# 金蝶天燕对广州市政府项目出现的系统卡顿问题的分析与解决过程报告

**摘要：**

2021年6月29日，金蝶天燕接到广州市政府项目中出现的系统卡顿问题反馈，经现场服务人员处理无果后，派出总部研发人员到现场支持，对系统运行进行监测和分析。在此过程中，金蝶天燕的技术人员根据现场监测信息，从中间件层面进行了优化，并通过联系华为毕昇JDK的技术人员，对JDK进行了升级，部分解决了系统卡顿及资源占用过高的问题；后经总部人员对监测日志的分析，发现了应用中存在隐藏的agent挂载操作，并基本确认该操作为引发系统卡顿的关键原因。本报告对问题的处理过程进行了详细的描述。

# 问题描述

## 主要现象

自2021年6月29日起，广州市政府部署的办公应用系统出现不定期卡顿的情况，最初表现为CPU使用率高、应用卡顿，经过我方调集研发人员及华为毕昇JDK的技术人员，从中间件平台层面进行排查，陆续发现了系统运行缓慢、Jconsole无法连接AAS、堆栈线程拥堵等问题。

## 应用运行环境

中间件平台： 金蝶天燕Apusic应用服务器

JDK： OpenJDK1.8.0\_181

服务器： 4台服务器（虚拟机）/ CPU 麒麟920 / 16核 / 内存64G

操作系统： NeoKylinV7

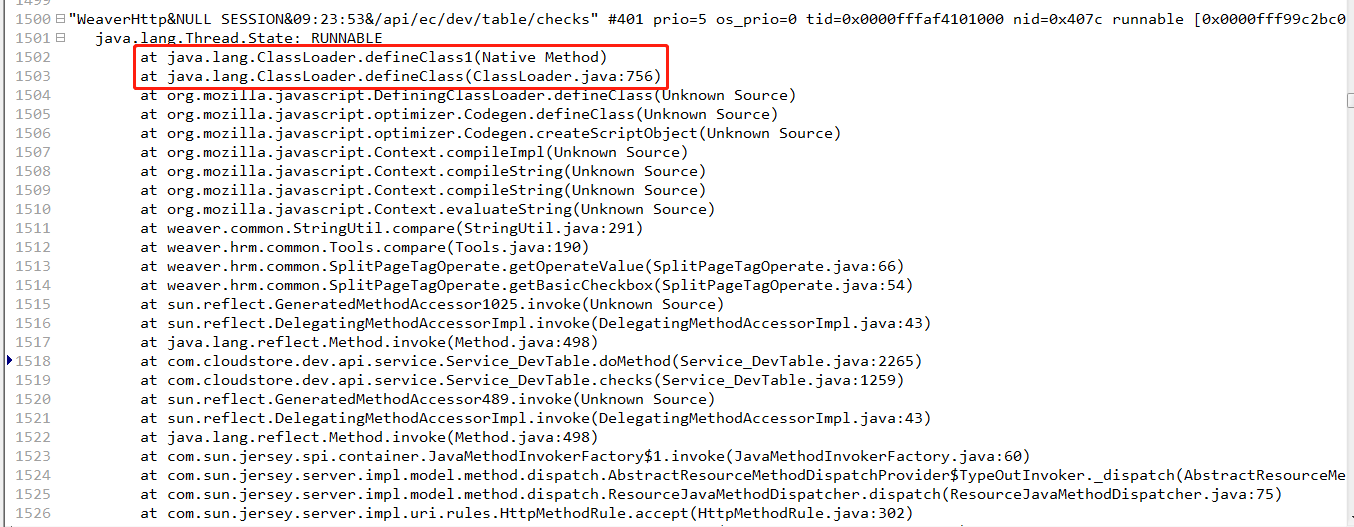
# 处理过程

## ISSUE 2021062901

**（1）现象描述：**

2021年6月29日，系统中出现了CPU使用率过高的问题。现场运维人员使用top命令，发现java进程的CPU使用率达到1500%，客户反馈系统很慢。

现场运维人员分析了堆栈和CPU使用率监控信息，发现是很多WeaverHttp的线程都在进行类加载操作，如下图：



**（2）分析步骤：**

1. 切换了OracleJDK、OpenJDK、AdoptJDK，类加载仍消耗大量CPU；
2. 分析了金蝶天燕Apusic应用服务器的类加载过程和机制，发现并不会重复加载类；
3. 分析了应用的代码，发现存在很多动态编译javascript表达式为java的代码，并且进行加载，除此之外，应用本身还存在其他类加载操作，如CGlib和反射。

**（3）解决方案：**

由于是虚拟机环境+国产芯片，只要是第一次加载的类就会消耗大量CPU资源，该问题后续可能还会出现，需要协调客户让应用厂商尽量减少动态生成类的方式。因此，该问题需由应用厂商升级自身的产品解决，我方将持续监控并反馈。

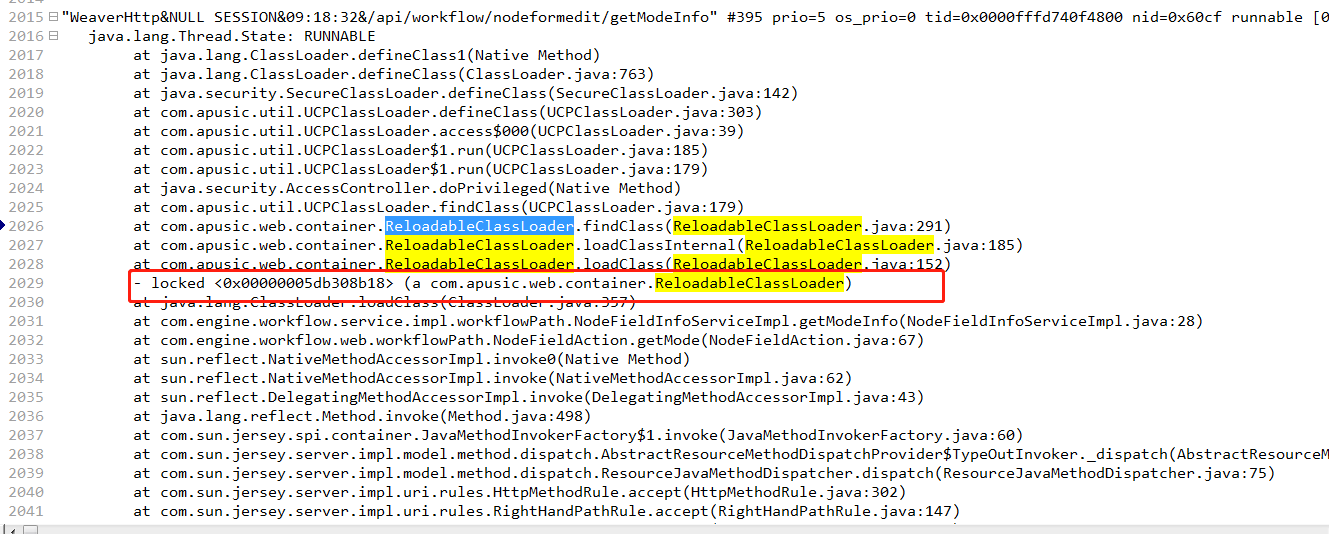
## **ISSUE2021062902**

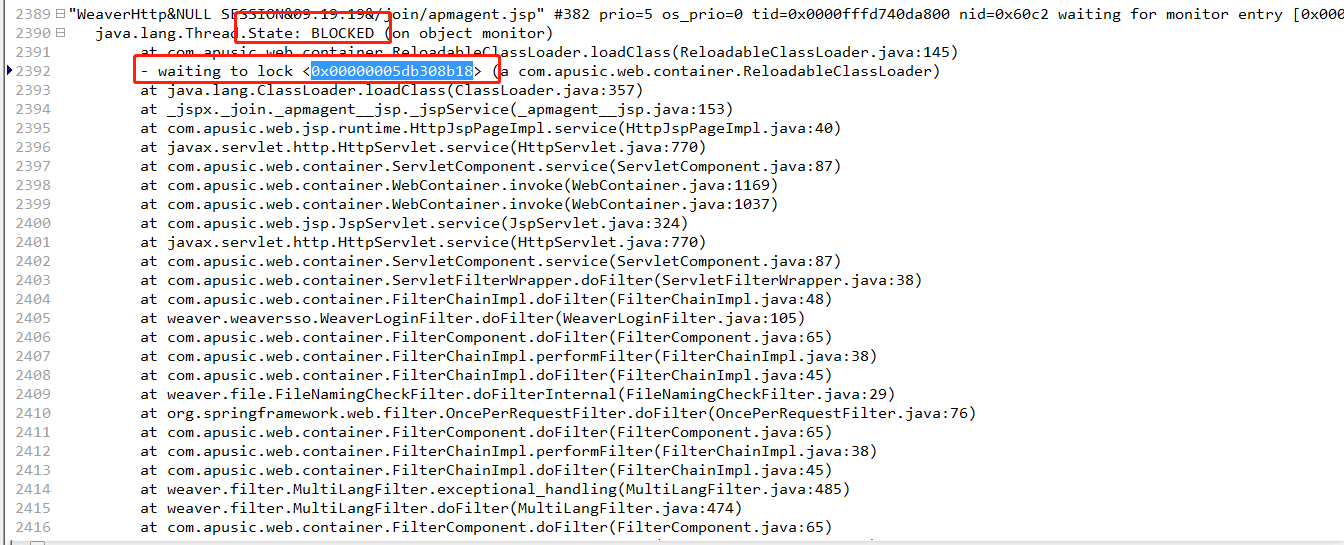
1. **现象描述：**

2021年6月29日，系统出现了运行缓慢的问题，系统CPU不高，始终有一个核是满的，但是用户反馈系统使用卡顿。

**（2）分析步骤：**

1. 查看堆栈，可以查看到堆栈有很多处于BLOCKED状态，都在等待锁，造成阻塞。同一时间只能有一个请求进行类加载，必须等该请求加载完成之后才能进行下一个请求执行类加载。如果加载类很慢就会出现阻塞时间很久的问题，关键线索如图。





1. 应用一般在晚上进行重启，第二天早上进行访问，很多类都是第一次加载，因此早上时间段出现卡顿的概率较大。

**（3）解决方案：**

增加一个工具类，在系统启动的时候执行类的预加载操作（加载WEB-INF/lib,classes目录下的类）

具体操作如下：

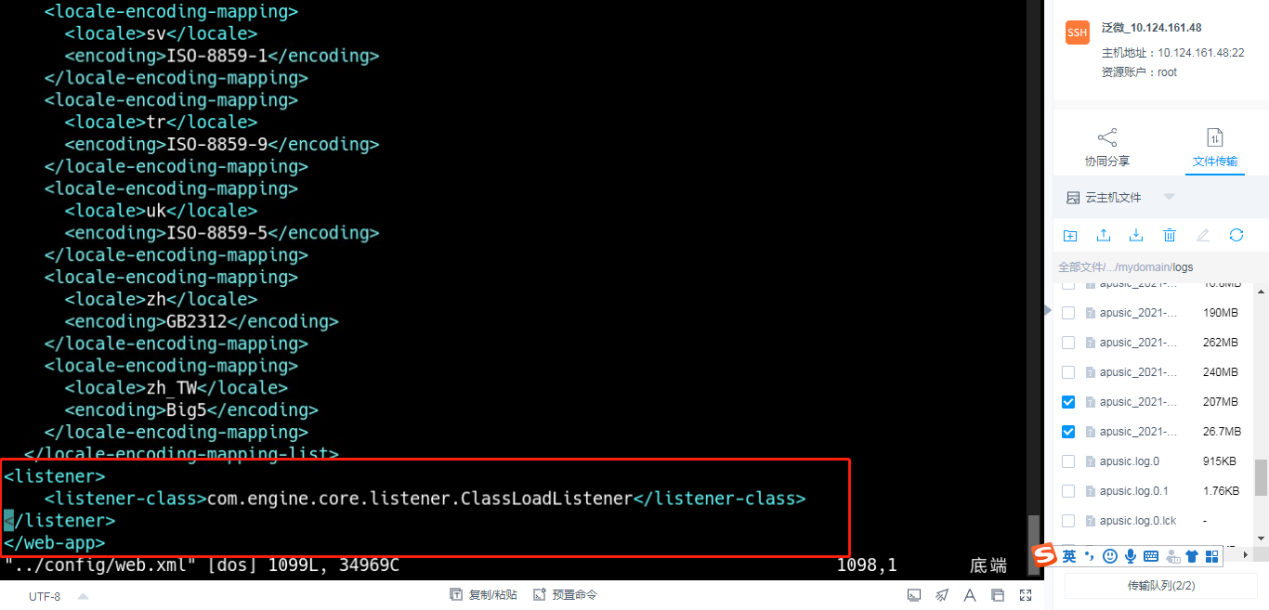
1. 把class-pre-loader-listener.jar放入到AAS/domains/mydomain/lib目录下
2. 修改AAS/domains/mydomain/config/web.xml文件，在最后增加如下内容：

<listener>

<listener-class>com.engine.core.listener.ClassLoadListener</listener-class>

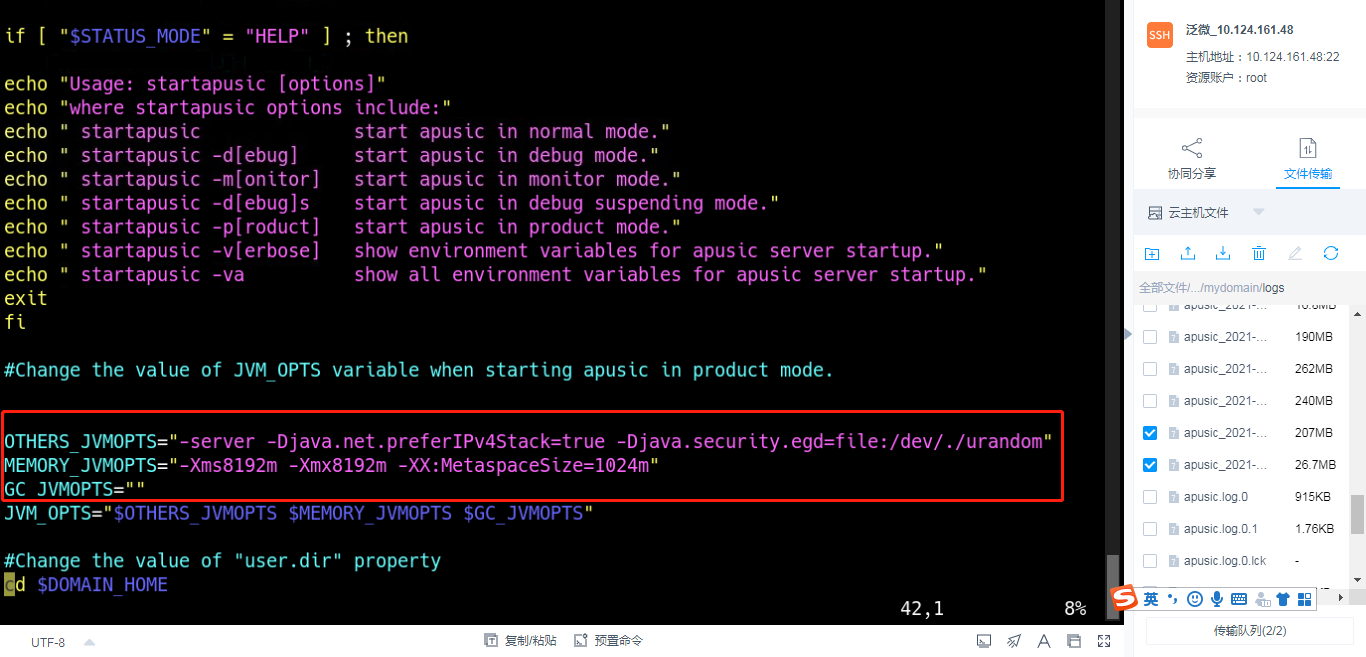
</listener>

如下图所示：



1. 修改JVM内存和GC算法

因为客户的类比较多，采用默认的配置时，需要把元数据空间设置大一些，如图。



另外，可以把XX:MetaspaceSize设置大一些，建议值1536m。

1. SP目录放了sp.jar，该模块用于监控AAS的UCPClassLoader调用defineClass的操作，增加了记录日志，将该模块去掉，可以降低对性能的影响。

## ISSUE2021070101

1. **现象描述：**

2021年7月1日，系统出现了Jconsole连接不上AAS的问题。

运维平台需要监控AAS，用JDK11的Jconsole连不上AAS，AAS运行在JDK1.8上面

**（2）解决方案：**

1. 如果telnet无法访问，则需找对应人员开放JMX端口的访问权限；
2. 由于JDK9往上的版本不支持iiop的协议，RMIAdaptorService需要使用rmi-jrmp协议进行重新实现，访问的url和端口发送了变化，查阅相关文档进行配置的更改，AAS/domains/mydomain/config/apusic.conf中修改RMIAdaptorService服务下，如下：

<SERVICE

CLASS="com.apusic.jmx.adaptors.rmi.RMIAdaptorService"

NAME="Adaptor:type=rmi"

>

<ATTRIBUTE NAME="ServerPort" VALUE="6666"/>

<ATTRIBUTE NAME="ServerAddress" VALUE="10.124.161.49"/>

</SERVICE>

## ISSUE2021070501

**（1）现象描述：**

2021年7月5日晚，再次出现卡顿和CPU高的问题，获取堆栈后发现多个线程堵住，堆栈信息如下：

"WeaverHttp&NULL SESSION&17:50:32&/api/msgcenter/homepage/getMsgCount" #342 prio=5 os\_prio=0 tid=0x0000fffb58172000 nid=0x19d5 waiting for monitor entry [0x0000fff9bedfc000]

java.lang.Thread.State: BLOCKED (on object monitor)

at sun.misc.ObjectInputFilter$Config.getSerialFilter(ObjectInputFilter.java:307)

- waiting to lock <0x00000005c2576300> (a java.lang.Object)

at java.io.ObjectInputStream.<init>(ObjectInputStream.java:393)

at com.cloudstore.dev.api.util.Util\_Serializer.deserialize(Util\_Serializer.java:45)

at com.cloudstore.dev.api.util.Util\_Account.getAccountMapRedis(Util\_Account.java:66)

at com.cloudstore.dev.api.util.Util\_Account.getAccountMap(Util\_Account.java:210)

at com.engine.msgcenter.cmd.homepage.GetMsgCountCmd.execute(GetMsgCountCmd.java:56)

at com.engine.msgcenter.cmd.homepage.GetMsgCountCmd.execute(GetMsgCountCmd.java:31)

at com.engine.core.interceptor.CommandInvoker.execute(CommandInvoker.java:38)

at com.engine.core.interceptor.CommandContextInterceptor.execute(CommandContextInterceptor.java:42)

at com.engine.core.interceptor.LogInterceptor.execute(LogInterceptor.java:19)

at com.engine.core.cfg.CommandExecutorImpl.execute(CommandExecutorImpl.java:37)

at com.engine.core.cfg.CommandExecutorImpl.execute(CommandExecutorImpl.java:42)

at com.engine.msgcenter.service.impl.MsgHomepageServiceImpl.getMsgCount(MsgHomepageServiceImpl.java:37)

at com.engine.msgcenter.web.MsgHomepageAction.getMsgCount(MsgHomepageAction.java:243)

at sun.reflect.GeneratedMethodAccessor217.invoke(Unknown Source)

at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)

at java.lang.reflect.Method.invoke(Method.java:498)

**（2）解决方案：**

经过与华为毕昇JDK工程师沟通，分析该问题可能是由于社区版JDK的一个锁问题导致，该问题在ARM环境下会更容易出现，毕升JDK对这个问题进行了修复，该修改没有提交到上游社区，所以社区版版本会存在该问题。

根据华为工程师的建议，现场运维人员在7月5日晚更新了各个环境的JDK至毕升JDK1.8版本。通过对7月6日的堆栈信息观察，由该问题导致的cpu高和卡顿问题已经解决。

## ISSUE2021070601

**（1）现象描述：**

07-06号早上7点24左右，再次出现卡顿和CPU使用率过高的情况。

**（2）分析过程：**

1. 获取堆栈后发现只有几个处于Runnable状态，其中有2个业务线程堆栈如下：

"WeaverHttp&NULL SESSION&07:25:23&/mobilemode/mobile/server.jsp" id="288" CPU Time=5469053860

java.lang.Thread.State: RUNNABLE

at sun.misc.Unsafe.defineClass(Native Method)

at sun.reflect.ClassDefiner.defineClass(ClassDefiner.java:63)

at sun.reflect.MethodAccessorGenerator$1.run(MethodAccessorGenerator.java:399)

at sun.reflect.MethodAccessorGenerator$1.run(MethodAccessorGenerator.java:394)

at java.security.AccessController.doPrivileged(Native Method)

at sun.reflect.MethodAccessorGenerator.generate(MethodAccessorGenerator.java:393)

at sun.reflect.MethodAccessorGenerator.generateMethod(MethodAccessorGenerator.java:75)

at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:53)

at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)

at java.lang.reflect.Method.invoke(Method.java:498)

at org.apache.commons.beanutils.PropertyUtilsBean.invokeMethod(PropertyUtilsBean.java:1773)

at org.apache.commons.beanutils.PropertyUtilsBean.getSimpleProperty(PropertyUtilsBean.java:1132)

at org.apache.commons.beanutils.PropertyUtilsBean.getNestedProperty(PropertyUtilsBean.java:686)

at org.apache.commons.beanutils.PropertyUtilsBean.getProperty(PropertyUtilsBean.java:715)

at org.apache.commons.beanutils.PropertyUtils.getProperty(PropertyUtils.java:290)

at net.sf.json.JSONObject.\_fromBean(JSONObject.java:733)

……

"Thread-60" id="210" CPU Time=4580332140

java.lang.Thread.State: RUNNABLE

at sun.management.ThreadImpl.getThreadInfo1(Native Method)

at sun.management.ThreadImpl.getThreadInfo(ThreadImpl.java:183)

at sun.management.ThreadImpl.getThreadInfo(ThreadImpl.java:144)

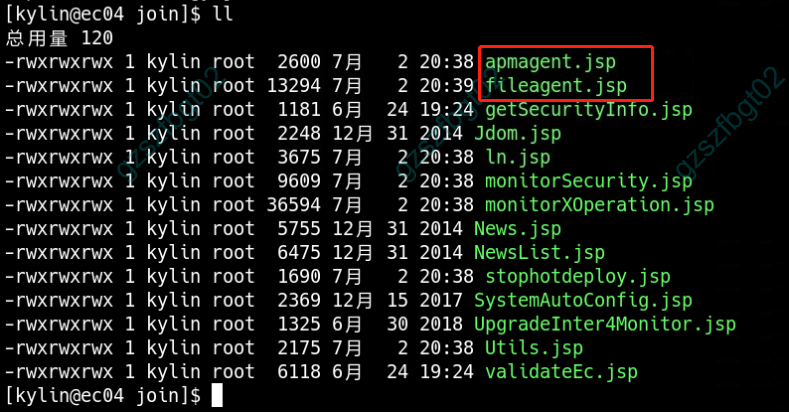
at weaver.monitor.monitor.SysThreadMonitor.getMonitorInfo(SysThreadMonitor.java:32)

at weaver.monitor.threads.SysThreadThread.doThreadWork(SysThreadThread.java:15)

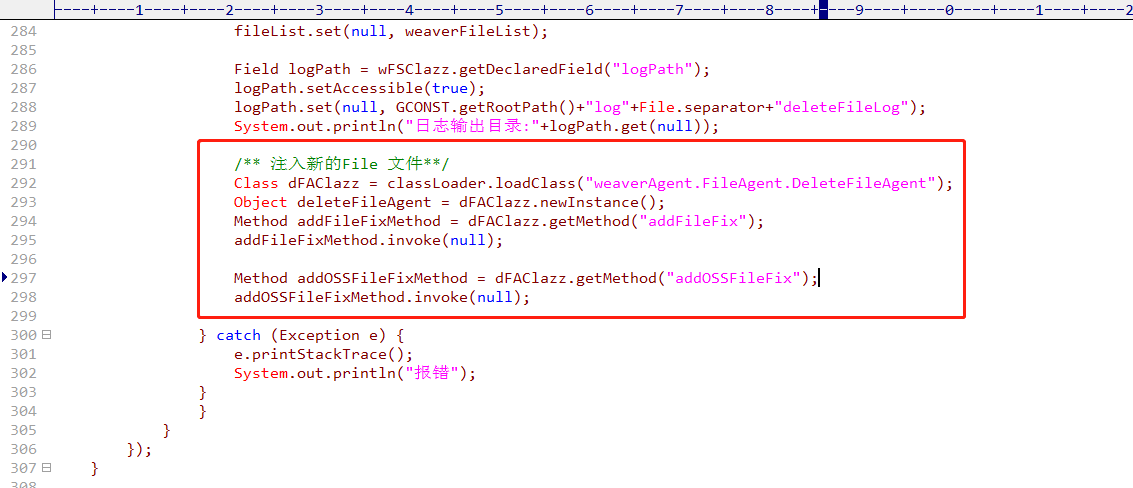
at weaver.system.ThreadWorkTimer.run(ThreadWorkTimer.java:67)

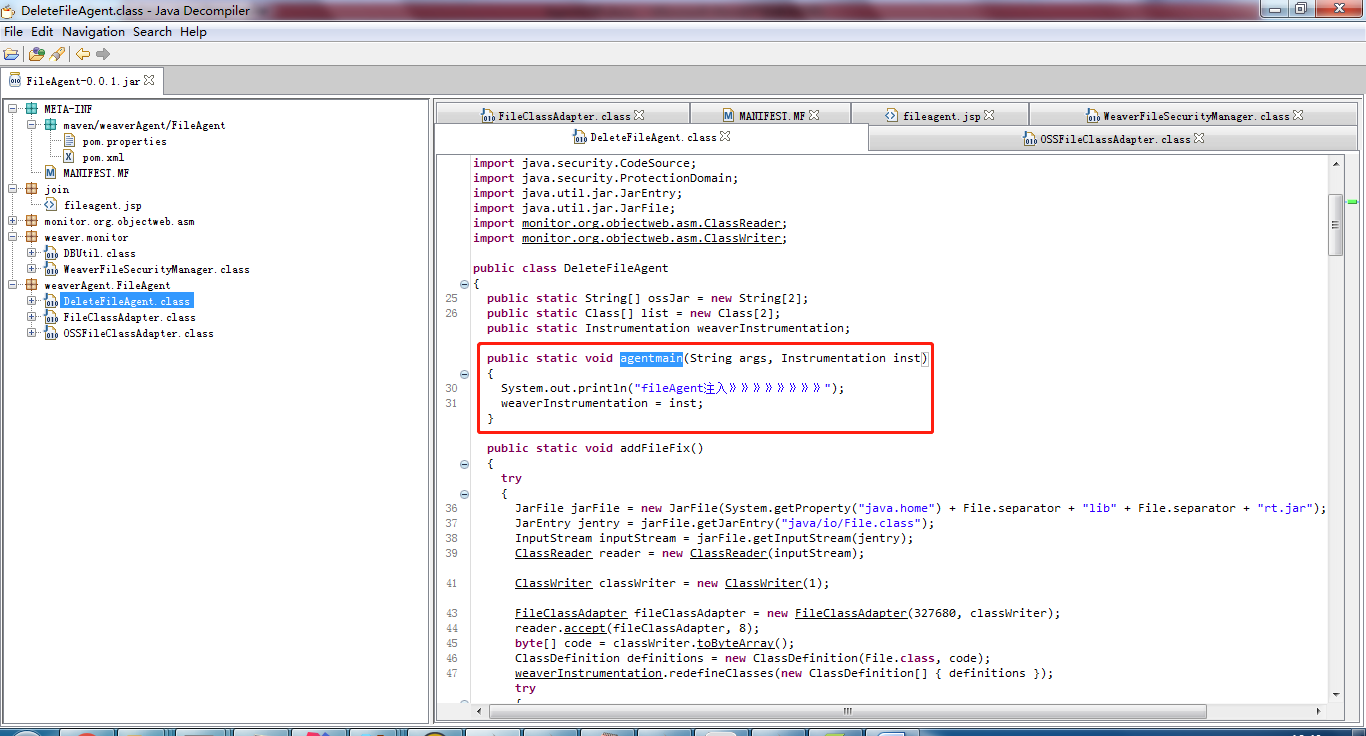
1. 通过对JDK调用进行跟踪和Perf的分析发现，defineClass方法耗时并占用CPU资源，其中热点方法是transfromClassFile。
2. 通过和开发商交流，了解到应用中有一个运维系统Agent会Attach到运行中的JVM中进行监控，defineClass的耗时和CPU使用率过高的问题大概率是由该操作导致。
3. 运维系统Agent分析

应用中的fileagent.jsp和apmagent.jsp如图：

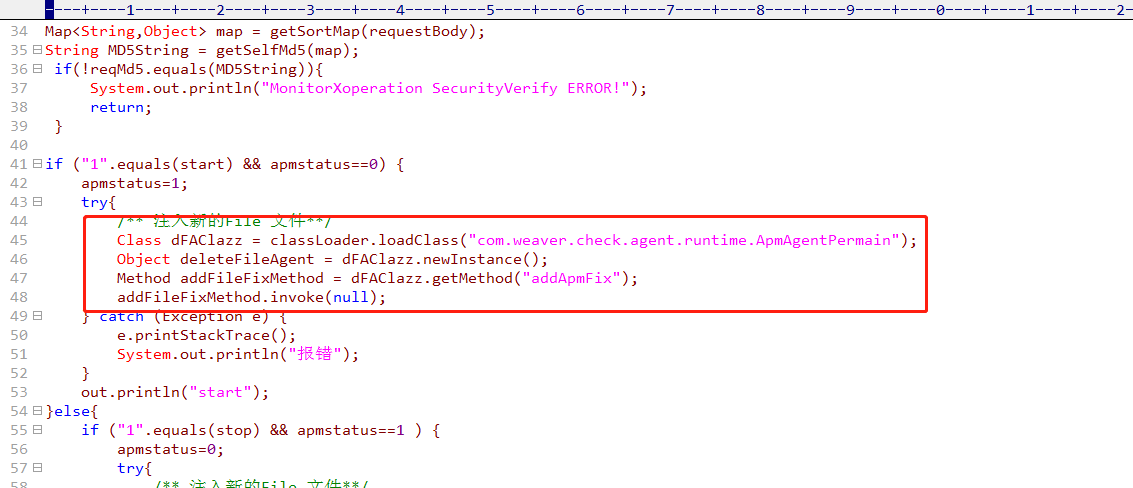


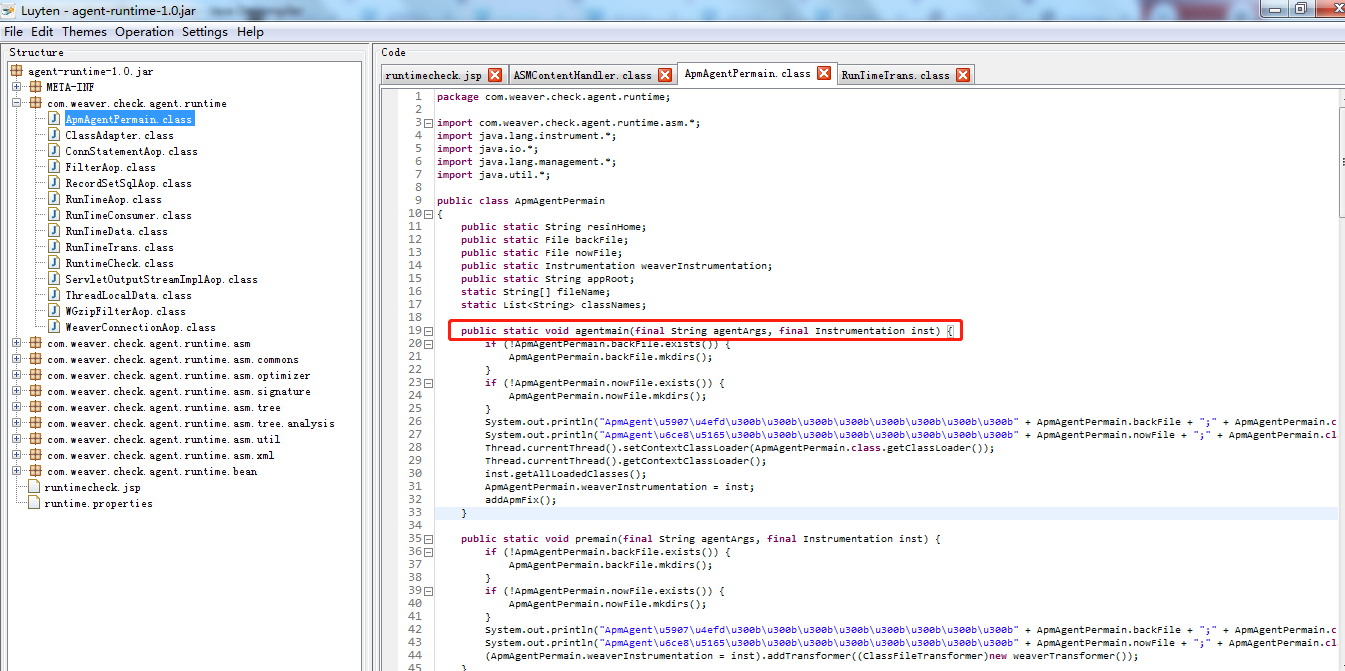
fileagent.jsp关键代码，如下图所示：

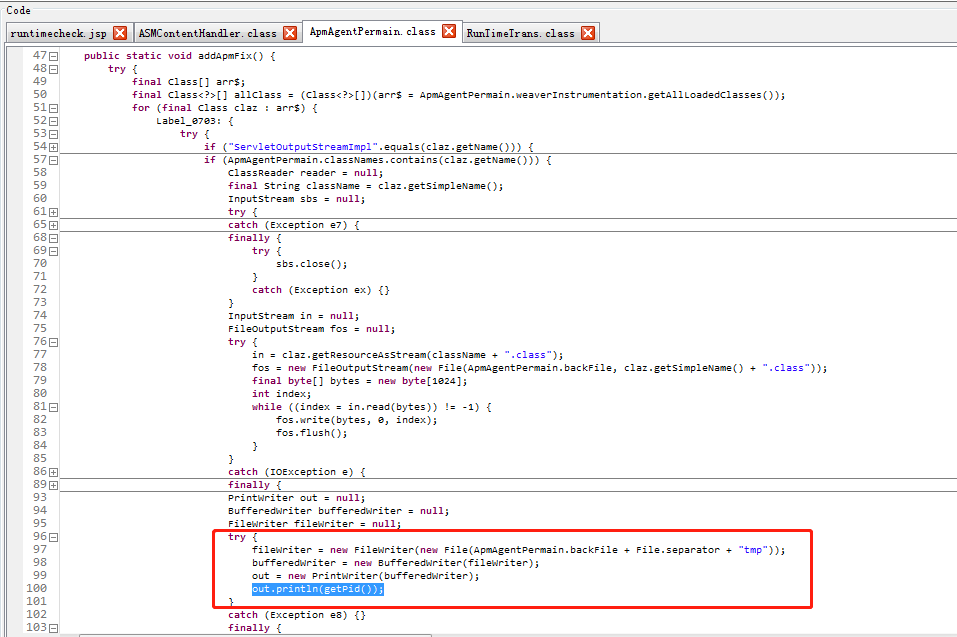




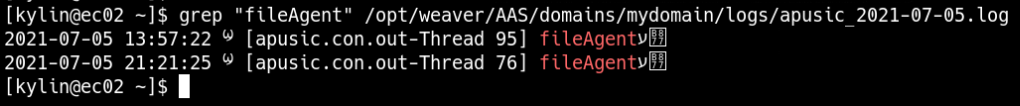
apmagent.jsp关键代码如下图所示：

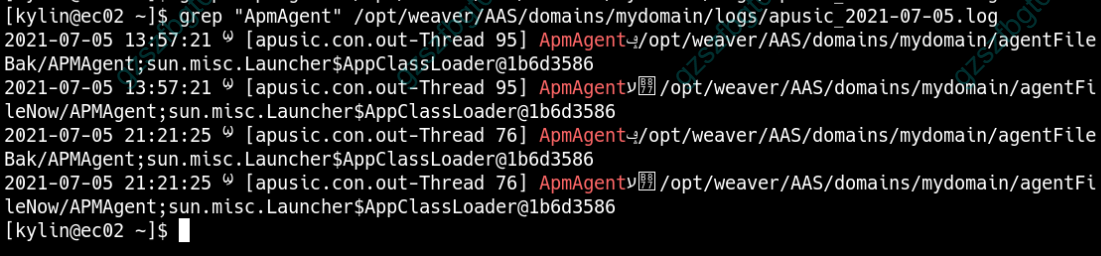






启动的时候输出attach的日志如下图所示：





**（3）解决办法：**

暂停运维系统的attach。

# 根因分析与结论

## 问题发现过程

使用sudo perf record -F 99 -a -g -p PID sleep 30 对所有CPU（-a）进行call stacks（-g）采样，采样频率为99 Hertz（-F 99），即每秒99次，持续30秒（sleep 30）,然后将采样结果生成火焰图：

图片包含 户外, 游戏机, 街道, 交通

描述已自动生成

从火焰图上发现JVM的transformClassFile方法耗时占用最大，达到70%以上。

结合JDK源码

<https://github.com/openjdk/jdk/blob/master/src/java.instrument/share/native/libinstrument/JPLISAgent.c>

表格

描述已自动生成

transformClassFile是JVMTI的callback，说明JVM中加载了agent，才会调用到这个callback。

AAS的启动脚本中没有挂载agent，但从应用启动日志中发现有agent被动态加载：

文本

描述已自动生成

通过分析应用源码，发现了应用中自定义的agent：

图形用户界面, 文本, 应用程序

描述已自动生成

通过和应用厂商交流，了解到运维系统的动态agent会动态attach到运行中的JVM中进行监控，基本确定了问题由动态agent引起的。

## 根因分析

根据JDK的文档

<http://openjdk.java.net/groups/hotspot/docs/RuntimeOverview.html>

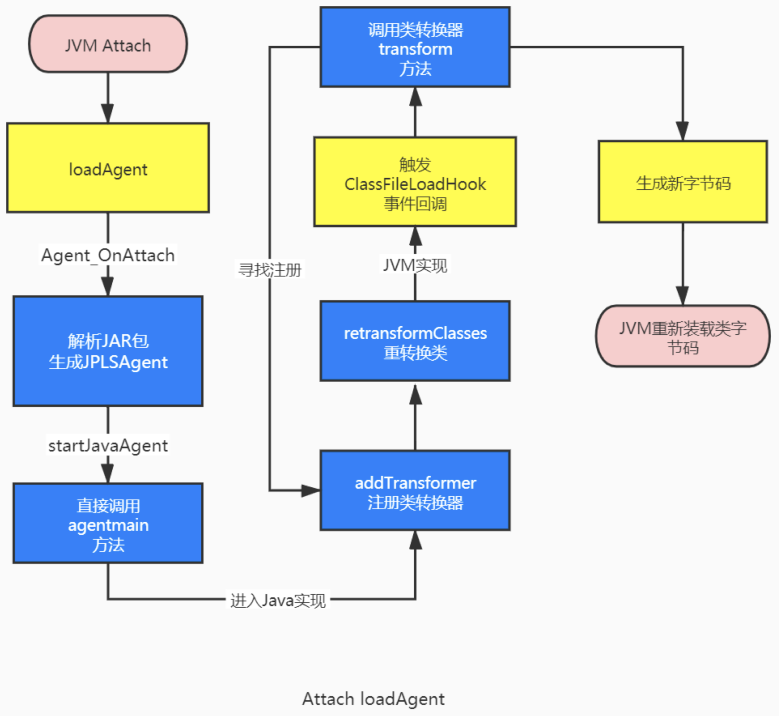
"VM Operations and Safepoints"

Safepoint是应用程序执行的一种状态，在这种状态下，JVM 可以完全访问对象的所有引用。

Safepoint意味着所有线程到达某个执行点后，全部暂停，然后执行 JVM 操作，之后所有的线程再恢复执行，也就是我们常说的stop-the-world.

JVM 有些操作要求所有线程到达一个Safepoint才能执行，最常见的操作是 GC。

除了GC，还有一些JVM操作会需要Safepoint，包括通过 JVMTI的修改操作。 Java Instrument 导致的 Agent 加载以及类的重定义，需要修改栈上和这个类相关的信息，所以需要stop-the-world。



需要Safepoint的JVM操作列表定义在JDK的vmOperation.hpp中：

<https://github.com/openjdk/jdk/blob/master/src/hotspot/share/runtime/vmOperation.hpp>

#define VM\_OPS\_DO(template) \

template(None) \

template(Cleanup) \

template(ThreadDump) \

template(PrintThreads) \

template(FindDeadlocks) \

template(ClearICs) \

template(ForceSafepoint) \

template(ForceAsyncSafepoint) \

template(DeoptimizeFrame) \

template(DeoptimizeAll) \

template(ZombieAll) \

template(Verify) \

template(HeapDumper) \

template(DeoptimizeTheWorld) \

template(CollectForMetadataAllocation) \

template(GC\_HeapInspection) \

template(GenCollectFull) \

template(GenCollectFullConcurrent) \

template(GenCollectForAllocation) \

template(ParallelGCFailedAllocation) \

template(ParallelGCSystemGC) \

……

## 结论

经过金蝶天燕现场运维人员及金蝶天燕后台研发人员对系统运行数据（JStack堆栈、Perf Data及火焰图、系统日志等）的综合分析，并查阅JVM代码及实现机制后，又对应用代码进行了反查，除前述2.1到2.4的问题外，还有一个非常关键的问题在于运维系统通过Instrument Agent方式动态挂载到了生产系统中，并引发了JVM中的Safepoint，导致了STW（stop-the-world）操作，进而引起一系列的线程堵塞与CPU高、系统卡顿现象。通过与应用厂商沟通，确认了应用中隐藏的agent挂载操作。

系统目前主要底层问题已基本排查定位清晰并给出了建议方案，但不排除后续应用如优化不够，依然可能出现堵塞与性能问题，建议对系统所有功能进行全面并发压测尤其是高频使用功能，同时需谨慎考虑运维系统监控生产环境的机制与方式应尽可能避免对生产环境产生高负载压力甚至堵塞。

# 后续服务保障建议

1. 提供现场驻场服务并提升现场人员素质，实时观测系统行为，及时发现系统问题，于4小时内给出解决方案并进行优化；
2. 提供专属的服务升级机制，当现场驻场人员未能及时解决问题或有具体可行的方案时，升级为总部服务；
3. 完善系统巡检机制，针对应用系统中的进程外挂载等可能会对资源造成额外开销的操作，建立起扫描、记录、跟踪以及运维巡检的机制。
4. 完善生产环境升级以及补丁的严格管控机制，对系统升级、漏洞补丁等生产环境变更进行完整、详实的记录，以确保每次系统变更的可追溯。
5. 从项目管控的角度来看，希望项目能有清晰详尽的部署架构图，明确给出协议和组件交互关系的说明文档，以及前后环境对比差异，帮助项目组加快从全局分析及前后差异分析角度定位判断问题；
6. 建议对应用系统进行全面压测，各方协同保障问题排查与分析解决。