**X网宕机分析总结V1.2**

目录

[1. 总体情况： 2](#_Toc364323377)

[1.1 部署情况 2](#_Toc364323378)

[1.2 故障描述 3](#_Toc364323379)

[2. 故障分析 4](#_Toc364323380)

[2.1 故障初步分析 4](#_Toc364323381)

[2.2 故障深入分析 8](#_Toc364323382)

[3. 访问日志分析 13](#_Toc364323383)

[3.1 正常访问 14](#_Toc364323384)

[3.2 SQL注入扫描 14](#_Toc364323385)

[3.3 恶意扫描 14](#_Toc364323386)

[3.4 搜索引擎抓取 15](#_Toc364323387)

[3.5 存活检测 16](#_Toc364323388)

[4. 解决方案 16](#_Toc364323389)

[4.1 修改程序（已实施） 16](#_Toc364323390)

[4.2 子系统隔离部署（已实施） 16](#_Toc364323391)

[4.3 IP黑名单机制（已实施） 17](#_Toc364323392)

[4.4 Robots文件定义（未实施） 17](#_Toc364323393)

[5. 测试情况 17](#_Toc364323394)

[6. 实际监控情况 18](#_Toc364323395)

[7. 存在隐患 21](#_Toc364323396)

[8. 总结 22](#_Toc364323397)

# 总体情况：

本节描述了XX网（以下简称XX网）的环境部署、软硬件情况以及故障的描述情况。

## 1.1 部署情况

* XX网中间件所在OS的相关信息为：

|  |
| --- |
| OS 名称: Microsoft(R) Windows(R) Server 2003, Enterprise Edition  OS 版本: 5.2.3790 Service Pack 2 Build 3790  系统制造商: IBM  系统型号: eserverxSeries 366-[8863CSC]-  系统类型: X86-based PC  处理器: 安装了 4 个处理器。  [01]: x86 Family 15 Model 4 Stepping 9 GenuineIntel ~3169 Mhz  [02]: x86 Family 15 Model 4 Stepping 9 GenuineIntel ~3169 Mhz  [03]: x86 Family 15 Model 4 Stepping 9 GenuineIntel ~3169 Mhz  [04]: x86 Family 15 Model 4 Stepping 9 GenuineIntel ~3169 Mhz  物理内存总量: 3,327 MB  页面文件: 最大值: 5,223 MB  网卡: [01]: Broadcom NetXtreme Gigabit Ethernet  IP 地址  [01]: 192.168.1.20  [02]: Broadcom NetXtreme Gigabit Ethernet  IP 地址  [01]: 192.168.1.18 |

* X网数据库所在OS的相关信息：

|  |
| --- |
| OS 名称: Microsoft(R) Windows(R) Server 2003, Enterprise Edition  OS 版本: 5.2.3790 Service Pack 2 Build 3790  OS 制造商: Microsoft Corporation  OS 配置: 独立服务器  OS 构件类型: Multiprocessor Free  产品 ID: 69813-650-4910687-45626  系统制造商: IBM  系统型号: IBM System x3650 -[7979RG7]-  系统类型: X86-based PC  处理器: 安装了 4 个处理器。  [01]: x86 Family 6 Model 15 Stepping 6 GenuineIntel ~1995 Mhz  [02]: x86 Family 6 Model 15 Stepping 6 GenuineIntel ~1995 Mhz  [03]: x86 Family 6 Model 15 Stepping 6 GenuineIntel ~1995 Mhz  [04]: x86 Family 6 Model 15 Stepping 6 GenuineIntel ~1995 Mhz  BIOS 版本: IBM - 1000  物理内存总量: 4,095MB  页面文件: 最大值: 5,978 MB |

* X网数据库为Oracle 10.2.0.4.0，单实例结构。
* X网中间件为IBM Websphere V6.1（以下简称WAS）其中HeapSize初始为512MB，最大为1024MB，使用JNDI的Oracle连接池，连接数最大150个。

## 1.2 故障描述

首先介绍一下Y系统，该系统是X网下面的一个子系统，和X网同属一个应用程序，使用一个Web根路径。

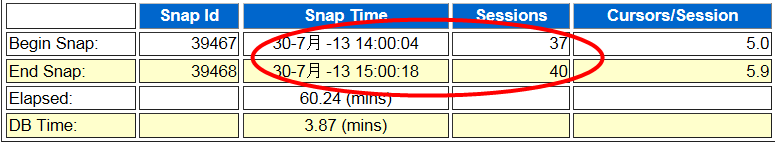
据运维服务人员反馈，自从Y子系统上线以来，截止8月6号，X网出现了超过4次的宕机，而且几次的宕机时间都在周二的15:10左右，经过询问，运维服务人员判断宕机的过程以及执行的操作步骤如下：

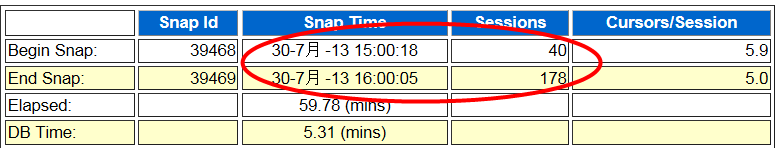
* SDZC网站会间隔性的发起对X网可用性的检测，检测方法为每隔5分钟对网站的页面发起HTTP请求，若请求成功则判断网站存活，否则认为网站已经不再存活从而发起警告。
* 运维服务人员收到警告后，首先会访问X网的页面，发现网页的确无法打开；然后查看后台的sysout和syserr日志发现报OutOfMemory错误，从而认为中间件内存溢出，于是将X网的生产系统的HTTP服务切换至灾备系统，然后对生产系统的WAS中间件进行重启。

从以上的描述可以看出，运维服务人员判断宕机的因素有两个：

* 服务器拒绝服务，表现为页面无法打开。
* 中间件报OutOfMemory内存溢出错误。

在整个过程中，数据库服务器出现了一个值得注意的现象，8月6号的AWR显示，在宕机前(15:10之前)，数据库的session数量维持在40左右，宕机后(15:10之后)，session数量突然上升到180左右，通过监测发现，X网的正常的数据库session数量应该维持在30-40之间。





# 故障分析

## 2.1 故障初步分析

对于宕机，或者中间件处于什么状态才认为是宕机，我们认为应该是首先中间件拒绝服务，进而生成DUMP文件和CORE文件；如果是OutOfMemory错误，则JAVA进程消失，否则JAVA进程可能会继续存活。但是实际出现的几次“宕机”，并没有人为确认JAVA进程是否消失就对中间件进行了重启。所以我们认为，在出现故障时，应该具有以下两个特点：

* X网WAS中间件拒绝服务，表现为网页打不开，出现500或者最终找不到网页。
* WAS中间件生成DUMP文件。

对于后台报错信息的分析，首先是sysout和syserror日志文件，如下是日志的片段：

|  |
| --- |
| **[13-8-6 16:02:13:953 CST] 00000259 SystemErr R Caused by: java.lang.OutOfMemoryError**  at java.lang.Class.getDeclaredFieldImpl(Native Method)  at java.lang.Class.getDeclaredField(Class.java:501)  at java.io.ObjectStreamClass.getDeclaredSUID(ObjectStreamClass.java:1638)  at java.io.ObjectStreamClass.access$600(ObjectStreamClass.java:104)  at java.io.ObjectStreamClass$2.run(ObjectStreamClass.java:464)  at java.security.AccessController.doPrivileged(AccessController.java:197)  at java.io.ObjectStreamClass.<init>(ObjectStreamClass.java:456)  at java.io.ObjectStreamClass.lookup(ObjectStreamClass.java:351)  at java.io.ObjectStreamClass.initNonProxy(ObjectStreamClass.java:587)  at java.io.ObjectInputStream.readNonProxyDesc(ObjectInputStream.java:1577)  at java.io.ObjectInputStream.readClassDesc(ObjectInputStream.java:1491)  at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:1724)  at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1330)  at java.io.ObjectInputStream.readObject(ObjectInputStream.java:364)  at com.ibm.ws.naming.util.Serialization$1.run(Serialization.java:106)  at com.ibm.ws.security.util.AccessController.doPrivileged(AccessController.java:118)  at com.ibm.ws.naming.util.Serialization.deserializeObject(Serialization.java:102)  at com.ibm.ws.naming.util.Helpers.processSerializedObjectForLookupExt(Helpers.java:795)  at com.ibm.ws.naming.util.Helpers.processSerializedObjectForLookup(Helpers.java:705)  at com.ibm.ws.naming.jndicos.CNContextImpl.cacheLookup(CNContextImpl.java:3626)  at com.ibm.ws.naming.jndicos.CNContextImpl.doLookup(CNContextImpl.java:1880)  at com.ibm.ws.naming.jndicos.CNContextImpl.doLookup(CNContextImpl.java:1862)  at com.ibm.ws.naming.jndicos.CNContextImpl.lookupExt(CNContextImpl.java:1552)  at com.ibm.ws.naming.jndicos.CNContextImpl.lookup(CNContextImpl.java:1354)  at com.ibm.ws.naming.util.WsnInitCtx.lookup(WsnInitCtx.java:172)  at javax.naming.InitialContext.lookup(InitialContext.java:363)  at soft.db.DBAdapter.connect(DBAdapter.java:266)  at soft.db.DBAdapter.connect(DBAdapter.java:195)  at soft.iweb.bean.DbBean.init(DbBean.java:612)  at soft.iweb.bean.DbBean.<init>(DbBean.java:594)  at soft.iweb.bean.UserBean.<init>(UserBean.java:432)  at soft.iweb.bean.admin.template.<init>(template.java:57)  **at com.ibm.\_jsp.\_view\_5F\_c.\_jspService(\_view\_5F\_c.java:203)**  at com.ibm.ws.jsp.runtime.HttpJspBase.service(HttpJspBase.java:87)  at javax.servlet.http.HttpServlet.service(HttpServlet.java:856)  at com.ibm.ws.webcontainer.servlet.ServletWrapper.service(ServletWrapper.java:1095)  at com.ibm.ws.webcontainer.servlet.ServletWrapper.service(ServletWrapper.java:1036)  at com.ibm.ws.webcontainer.filter.WebAppFilterChain.doFilter(WebAppFilterChain.java:118)  at com.ibm.ws.webcontainer.filter.WebAppFilterChain.\_doFilter(WebAppFilterChain.java:87)  at com.ibm.ws.webcontainer.filter.WebAppFilterManager.doFilter(WebAppFilterManager.java:832)  at com.ibm.ws.webcontainer.filter.WebAppFilterManager.doFilter(WebAppFilterManager.java:679)  at com.ibm.ws.webcontainer.servlet.ServletWrapper.handleRequest(ServletWrapper.java:565) |

日志中报出了让运维服务人员判断WAS宕机的关键错误“java.lang.OutOfMemoryError”，从日志分析来看，是由于一个查看文章的页面view.jsp的处理请求导致，这个请求提交了一个数据库查询请求，中间件在处理这个数据库查询请求的时候Heap空间不够导致内存溢出。对WAS的DUMP文件首先委托专人进行了分析，结论为内存中存在大量的File对象，怀疑是这个对象没有被及时GC导致，可能存在针对该对象的内存泄露。于是我们针对资源泄露的问题进行了仔细的代码复查，但是未发现值得怀疑的可能导致内存泄露的问题代码。

在宕机时间区间内，基于以下两点因素：

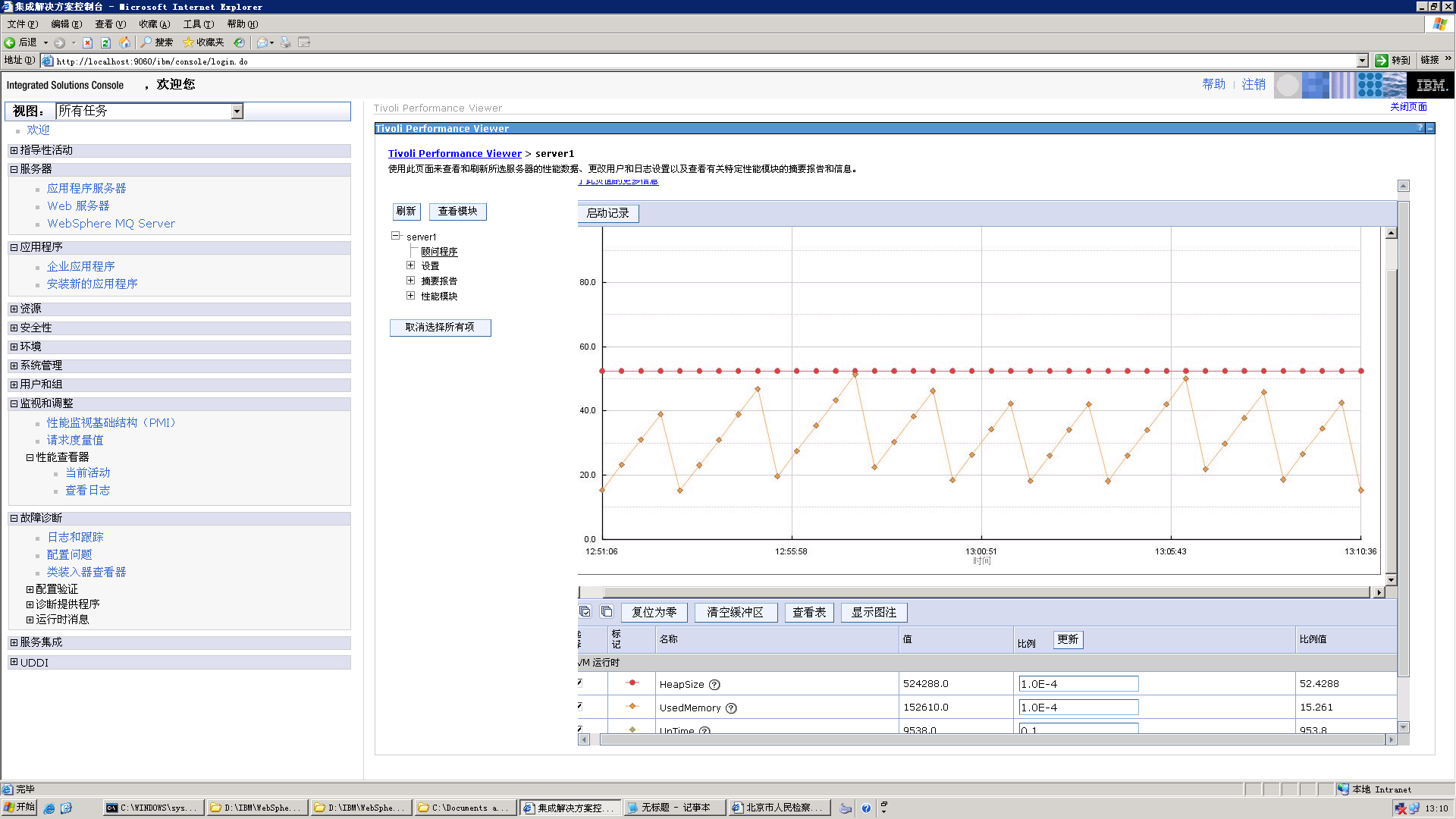
* 数据库出现了大量异常的连接会话数；
* WAS-HTTP Server访问日志中大量的请求。

我们怀疑是由于大量的HTTP请求带来了高并发的访问压力超出中间件吞吐量导致内存溢出，推测过程如下：

* 在高并发的访问压力下，在某时刻中间件的Heap内存到达一个临界点，此时堆内存还未溢出。
* 在到达临界点的下一个时刻，一个view.jsp页面的请求导致堆内存的使用超出临界点由于view.jsp涉及到数据库查询，该查询需要一定的内存消耗，从而诱发了OutOfMemoryError（见上文中的日志红色字体部分），WAS的JAVA进程停止服务。

我们推测，view.jsp是最终触发内存溢出的原因，却不一定是导致Heap内存使用到达临界点的原因。其实如果有这段时间的WAS监控日志，可以很好的排除这个问题，但是由于生产上的监控使用的都是默认配置，只记录20分钟的WAS内核信息而且还不能持久化，所以没有拿到当时的监控记录。

为了验证是否是这个原因导致内存溢出，我们统计了8月6号的WAS-HTTP Server访问日志的访问量，并与8月8号上午9点至10点以高于该访问量的压力在测试环境上对X网进行了压力测试，在压力测试下，JVM堆内存的曲线如下：

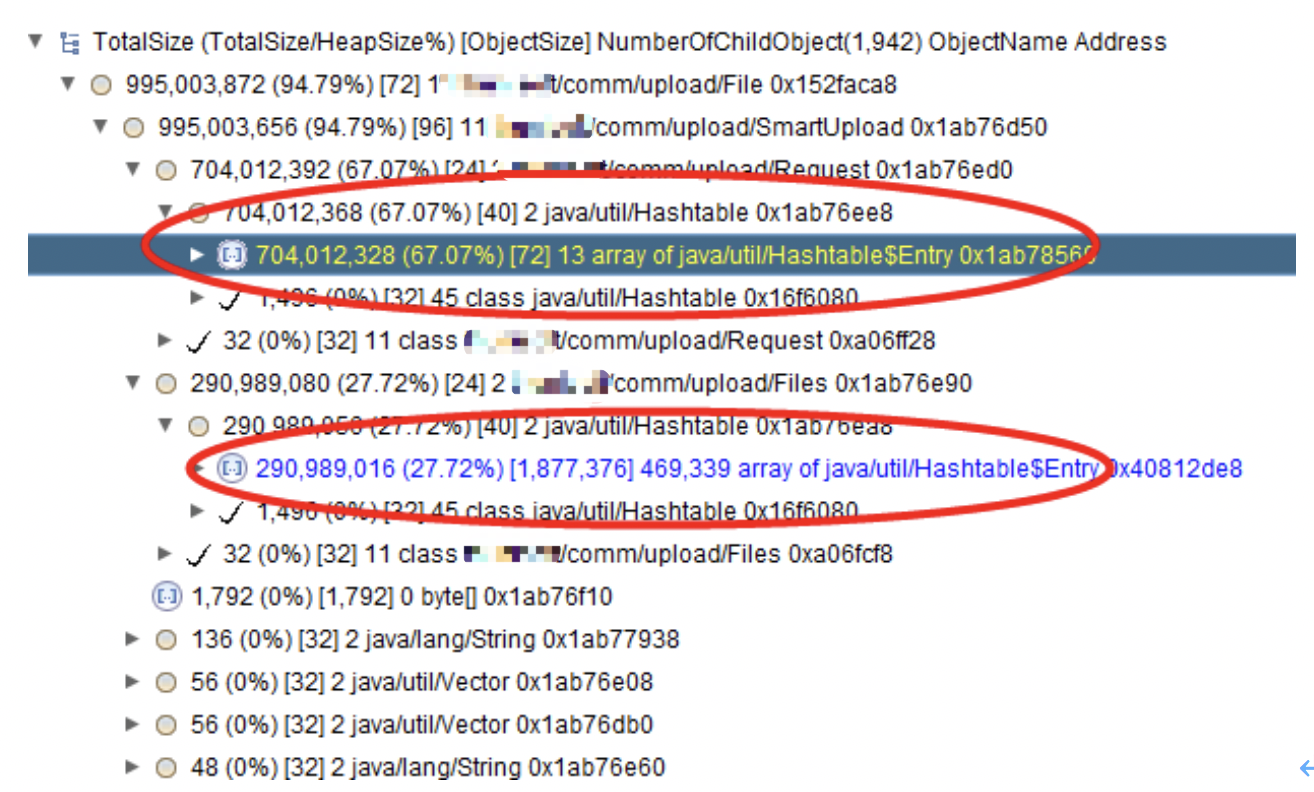


由堆内存变化曲线可以看出，JVM的堆使用从未超出初始值512M，有时达到512MB,但是GC很快进行内存回收，总体来看GC工作状态良好。从WAS-HTTP Server的访问日志分析来看，虽然个别IP地址出现了大量的访问请求，但是还远没有超过压力测试的访问压力，由会话统计显示，压力测试下的活动会话数已经超过了4000。因此可以断定高并发的请求不是宕机的主因。

## 2.2 故障深入分析

针对提出的假设被排除，而且没有拿到当时的WAS监控日志，代码复查未发现可能存在内存泄露的问题代码，所以我们在8月8号上午再次对WAS的DUMP日志进行了详细的分析（蒋拯强和张进炎负责）。这次分析有很大的收获，如，通过对“heapdump.20130806.151911.564.0003.phd”文件进行分析，发现内存中引起问题的不是诸多的File对象内存泄露，而是“SmartUpload”对象占用过多内存。所谓内存泄露，是指内存超出应用程序的控制，游离在可控的范围之外导致无法被回收、重用。而该DUMP显示，该对象的内存是可控的，只是该对象占用的空间太大而已。

该对象的某个实例在的内存占用情况如下：



发现某个HashTable中的Entry占据了67.7%的堆内存容量，这些Entry和其他对象的依赖关系如下图所示：

由图可见，该HashTable对象是由Request类所持有，查找该类可以发现，这是X网网站自定义的一个类，该类和Servlet中的Request完全不一样，只是为了暂时持有一下某些对象以及值，综合SmartUpload来看，在Request中操作HashTable对象的代码如下：

protected void putParameter(String name, String **value**)

{

if(name == null)

throw new IllegalArgumentException("The name of an element cannot be null.");

if(m\_parameters.containsKey(name))

{

**Hashtable values = (Hashtable)m\_parameters.get(name);**

**values.put(new Integer(values.size()), value);**

} else

{

**Hashtable values = new Hashtable();**

**values.put(new Integer(0), value);**

**m\_parameters.put(name, values);**

m\_counter++;

}

该段代码实际上是将POST到后台的数据解析出来非File域的值进行缓存，缓存的数据来自于SmartUpload的m\_totalBytes数组，对于方法putParameter的参数value，最终是设置到名为HashTable中的变量中，在SmartUpload中，调用putParameter的代码（蓝色字体）为：

public void upload()

throwsSmartUploadException, IOException, ServletException

{

//此处省略部分代码

boolean found = false;

String dataHeader = new String();

String fieldName = new String();

String fileName = new String();

String fileExt = new String();

String filePathName = new String();

String contentType = new String();

String contentDisp = new String();

String typeMIME = new String();

String subTypeMIME = new String();

booleanisFile = false;

***m\_totalBytes = m\_request.getContentLength();***

if(m\_request.getContentLength()<0){

m\_totalBytes=0;

}else{

m\_totalBytes=m\_request.getContentLength();

}

***m\_binArray = new byte[m\_totalBytes];***

m\_totalBytes = m\_request.getContentLength();

//此处也省略部分代码

String value = new String(m\_binArray, m\_startData, (m\_endData - m\_startData) + 1);

m\_formRequest.***putParameter***(fieldName, value);

其中代码“m\_totalBytes = m\_request.getContentLength();”意为在上传文件时，从request中获取contentLength（可以简单的认为是上传文件大小），然后申请一个同样大小的字节数组用来保存文件内容，最后在某个条件下以该数组生成一个字符串，放入Request对象中的Map中。

这样便会存在问题，首先，如果上传的文件超大，如达到几G甚至几十G的容量，那么该段代码便会申请相应大小的内存来存放该文件的数据，这样就会导致内存溢出，因为根本没有这么大的堆内存（WAS的堆内存最大才1G）。通常正常的上传文件方法是要么对文件做大小限制，要么在上传文件时分段读取然后进行磁盘缓存。但是这段代码没有做任何限制，因此我们认为这可能就是导致WAS内存溢出的主因。为了验证这个结论，我们在8月8号11点到15点间对文件上传功能进行了测试。

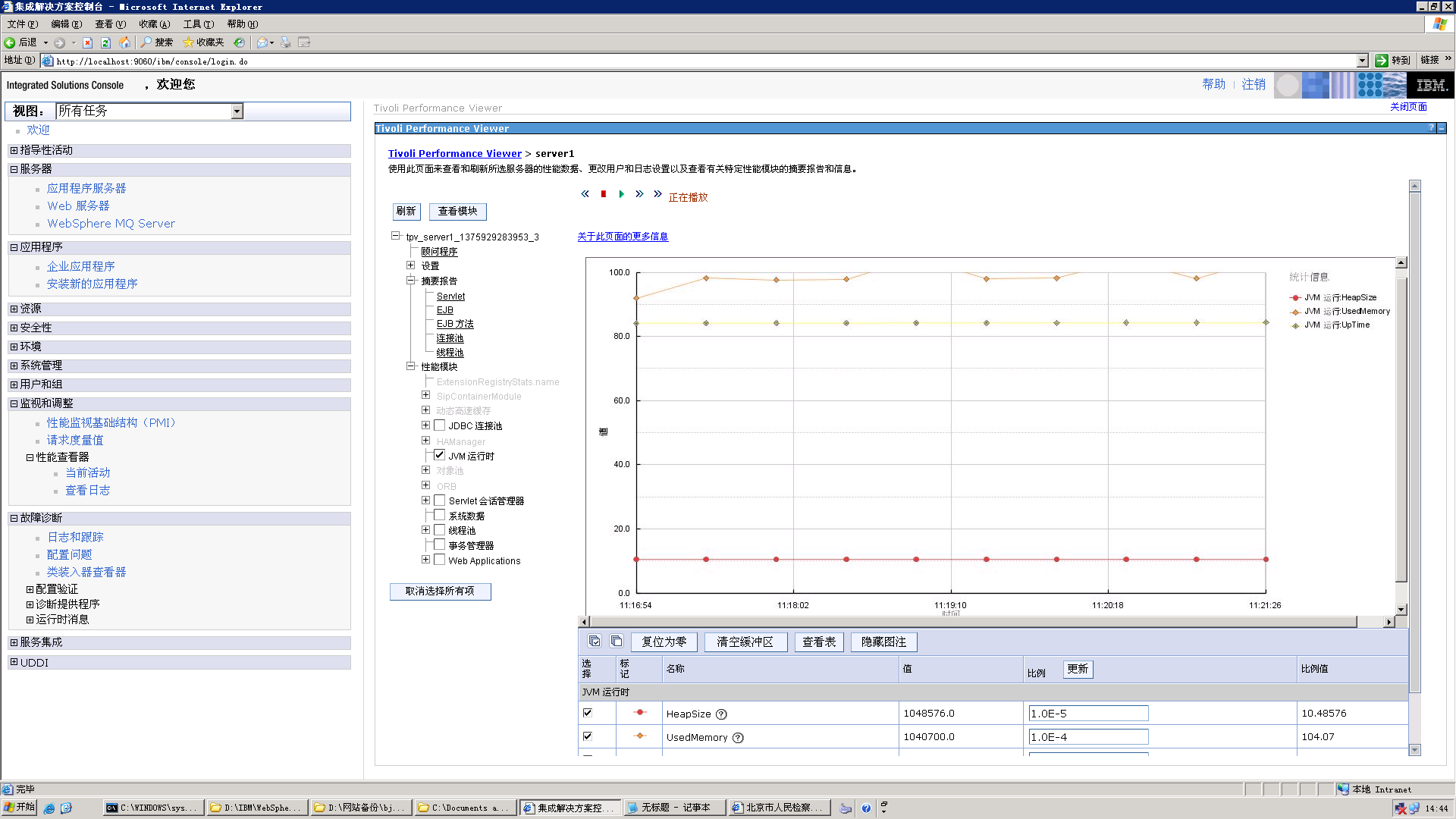
测试方法：

1. X网的Y页面上传一个2.8GB的压缩包。



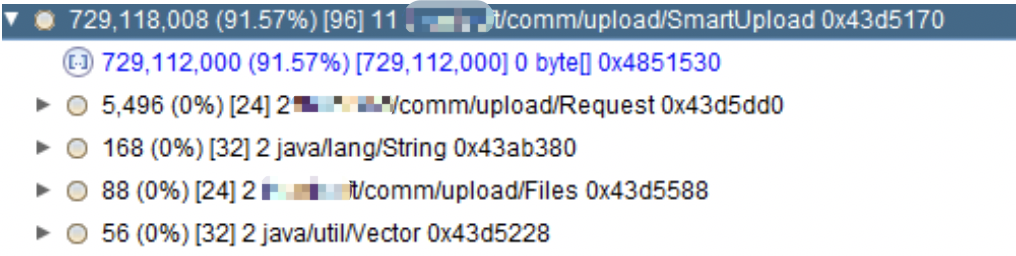
1. 开启WAS的内存监控。

当执行这个大文件上传时，WAS的JVM堆内存的变化情况如下（红色曲线表示）：

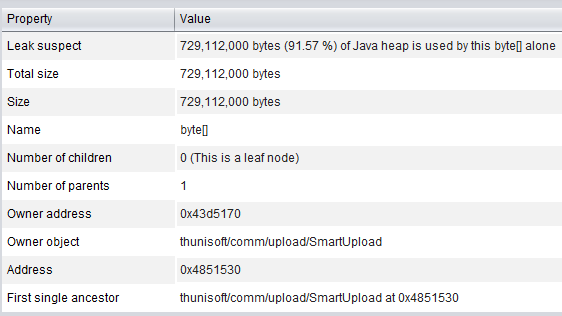


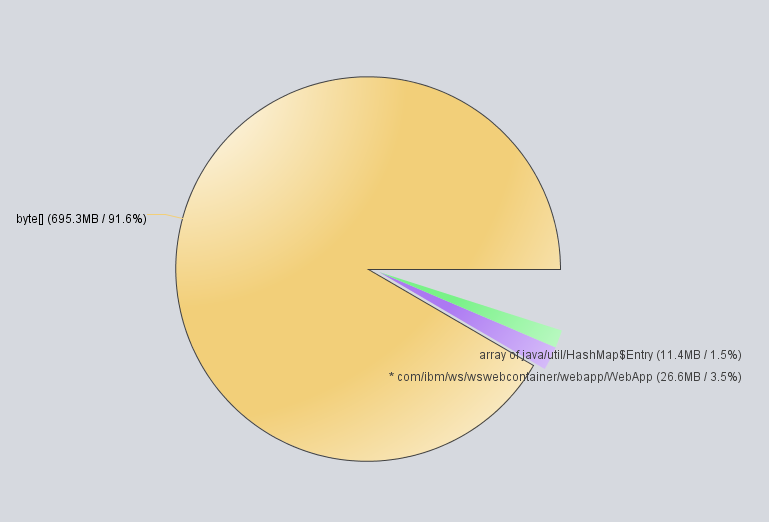
可以看到，堆的使用立刻飙升到90%以上甚至到达1024的极限。此时继续再发起一些文件上传的操作发现已经打不开页面，进行普通的页面访问操作也极其缓慢，最终报500错误，此时中间件已经拒绝服务，后台出现DUMP文件，但是还没有报OOM的错误，java进程还继续存活。为了判断此时是否是宕机的临界点，我们又发起了普通页面请求的压力测试，发现WAS仍然拒绝服务，未出现内存溢出问题。

我们对DUMP文件进行了分析：



发现这个对象的实例占用了91.75%的堆内存，而空间的过度占用是由它的一个数组导致，这个数组的详细信息如下：





该数组占用了Java堆内存的91.75%，该数组即是本文前面所描述的m\_totalBytes数组，这个对象的数组引起了堆内存的过度占用，导致了OOM。为了验证该结论，我们深入分析了HTTP访问日志，以确认日志中是否存在大量的文件上传动作。

# 访问日志分析

在寻找宕机原因时，我们对WAS-HTTP Server的访问日志也进行了详细的分析，进而将这些日志的访问动作类型归结为以下几类（根据访问量排序）。

## 3.1 正常访问

这些是正常的HTTP访问，也就是普通的浏览者或者子系统使用者的访问。在这些访问者中，有来自市检内网的用户，也有来自于互联网的用户，这二者占访问总量的70%。

## 3.2 SQL注入扫描

外网的网站都要不时的被SDZC的SQL注入扫描器进行扫描，以发现漏洞反馈给网站运维部门进行防范，从而减少网站被注入的风险。进行SQL注入扫描要频繁的访问同一个URL多次，变换不同的条件猜解，所以出现了较多该类IP的访问动作。经过统计，和SDZC确认了一些TOP IP是该部门所属的IP，还有一些IP属于恶意IP。

## 3.3 恶意扫描

这部分IP的动作很明显是进行网站的漏洞扫描并伺机发起攻击，如以下日志：

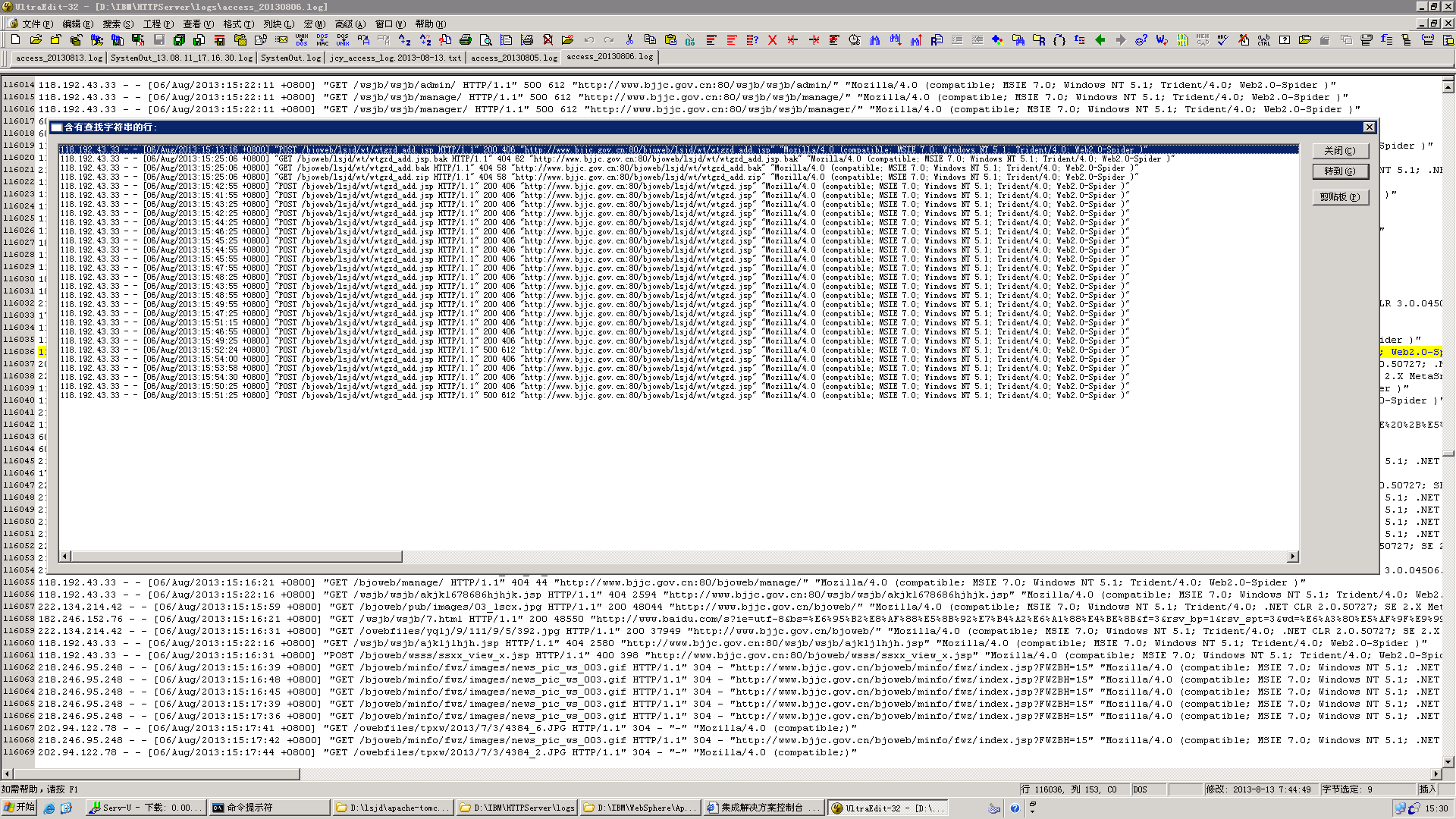
118.192.43.33 - - [06/Aug/2013:15:13:03 +0800] "GET **/jcgdy/view.jsp.bak**HTTP/1.1" 404 292 "http://www.xxx.gov.cn:80/jcgdy/view.jsp.bak" "Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0; **Web2.0-Spider** )"

来自于主机“118.192.43.33”的访问以GET的方式请求了“/jcgdy/view.jsp.bak”页面，很明显没有请求成功返回404错误，通过其User-Agent可以发现，这是一个Web2.0的爬虫，不过这不是一个正常的爬虫，结合该爬虫的其他访问动作来看，它请求的都是一些非正常的页面，如view.jsp.bak页面。假设我们在使用UE编辑工具编辑文件时，UE默认为生成一个文件名.bak的备份文件，如果开发人员不小心在线编辑这个文件并且留下了这个备份文件被蜘蛛扫描到，蜘蛛便可以下载这个文件从而看到里面的源码，如果里面写了一些比如连接数据库的信息，信息就能暴露。

如以下日志：

|  |
| --- |
| 118.192.43.33--[06/Aug/2013:15:13:16 +0800] "POST /bjoweb/lsjd/wt/wtgzd\_add.jsp HTTP/1.1" 200 406 "http://www.xxx.gov.cn:80/bjoweb/lsjd/wt/wtgzd\_add.jsp" "Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0; Web2.0-Spider )" |

在15:13分，Web2.0-Spider这个爬虫访问了上传文件表单的action（我们做超大文件上传测试的表单），这个action是一个jsp，上传文件的逻辑就由名为wtgzd\_add.jsp的jsp文件调用。综合后面该爬虫的访问请求，它多次以POST的方式请求该JSP，不排除它伪造HTTP协议的ContentLength长度进行虚假文件上传，而SmartUpload方法没有做大小限制，从而导致was内存溢出。观察该爬虫的其他几个动作，如（请放大查看）：



它要么在请求.bak文件或者.zip文件，要么频繁的POST上传文件的地址。时间都集中在15:10前后。正好和宕机时间段吻合，并且7月30号的访问日志中该IP的行为在访问时间，访问量，访问类型等都和8月6号的极为相似。

## 3.4 搜索引擎抓取

在分析时我们还发现了大量搜索引擎的爬虫，如百度、搜狗、一搜等。如百度的爬虫抓取痕迹：

123.125.71.59--[06/Aug/2013:09:30:53+0800]"GET /bjoweb/tpxw/view\_tpxw.jsp?XWBH=3960 HTTP/1.1" 200 6435 "-" "Mozilla/5.0 (compatible; **Baiduspider**/2.0; +http://www.baidu.com/search/spider.html)"

以下是各搜索引擎的访问量统计图：

可以考虑定义robots.txt文件限制其抓取甚至拒绝抓取。

## 3.5 存活检测

前文提到SDZC会发起对网站可用性的检测，定时访问网站的页面判断网站是否存活。

# 解决方案

结合前面的分析，提出了解决思路并部分进行了实施。

## 4.1 修改程序（已实施）

SmartUpload类的upload方法存在明显的内存溢出漏洞，考虑到网站上传文件的特点，进行了上传文件大小的限制，如果文件超过默认的500KB则返回错误提示。

由于SmartUpload还对非File类型的数据进行了封装，而且没有做任何大小限制，这会产生额外的内存占用，也会存在风险。

所以，建议后续上传文件的操作使用apache的commons包来实现上传文件功能。

## 4.2 子系统隔离部署（已实施）

在复现宕机故障的测试中，上传超大文件时虽然JVM生成了DUMP文件，也拒绝服务，但是最终没有OOM，所以不能100%确认是超大文件上传导致，因此，为了保障X网的顺利运行，保险起见，在修改问题代码之后将该子系统从X网中拆分出来单独部署，目的首先是确认宕机是否真正由该系统引起，其次确认上传超大文件是否是宕机的真正原因。如果系统仍然存在问题再次宕机，也不会影响X网的正常运行，减少问题影响面。

## 4.3 IP黑名单机制（已实施）

建立IP黑名单，将发起恶意扫描的主机加入黑名单并拒绝其连接请求以保障网站的安全。

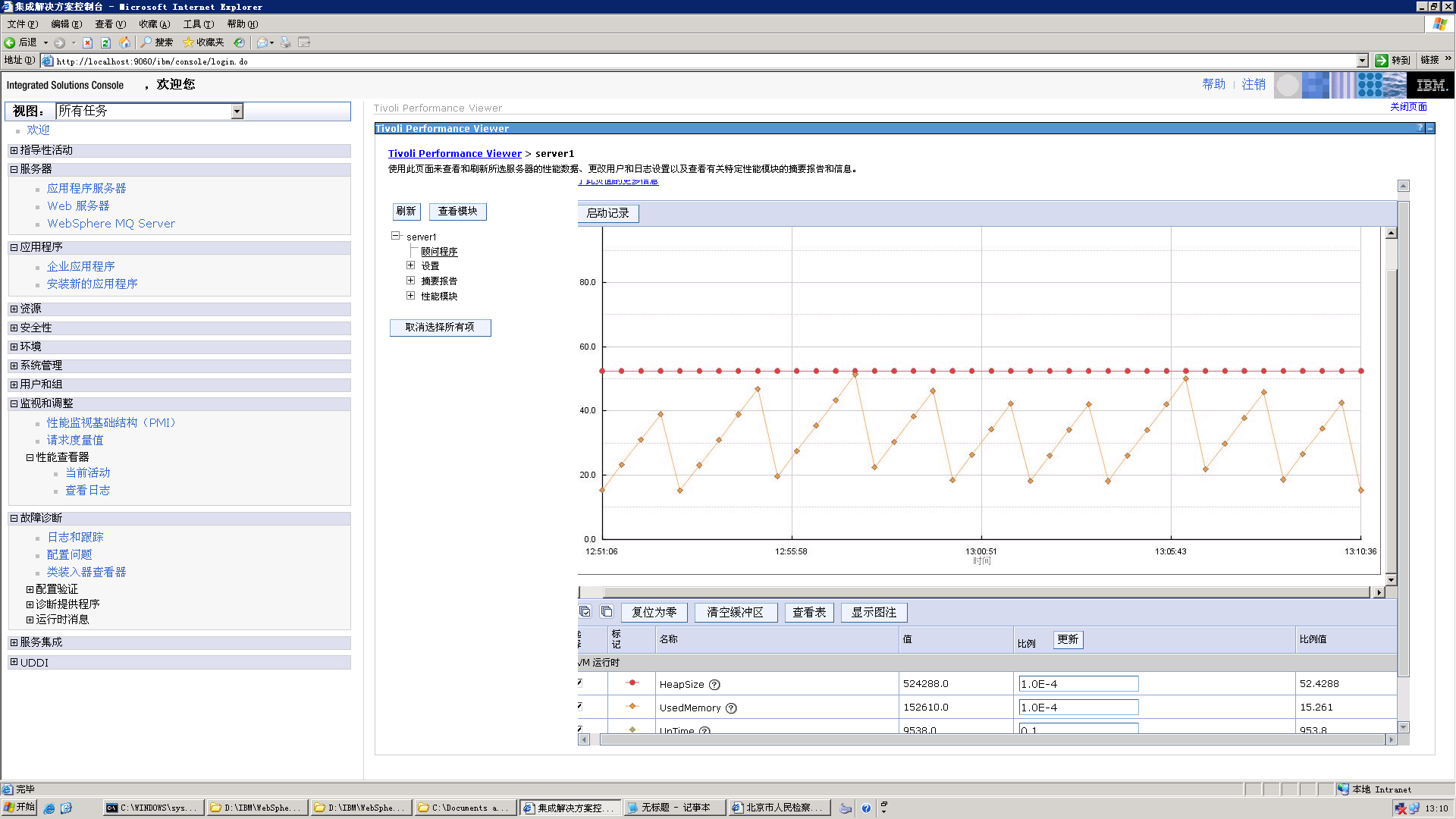
## 4.4 Robots文件定义（未实施）

搜索引擎的抓取目前访问量并不是很大，而且不同的搜索引擎的访问量不同，如百度爬虫8月6号全天的访问量在1600次左右，搜狗约在4600次左右。

未来根据监控情况考虑是否进行限制。目前暂不限制。

# 测试情况

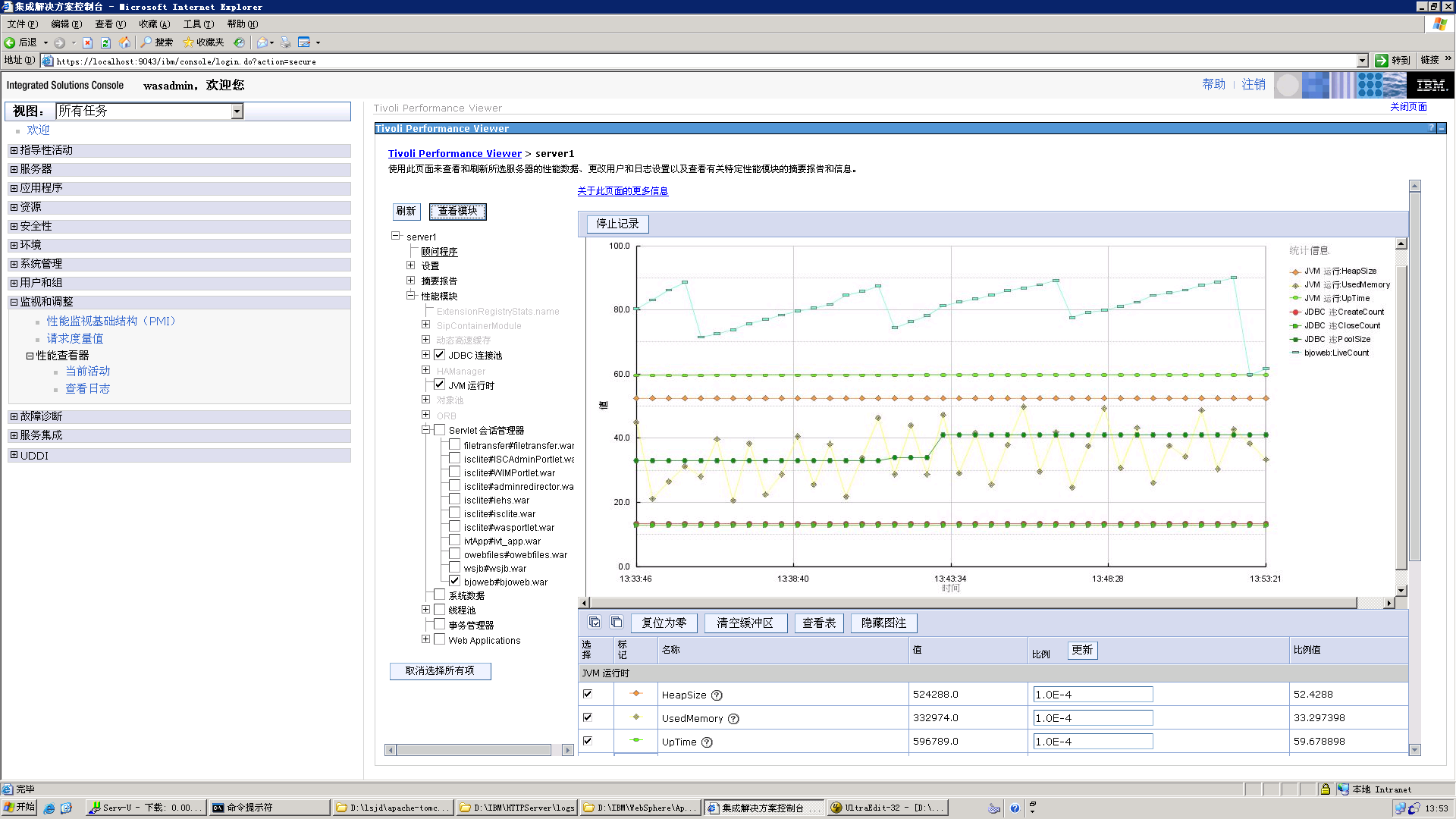
针对上传文件大小的修改，我们在测试环境上进行了测试，按照本文前面所进行的超大文件的上传测试步骤进行，测试结果显示文件超过500KB后直接上传失败，压力测试下也未见JVM堆内存出现大幅变化。



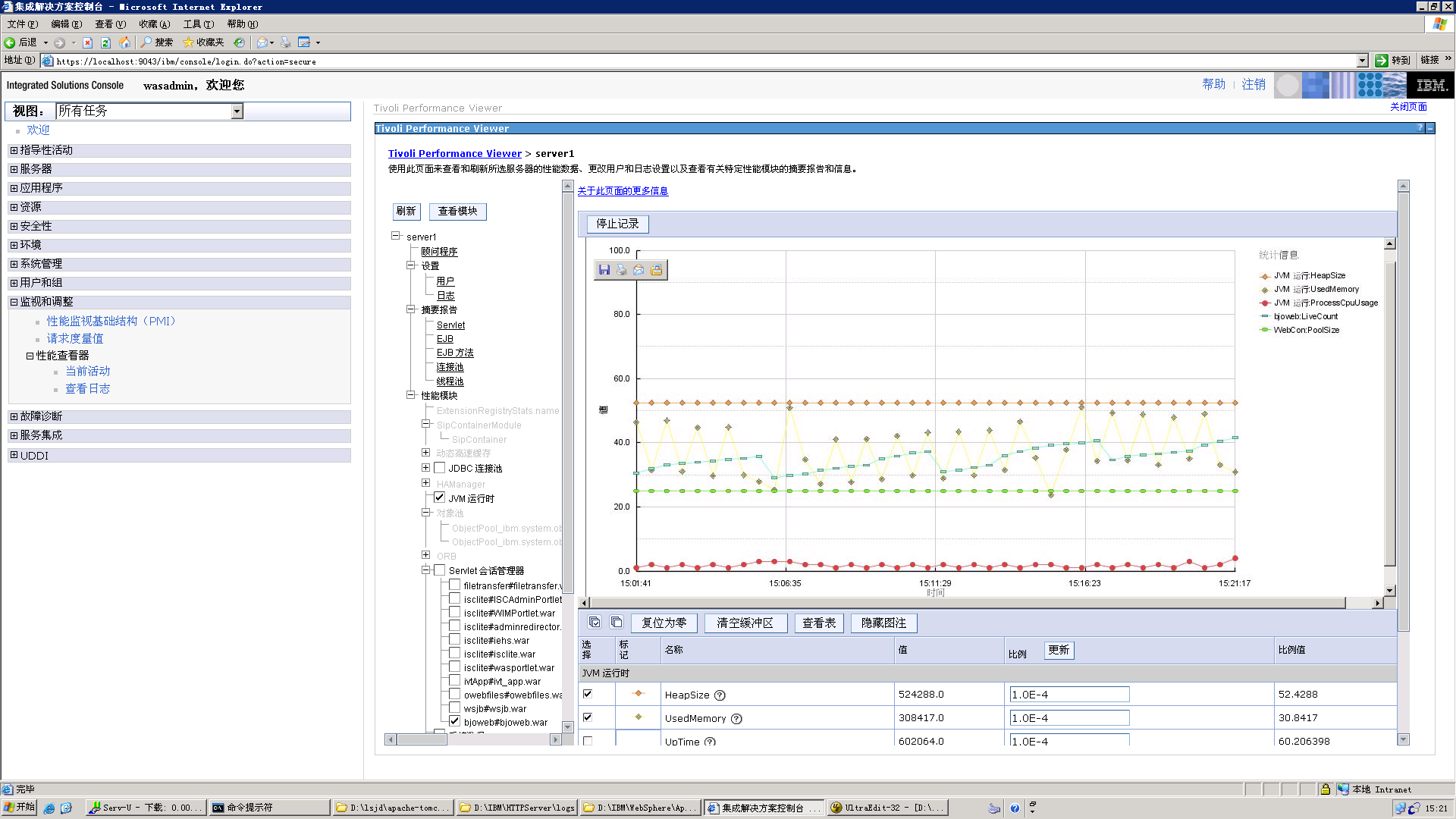
# 实际监控情况

在后续，我们立刻将测试后的解决方案实施到生产环境上，由于每次宕机时间都在每周二下午15:10分左右，因此周二下午我在市检开启了各种监控以查看中间件的各种指标情况。

周二下午13:33-13:53：

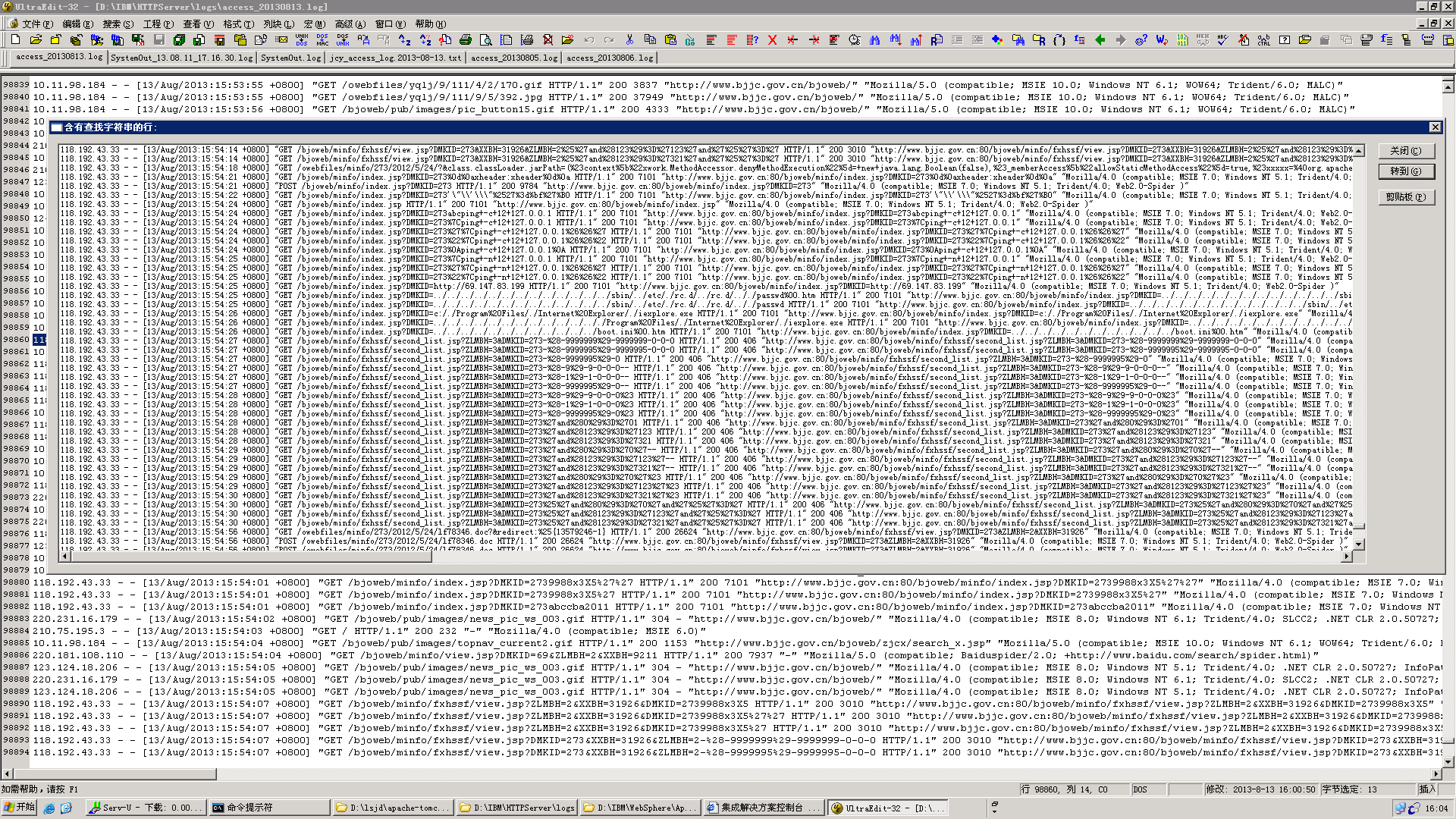


监控信息：JVM内存的使用和回收频率加快（黄色曲线），此时HTTP的会话数达到700左右（浅蓝色曲线）。

15:01-15:21：

总体运行平稳，JVM内存的使用和回收有条不紊，HTTP会话数为400左右，数据库连接数为25-30左右。但是在15:35分开始，网站的访问量突然激增，活动会话数从400飙升到1200多，查看后台访问日志，发现了IP118.192.43.33的访问量正在逐渐加大：

查看其部分访问路径（请放大查看）：

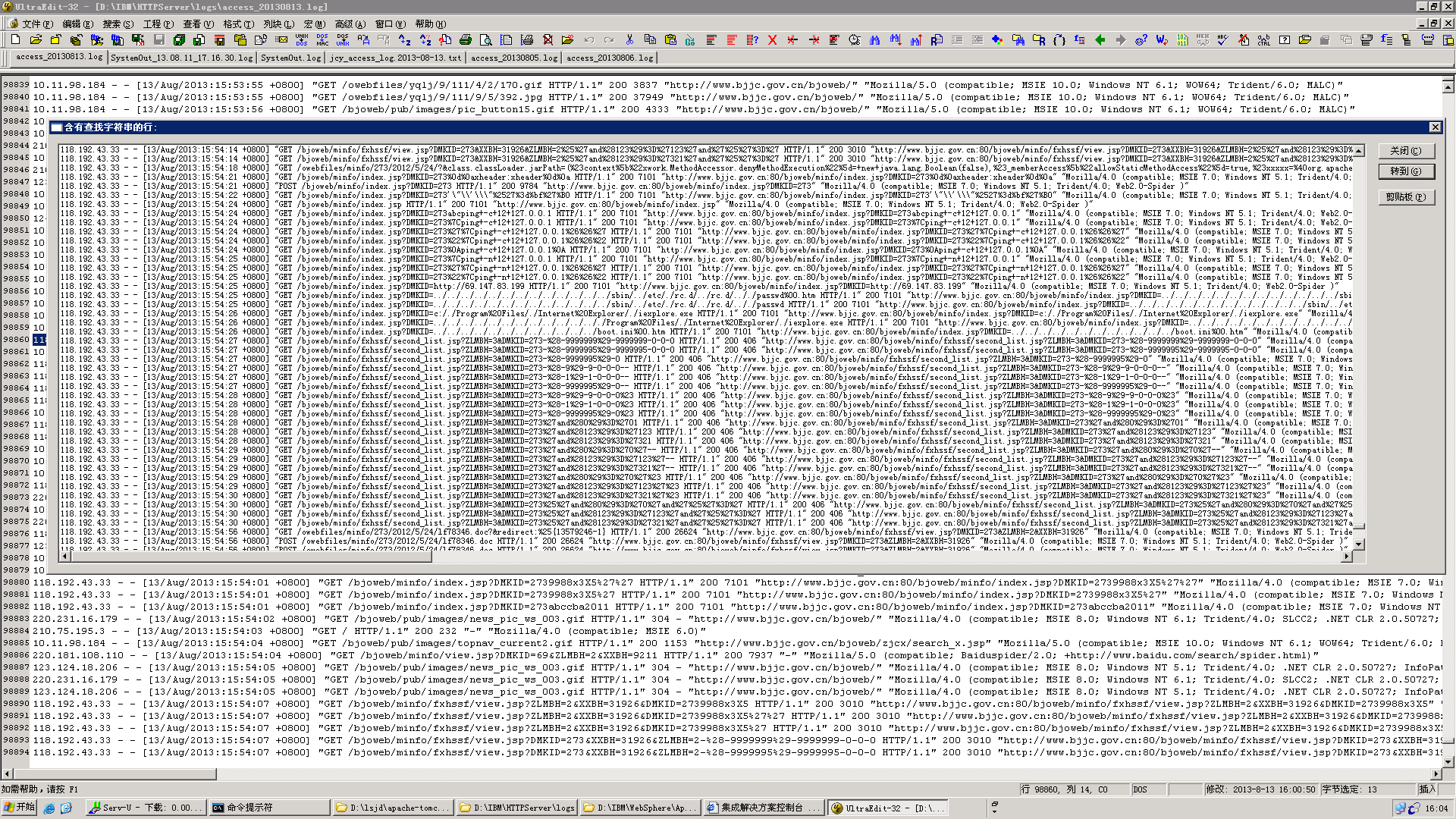
如以下访问日志：

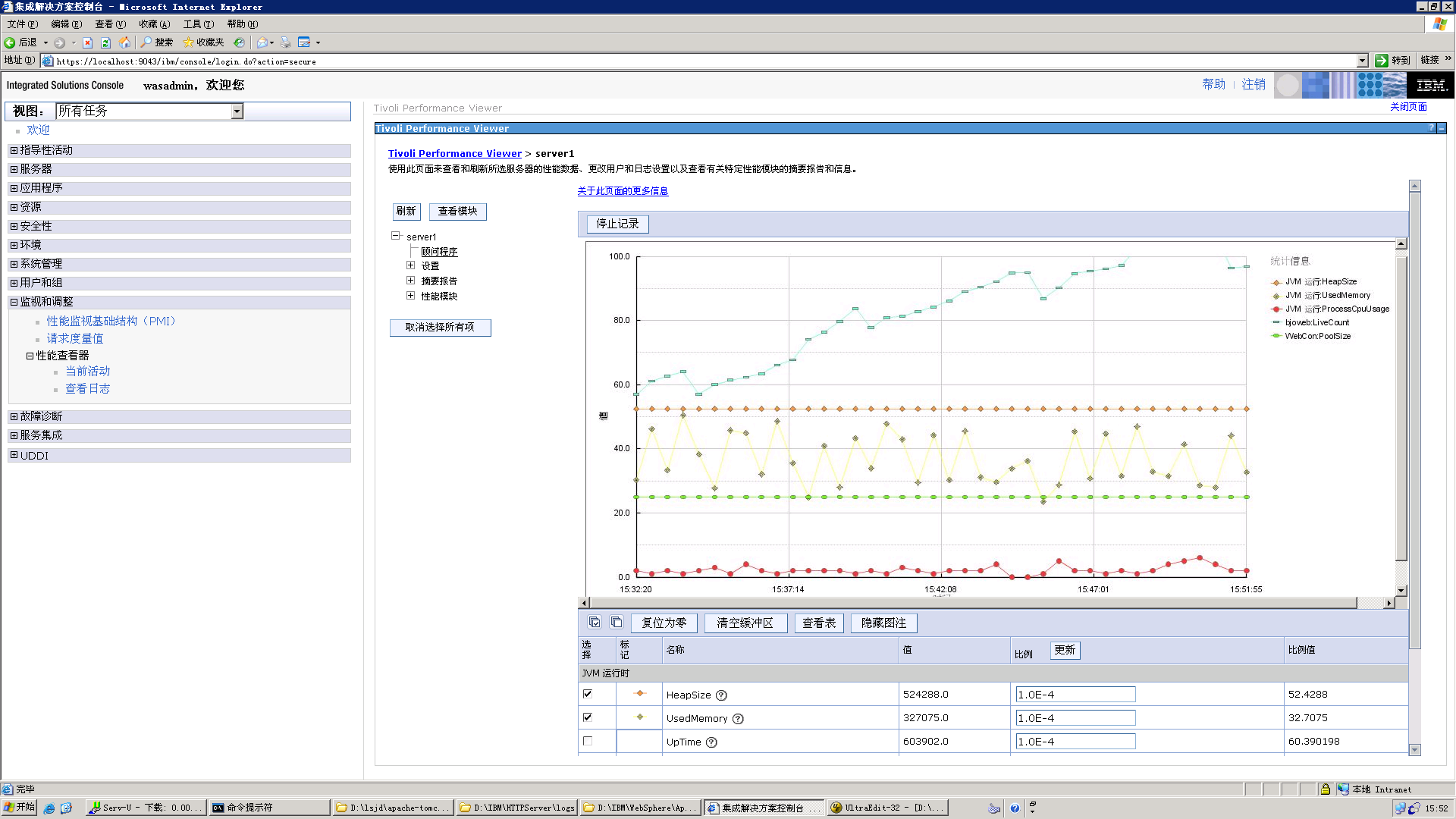
|  |
| --- |
| 118.192.43.33 - - [13/Aug/2013:15:54:30 +0800] "GET /bjoweb/minfo/fxhssf/second\_list.jsp?ZLMBH=3&DMKID=273%25%27and%28123%29%3D%27123%27and%27%25%27%3D%27HTTP/1.1" 200 406 "http://www.xxx.gov.cn:80/bjoweb/minfo/fxhssf/second\_list.jsp?ZLMBH=3&DMKID=273%25%27and%28123%29%3D%27123%27and%27%25%27%3D%27" "Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0; **Web2.0-Spider** )" |

该Web爬虫在本文前面曾涉及到，该爬虫访问的URL为“/bjoweb/minfo/fxhssf/second\_list.jsp?ZLMBH=3&DMKID=273%25%27and%28123%29%3D%27123%27and%27%25%27%3D%27”，这个页面解码之后为：

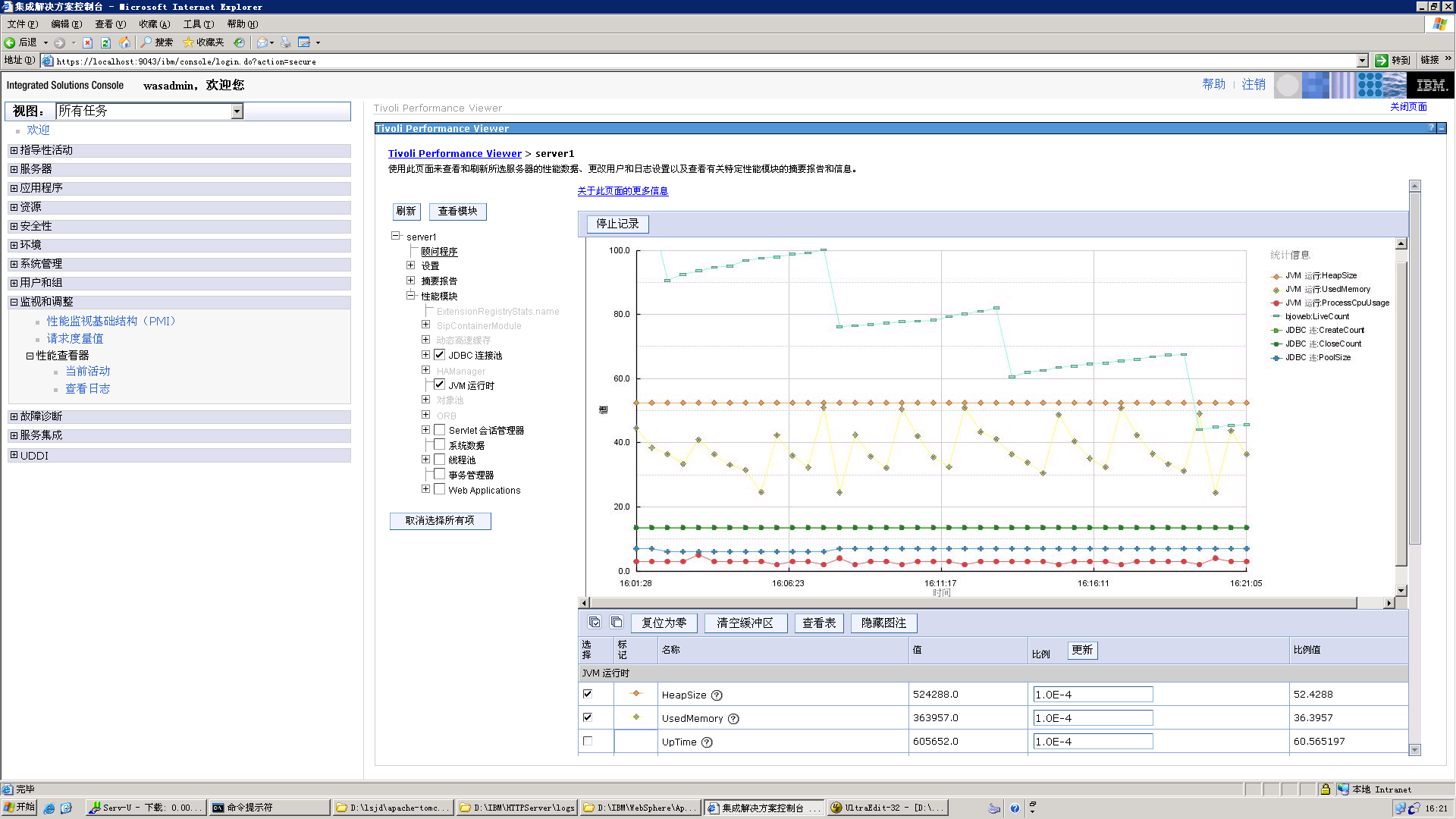
/bjoweb/minfo/fxhssf/second\_list.jsp?ZLMBH=3&DMKID=273%'and(123)='123'and'%'='

很明显，这个爬虫在进行sql注入漏洞扫描，它不断的在URL后面尝试拼接一些SQL关键词以寻找注入点，同时尝试请求一些不存在的文件以发现漏洞，对同一URL它都会请求多次：

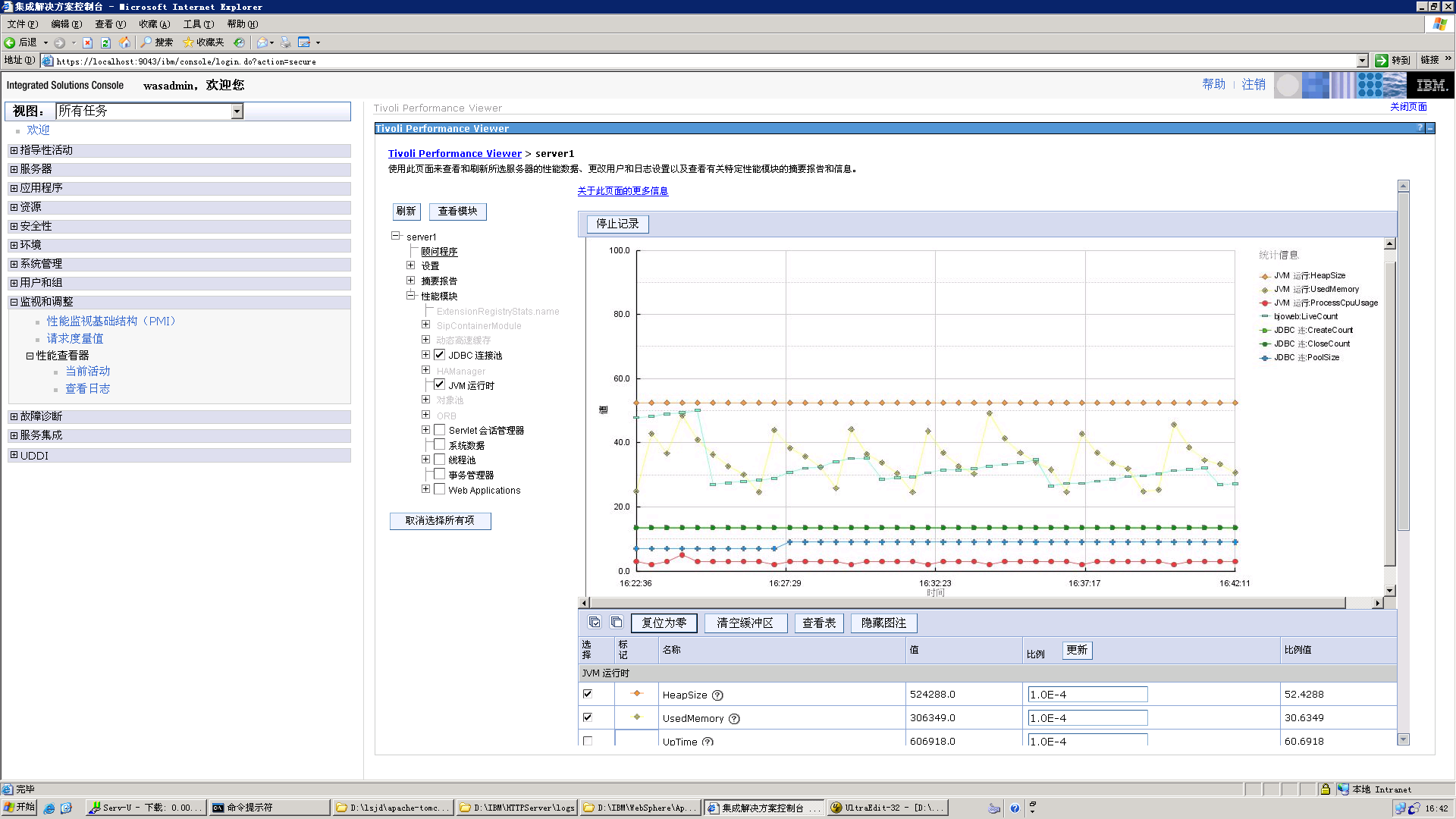




到了16:00左右，由于该IP疯狂的扫描，WAS的活动连接数超过了1400,不过JVM的表现一直很平稳，这个压力比我们8月8号上午的压力（4000）低很多，但是为了网站的安全，我们立刻将其加入防火墙DENY列表，瞬间访问量急剧下降，降低到400左右：



在16:30之后，可以看到，网站的各项指标都很正常，会话数也不高：



由于将Y的系统进行了隔离部署，因此该爬虫暂时未找到Y上传文件表单以及action，所以不能发起恶意的文件上传操作。不过可以肯定的是，即使该爬虫找到该页面，由于做了大小限制，也不会对系统造成影响。

应该可以确定，宕机是由Y子系统导致，Y子系统上传文件的确存在BUG。结合历史访问日志来看，X网的内存溢出过程如下：

通过这些访问动作基本和我们开始推测的内存溢出原因吻合：

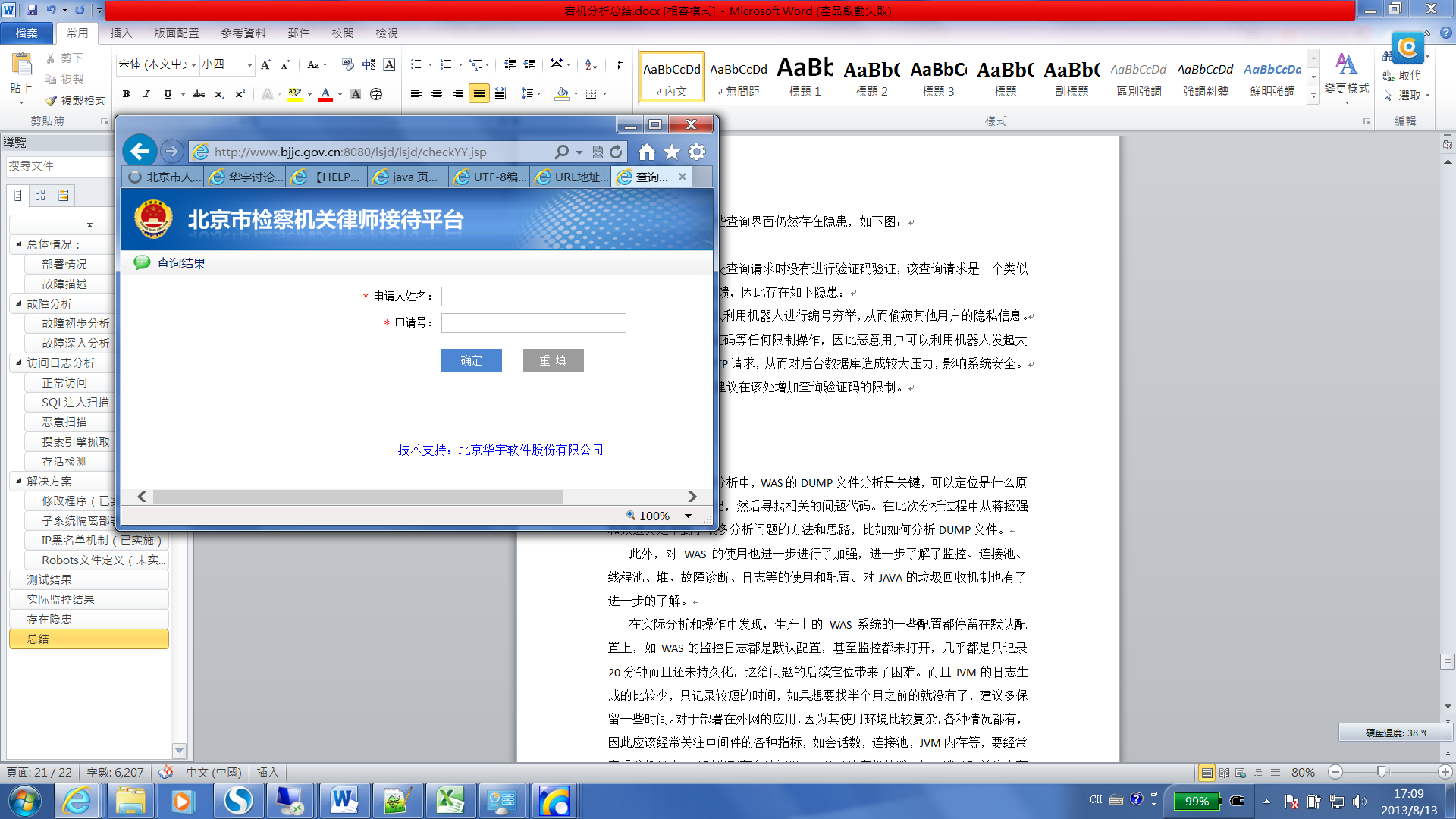
* 首先该爬虫或者真正的发起了一个超大文件的上传，或者伪造HTTP头部的ContentLength长度，进行一个虚假的超大文件上传请求。
* SmartUpload处理这个请求时，不加限制的根据ContentLength长度申请了一个同样大小的二进制数组。导致堆内存的使用到达极限。
* 爬虫连续发起该类请求，导致申请堆内存的线程出现等待，中间件开始拒绝服务。
* 而该爬虫仍继续大规模的对其他URL进行SQL注入漏洞扫描，源源不断的HTTP请求发来，从而在某个时刻的view.jsp请求诱发了OOM错误。

系统内存溢出的原因可以简单总结为：上传文件的java程序未对上传文件做大小限制也未对文件的接收进行缓存，因此被互联网上的某些主机进行了超大文件的恶意上传和高并发的SQL漏洞扫描操作操作导致内存溢出。

但是由于Y已经隔离部署，所以，不能100%确定该子系统再没有导致内存溢出的BUG。因此还要实际监控Y所在Tomcat的运行情况。

# 存在隐患

实际中发现，一些查询界面仍然存在隐患，如下图：



该查询界面在提交查询请求时没有进行验证码验证，该查询是一个类似于提交处理结果的反馈，因此存在如下隐患：

* 恶意用户可以利用机器人进行编号穷举，从而偷窥其他用户的隐私信息。
* 由于没有验证码等任何限制操作，因此恶意用户可以利用机器人发起大量连续的HTTP请求，从而对后台数据库造成较大压力，影响系统安全。

基于以上两点，建议在该处增加查询验证码的限制。

# 总结

在此次宕机故障分析中，WAS的DUMP文件分析是关键，可以定位是什么原因导致JVM内存溢出，然后寻找相关的问题代码。在此次分析过程中从蒋拯强和张进炎处学到了很多分析问题的方法和思路，比如如何分析DUMP文件。

此外，对WAS的使用也进一步进行了加强，进一步了解了监控、连接池、线程池、堆、故障诊断、日志等的使用和配置。对JAVA的垃圾回收机制也有了进一步的了解。

在实际分析和操作中发现，生产上的WAS系统的一些配置都停留在默认配置上，如WAS的监控日志都是默认配置，甚至监控都未打开，几乎都是只记录20分钟而且还未持久化，这给问题的后续定位带来了困难。而且JVM的日志生成的比较少，只记录较短的时间，如果想要找半个月之前的就没有了，建议多保留一些时间。对于部署在外网的应用，因为其使用环境比较复杂，各种情况都有，因此应该经常关注中间件的各种指标，如会话数，连接池，JVM内存等，要经常查看分析日志，及时发现存在的问题。

实际开发中对于老系统的新用法要做评估，如X网，其实SmartUpload的方法很久就有了，只是这个方法几乎都是在网站的后台使用，后台一般都是管理员，主观上不会恶意上传，但是如果放在外网且不用注册不用登录即可上传，就可能存在恶意上传的问题。因此代码要做仔细的研究、评估，对新的使用场景要做新的考虑。