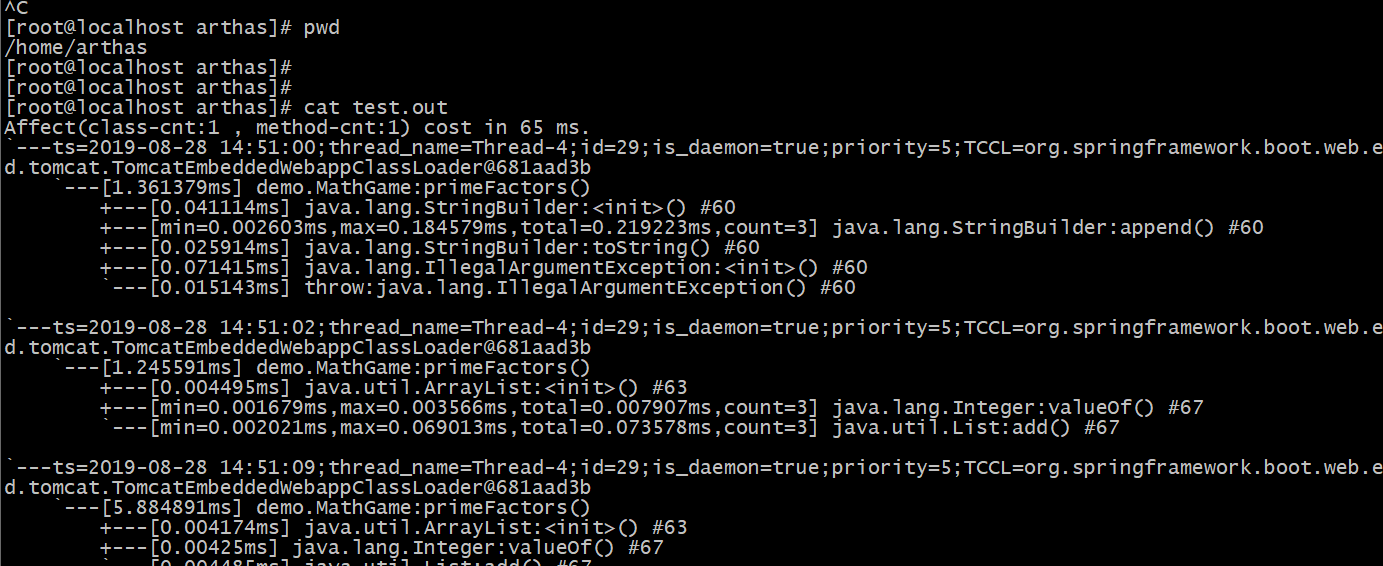
### fg、bg命令，将命令转到前台、后台继续执行

* 任务在后台执行或者暂停状态（ctrl + z暂停任务）时，执行fg 将可以把对应的任务转到前台继续执行。在前台执行时，无法在console中执行其他命令
* 当任务处于暂停状态时（ctrl + z暂停任务），执行bg 将可以把对应的任务在后台继续执行
* 非当前session创建的job，只能由当前session fg到前台执行.

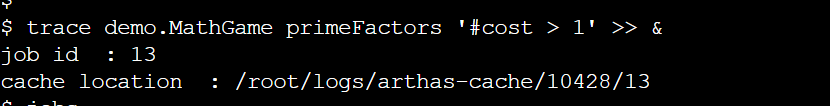
### 任务输出重定向

可通过>或者>>将任务输出结果输出到指定的文件中，可以和&一起使用，实现arthas命令的异步调用。比如：

$ trace demo.MathGame primeFactors '#cost > 1' >> /home/arthas/test.out &



当连接到远程的arthas server时，可能无法查看远程机器的文件，arthas同时支持了自动重定向到本地缓存路径。使用方法如下：



可以看到并没有指定重定向文件位置，arthas自动重定向到缓存中了，执行命令后会输出job id和cache location。cache location就是重定向文件的路径，在系统logs目录下，路径包括pid和job id，避免和其他任务冲突。命令输出结果到/root/logs/arthas-cache/10428/13

### 停止任务

$ kill <job-id>

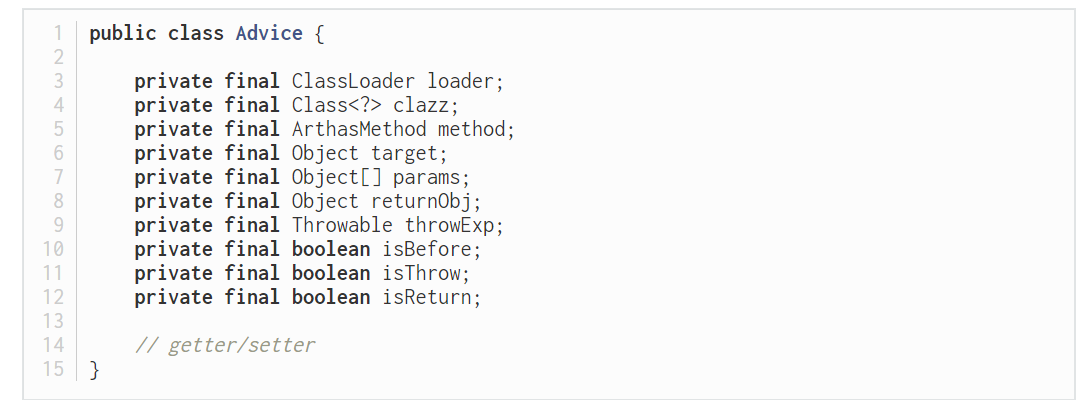
### 其他

* 最多同时支持8个命令使用重定向将结果写日志
* 请勿同时开启过多的后台异步命令，以免对目标JVM性能造成影响

# 表达式核心变量

无论是匹配表达式也好、观察表达式也罢，他们核心判断变量都是围绕着一个 Arthas 中的通用通知对象 Advice 进行。

它的简略代码结构如下



这里列一个表格来说明不同变量的含义

| **变量名** | **变量解释** |
| --- | --- |
| loader | 本次调用类所在的 ClassLoader |
| clazz | 本次调用类的 Class 引用 |
| method | 本次调用方法反射引用 |
| target | 本次调用类的实例 |
| params | 本次调用参数列表，这是一个数组，如果方法是无参方法则为空数组 |
| returnObj | 本次调用返回的对象。当且仅当 isReturn==true 成立时候有效，表明方法调用是以正常返回的方式结束。如果当前方法无返回值 void，则值为 null |
| throwExp | 本次调用抛出的异常。当且仅当 isThrow==true 成立时有效，表明方法调用是以抛出异常的方式结束。 |
| isBefore | 辅助判断标记，当前的通知节点有可能是在方法一开始就通知，此时 isBefore==true 成立，同时 isThrow==false 和 isReturn==false，因为在方法刚开始时，还无法确定方法调用将会如何结束。 |
| isThrow | 辅助判断标记，当前的方法调用以抛异常的形式结束。 |
| isReturn | 辅助判断标记，当前的方法调用以正常返回的形式结束。 |

所有变量都可以在表达式中直接使用，如果在表达式中编写了不符合 OGNL 脚本语法或者引入了不在表格中的变量，则退出命令的执行；用户可以根据当前的异常信息修正条件表达式或观察表达式

# Docker

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

# 案例

<https://github.com/alibaba/arthas/issues?q=label%3Auser-case>

# 常见问题

<https://github.com/alibaba/arthas/issues?q=label%3Aquestion-answered>

# QA

…