容器CPU满载问题诊断思路

1. 问题现象
    1. 最直接的现象，是该模块相关功能（通过页面或者自动化测试触发），响应变慢甚至超时。
    2. k8s命令查看
        1. 例如在deploy配置中，inetwork的配额是2核4g

```
resources:
  limits:
    cpu: "2"
    memory: 4Gi
```

        2. 使用`kubectl top po -n incloud | grep inetwork`命令查看inetwork pod的cpu负载

```
[root@node3 ~]# kubectl top po -n incloud | grep inetwork
inetwork-54b9b5f654-76c67                        6m           818Mi
inetwork-54b9b5f654-gqjrh                        3m           816Mi
inetwork-54b9b5f654-x44nk                        9m           915Mi
```

        这里的单位是m的含义是 1000m=1核，所以如果该列的值接近或者超过2000m，代表该pod cpu接近满载。
2. 可能原因
    1. 系统内线程数量过多。
    2. 死循环。
    3. Full GC次数增大。
3. 定位方法
    1. 查看线程总数是否过高

        曾经出现过的一个问题，由于业务模块不合理的创建线程池，导致同时存活超过10000个线程。

        虽然每个线程的负载不高，但是消耗在线程之间上下文切换的CPU过高，进而导致整个系统响应迟滞。

        这里我们借助arthas观测系统内线程，示例如下：

        - 查看线程总数：`thread --all | wc -l`

```
[arthas@1]$ thread --all | wc -l
187
```

- 查看占CPU负载前5的线程：`thread -n 5`

```
[arthas@1]$ thread -n 5
"arthas-command-execute" Id=3121 cpuUsage=1.29% deltaTime=2ms time=66ms RUNNABLE
    at sun.management.ThreadImpl.dumpThreads0(Native Method)
    at sun.management.ThreadImpl.getThreadInfo(ThreadImpl.java:461)
    at com.taobao.arthas.core.command.monitor200.ThreadCommand.processTopBusyThreads(ThreadCommand.java:206)
    at com.taobao.arthas.core.command.monitor200.ThreadCommand.process(ThreadCommand.java:122)
    at com.taobao.arthas.core.shell.command.impl.AnnotatedCommandImpl.process(AnnotatedCommandImpl.java:82)
    at com.taobao.arthas.core.shell.command.impl.AnnotatedCommandImpl.access$100(AnnotatedCommandImpl.java:18)
    at com.taobao.arthas.core.shell.command.impl.AnnotatedCommandImpl$ProcessHandler.handle(AnnotatedCommandImpl.java:111)
    at com.taobao.arthas.core.shell.command.impl.AnnotatedCommandImpl$ProcessHandler.handle(AnnotatedCommandImpl.java:108)
    at com.taobao.arthas.core.shell.system.impl.ProcessImpl$CommandProcessTask.run(ProcessImpl.java:385)
    at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:511)
    at java.util.concurrent.FutureTask.run(FutureTask.java:266)
    at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.access$201(ScheduledThreadPoolExecutor.java:180)
    at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.run(ScheduledThreadPoolExecutor.java:293)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
    at java.lang.Thread.run(Thread.java:750)


"C1 CompilerThread1" [Internal] cpuUsage=1.16% deltaTime=2ms time=27527ms


"VM Periodic Task Thread" [Internal] cpuUsage=0.05% deltaTime=0ms time=50348ms


"pool-3-thread-1" Id=39 cpuUsage=0.03% deltaTime=0ms time=27284ms TIMED_WAITING
    at java.lang.Thread.sleep(Native Method)
    at io.netty.util.HashedWheelTimer$Worker.waitForNextTick(HashedWheelTimer.java:579)
    at io.netty.util.HashedWheelTimer$Worker.run(HashedWheelTimer.java:478)
    at java.lang.Thread.run(Thread.java:750)


"nioEventLoopGroup-5-3" Id=124 cpuUsage=0.02% deltaTime=0ms time=2496ms RUNNABLE (in native)
    at sun.nio.ch.EPollArrayWrapper.epollWait(Native Method)
    at sun.nio.ch.EPollArrayWrapper.poll(EPollArrayWrapper.java:269)
    at sun.nio.ch.EPollSelectorImpl.doSelect(EPollSelectorImpl.java:93)
    at sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:86)
    at sun.nio.ch.SelectorImpl.select(SelectorImpl.java:97)
    at io.netty.channel.nio.SelectedSelectionKeySetSelector.select(SelectedSelectionKeySetSelector.java:62)
    at io.netty.channel.nio.NioEventLoop.select(NioEventLoop.java:791)
    at io.netty.channel.nio.NioEventLoop.run(NioEventLoop.java:439)
    at io.netty.util.concurrent.SingleThreadEventExecutor$5.run(SingleThreadEventExecutor.java:906)
    at io.netty.util.internal.ThreadExecutorMap$2.run(ThreadExecutorMap.java:74)
    at io.netty.util.concurrent.FastThreadLocalRunnable.run(FastThreadLocalRunnable.java:30)
    at java.lang.Thread.run(Thread.java:750)
```

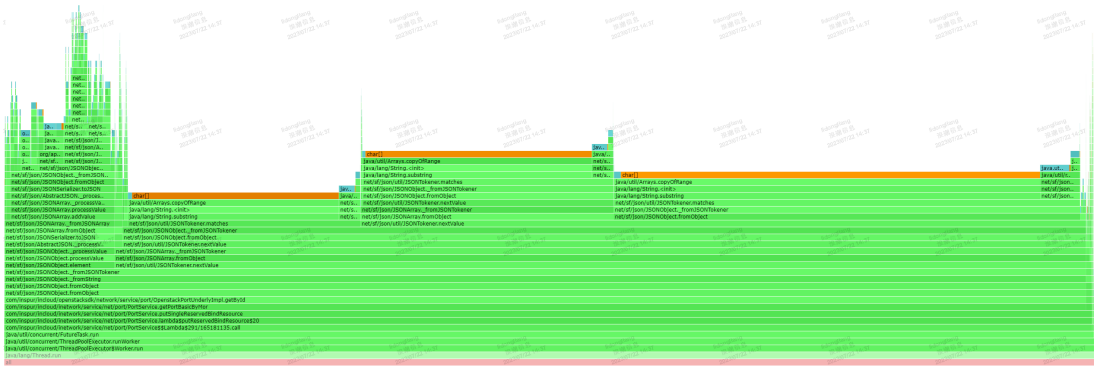
- 查看全部线程堆栈，按照CPU负载从高到底排序，并导出为容器内某个文件：`thread -n -1 > a`

2. 借助火焰图直观的观测CPU时间片消耗在哪里

生成火焰图，这里我们使用arthas的profiler命令：

- 开始收集：`profiler start -e cpu`。
- 等待一段时间，过程中可以使用命令`profiler status`查看收集时长，使用命令`profiler getSamples`查看已经收集的样本数。
- 停止收集：`profiler stop`，默认生成html格式的火焰图，存储在容器内的指定位置。

阅读火焰图：



Arthas的profiler命令收集CPU的执行样本，如30s收集到10000个样本，生成如上火焰图。

y 轴表示调用栈，每一层都是一个函数。调用栈越深，火焰就越高，顶部就是正在执行的函数，下方都是它的父函数。

x 轴表示抽样数，如果一个函数在 x 轴占据的宽度越宽，就表示它被抽到的次数多，即执行的时间长。注意，x 轴不代表时间，而是所有的调用栈合并后，按字母顺序排列的。

火焰图就是看顶层的哪个函数占据的宽度最大。只要有"平顶"（plateaus），就表示该函数可能存在性能问题。

颜色没有特殊含义，因为火焰图表示的是 CPU 的繁忙程度，所以一般选择暖色调。

不同时间段收集到的火焰图不同，如果想得到清晰的火焰图，需要多收集几次。

4. 预防方法

在新增一个函数后，如果包含多线程的应用，或者代码里存在while死循环，或者担心占用内存过大频繁的触发gc。可以在提交测试之前，使用**jmeter并发压测**，同时观察cpu负载及系统响应速度。

参考资料：

- 如何读懂火焰图？