

14个JVM内存配置参数:

1.栈内存大小相关设置

-Xss1024k

- 意义：设置线程栈占用内存大小。
- 默认值：不同的操作系统平台，其默认值不同，具体看官网说明。

2.堆内存大小相关设置

-Xms512m

- 意义：设置堆内存初始值大小。
- 默认值：如果未设置，初始值将是老年代和年轻代分配制内存之和。

-Xmx1024m

- 意义：设置堆内存最大值。
- 默认值：default value is chosen at runtime based on system configuration,具体请查看官网或者查看讨论[How is the default Java heap size determined?](#)。

3.年轻代内存大小相关设置

-Xmn512m

- 意义：设置新生代的初始值及最大值。
- 默认值：堆内存的1/4（这里要记住不是最大堆内存，还是已经分配的堆内存的1/4）。

-XX:NewSize=512m

- 意义：设置新生代的初始值。

-XX:MaxNewSize=512m

- 意义：设置新生代的最大值。

4.比率方式设置

-XX:NewRatio=8

- 意义：设置老年代和年轻代的比例。比如：-XX:NewRatio=8 表示老年代内存:年轻代内存=8:1 => 老年代占堆内存的8/9;年轻代占堆内存的1/9。
- 默认值：2。

-XX:SurvivorRatio=32

- 意义：设置新生代和存活区的比例（这里需要注意的是存活区指的是其中一个）。比如：-XX:SurvivorRatio=8 表示存活区：新生代=1：8 =》新生代占年轻代的8/10,每个存活区各占年轻代的1/10。
- 默认值：8。

-XX:MinHeapFreeRatio=40

- 意义：GC后，如果发现空闲堆内存占到整个预估上限值的40%，则增大上限值。
- 默认值：40。

-XX:MaxHeapFreeRatio=70

- 意义：GC后，如果发现空闲堆内存占到整个预估上限值的70%，则收缩预估上限值。
- 默认值：70。

5.Meta大小相关设置

-XX:MetaspaceSize=128m

- 意义：初始元空间大小，达到该值就会触发垃圾收集进行类型卸载，同时GC会对该值进行调整：如果释放了大量的空间，就适当降低该值；如果释放了很少的空间，那么在不超过MaxMetaspaceSize时，适当提高该值。
- 默认值：依赖平台。

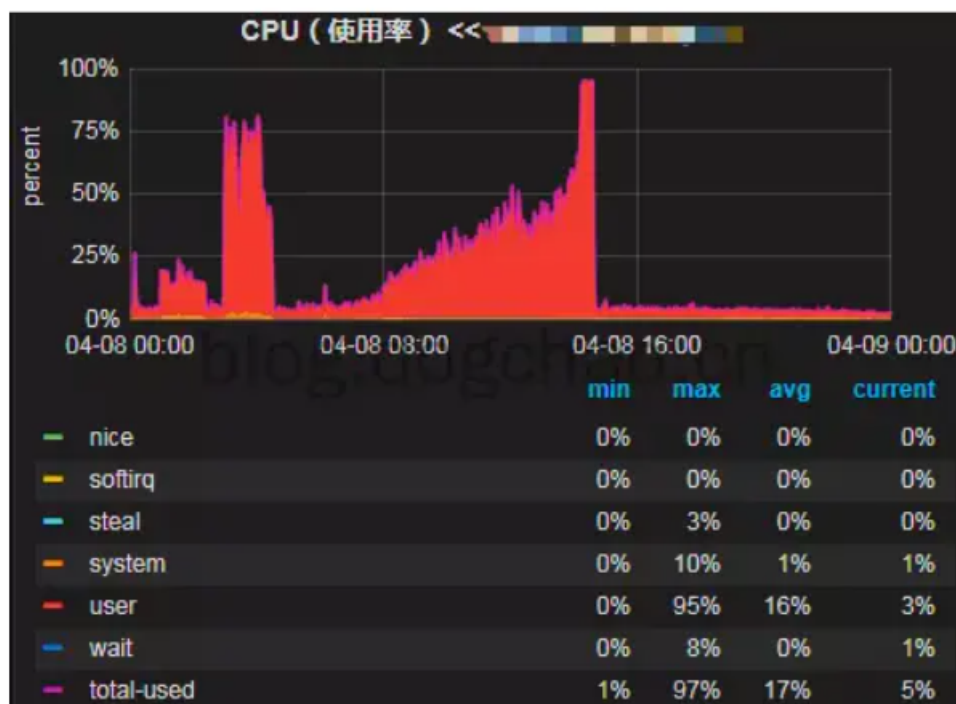
-XX:MaxMetaspaceSize=256m

- 意义：设置元空间的最大值，默认是没有上限的，也就是说你的系统内存上限是多少它就是多少。
- 默认值：默认没有上限，在技术上，Metaspace的尺寸可以增长到交换空间。

以上就是14个参数，为了深刻理解，建议本地配置让后观察内存大小变化（可以使用jmap -heap pid或者 visualGC来帮助观察）验证自己的理解是否正确。

项目中是否处理过JVM相关问题

下面是线上机器的cpu使用率，可以看到从4月8日开始，随着时间cpu使用率在逐步增高，最终使用率达到100%导致线上服务不可用，后面重启了机器后恢复：



1、排查思路

简单分析下可能出问题的地方，分为5个方向：

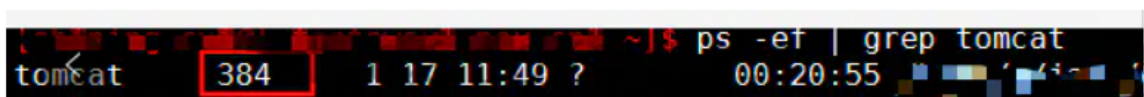
- 系统本身代码问题
- 内部下游系统的问题导致的雪崩效应
- 上游系统调用量突增
- http请求第三方的问题
- 机器本身的问题

2、开始排查

- 1.查看日志，没有发现集中的错误日志，初步排除代码逻辑处理错误。
- 2.首先联系了内部下游系统观察了他们的监控，发现一切正常。可以排除下游系统故障对我们的影响。
- 3.查看provider接口的调用量，对比7天没有突增，排除业务方调用量的问题。
- 4.查看tcp监控，TCP状态正常，可以排除是http请求第三方超时带来的问题。
- 5.查看机器监控，6台机器cpu都在上升，每个机器情况一样。排除机器故障问题。即通过上述方法没有直接定位到问题。

3、解决方案

- 1、重启了6台中问题比较严重的5台机器，先恢复业务。保留一台现场，用来分析问题。
- 2、查看当前的tomcat线程pid。



```
tomcat 384 1 17 11:49 ? 00:20:55
```

- 3、查看该pid下线程对应的系统占用情况。top -Hp 384

```
top - 17:16:25 up 887 days, 2:13, 2 users, load average: 2.17, 2.26, 2.08
Tasks: 388 total, 1 running, 387 sleeping, 0 stopped, 0 zombie
Cpu(s): 39.3%us, 0.5%sy, 0.0%ni, 60.2%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 8059648k total, 7916540k used, 143108k free, 89472k buffers
Swap: 4194296k total, 0k used, 4194296k free, 4582308k cached

  PID USER      PR  NI  VIRT  RES  SHR  S  %CPU  %MEM    TIME+  COMMAND
 4433 tomcat    20   0 5001m 2.4g 6648 S 38.2 31.2 133:47.97 java
 4430 tomcat    20   0 5001m 2.4g 6648 S 37.5 31.2 133:51.40 java
 4431 tomcat    20   0 5001m 2.4g 6648 S 37.5 31.2 133:41.98 java
 4432 tomcat    20   0 5001m 2.4g 6648 S 37.2 31.2 133:47.82 java
 5108 tomcat    20   0 5001m 2.4g 6648 S  2.7 31.2   4:49.57 java
 4434 tomcat    20   0 5001m 2.4g 6648 R  2.0 31.2 11:41.40 java
 4582 tomcat    20   0 5001m 2.4g 6648 S  1.0 31.2 82:40.83 java
 4455 tomcat    20   0 5001m 2.4g 6648 S  0.3 31.2  1:34.72 java
 4583 tomcat    20   0 5001m 2.4g 6648 S  0.3 31.2  4:44.83 java
 4850 tomcat    20   0 5001m 2.4g 6648 S  0.3 31.2 18:11.02 java
 5165 tomcat    20   0 5001m 2.4g 6648 S  0.3 31.2  1:51.78 java
 4427 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  0:00.00 java
 4429 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  0:21.24 java
 4435 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  0:13.00 java
 4436 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  0:13.60 java
 4437 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  0:00.00 java
 4438 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  0:43.22 java
 4439 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  0:40.96 java
 4440 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  0:00.00 java
 4441 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  3:06.05 java
 4442 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  0:00.00 java
 4443 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  0:08.78 java
 4450 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  0:08.77 java
 4451 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  0:26.11 java
 4452 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  0:19.39 java
 4453 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  0:19.24 java
 4454 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  1:33.12 java
 4456 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  0:23.19 java
 4457 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  0:00.08 java
 4458 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  0:00.04 java
 4459 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  0:00.07 java
 4500 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  0:00.01 java
 4581 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  0:00.24 java
 4584 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  0:08.74 java
 4681 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  0:08.79 java
 4682 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  0:17.00 java
 4683 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  0:01.51 java
 4684 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  0:09.27 java
 4685 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  0:00.12 java
 4686 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  0:06.66 java
 4687 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  1:52.11 java
 4688 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  1:37.86 java
 4689 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  1:54.34 java
 4690 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  1:26.35 java
 4691 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  1:20.71 java
 4692 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  1:46.02 java
 4693 tomcat    20   0 5001m 2.4g 6648 S  0.0 31.2  1:46.68 java
```

- 4、发现pid 4430 4431 4432 4433 线程分别占用了约40%的cpu
- 5、将这几个pid转为16进制，分别为114e 114f 1150 1151
- 6、下载当前的java线程栈 `sudo -u tomcat jstack -l 384>/1.txt`
- 7、查询5中对应的线程情况，发现都是gc线程导致的

```
"GC task thread#0 (ParallelGC)" prio=10 tid=0x00007f7ce001f000 nid=0x114e runnable
"GC task thread#1 (ParallelGC)" prio=10 tid=0x00007f7ce0021000 nid=0x114f runnable
"GC task thread#2 (ParallelGC)" prio=10 tid=0x00007f7ce0022800 nid=0x1150 runnable
"GC task thread#3 (ParallelGC)" prio=10 tid=0x00007f7ce0024800 nid=0x1151 runnable
```

- 8、dump java堆数据

`sudo -u tomcat jmap -dump:live,format=b,file=/dump201612271310.dat 384`

- 9、使用MAT加载堆文件，可以看到javax.crypto.JceSecurity对象占用了95%的内存空间，初步定位到问题。

MAT查看：

▼ Shortest Paths To the Accumulation Point

Class Name	Shallow Heap	Retained Heap
java.lang.Object[32768] @ 0x7c8312848	131,088	1,937,502,816
table java.util.IdentityHashMap @ 0x77ffecdb8	40	1,937,502,856
verificationResults class javax.crypto.JceSecurity @ 0x77ffef7b8 System Class	40	1,937,504,384

▼ Accumulated Objects in Dominator Tree

Class Name	Shallow Heap	Retained Heap	Percentage
class javax.crypto.JceSecurity @ 0x77ffef7b8	40	1,937,504,384	95.36%
java.util.IdentityHashMap @ 0x77ffecdb8	40	1,937,502,856	95.36%
java.lang.Object[32768] @ 0x7c8312848	131,088	1,937,502,816	95.36%
org.bouncycastle.jce.provider.BouncyCastleProvider @ 0x7c79fd58	96	207,168	0.01%
org.bouncycastle.jce.provider.BouncyCastleProvider @ 0x7c79fd048	96	207,168	0.01%
org.bouncycastle.jce.provider.BouncyCastleProvider @ 0x7c7a2f988	96	207,168	0.01%
org.bouncycastle.jce.provider.BouncyCastleProvider @ 0x7c7a858a0	96	207,168	0.01%
org.bouncycastle.jce.provider.BouncyCastleProvider @ 0x7c7ac6258	96	207,168	0.01%
org.bouncycastle.jce.provider.BouncyCastleProvider @ 0x7c7af8508	96	207,168	0.01%

10、查看类的引用树，看到BouncyCastleProvider对象持有过多。即我们代码中对该对象的处理方式是错误的，定位到问题。

4、代码分析

我们代码中有一块是这样写的

```
private static String decrypt(String encryptedStr, PrivateKey privateKey) throws Exception {  
    Cipher cipher = Cipher.getInstance("RSA", new BouncyCastleProvider());  
    cipher.init(Cipher.DECRYPT_MODE, privateKey);  
    byte[] encryptedData = Base64Utils.decodeFromString(encryptedStr);  
    int inputLen = encryptedData.length;  
    ByteArrayOutputStream out = new ByteArrayOutputStream();  
    int offSet = 0;  
    byte[] cache;  
    int i = 0;
```

这是加解密的功能，每次运行加解密都会new一个BouncyCastleProvider对象，放到Cipher.getInstance()方法中。

看下Cipher.getInstance()的实现，这是jdk的底层代码实现，追踪到JceSecurity类中


```

static synchronized Exception getVerificationResult(Provider var0) {
    Object var1 = verificationResults.get(var0);
    if (var1 == PROVIDER_VERIFIED) {
        return null;
    } else if (var1 != null) {
        return (Exception)var1;
    } else if (verifyingProviders.get(var0) != null) {
        return new NoSuchProviderException("Recursion during verification");
    } else {
        Exception var3;
        try {
            verifyingProviders.put(var0, Boolean.FALSE);
            URL var2 = getCodeBase(var0.getClass());
            verifyProviderJar(var2);
            verificationResults.put(var0, PROVIDER_VERIFIED);
            var3 = null;
            return var3;
        } catch (Exception var7) {
            verificationResults.put(var0, var7);
            var3 = var7;
        } finally {
            verifyingProviders.remove(var0);
        }

        return var3;
    }
}

```

verifyingProviders每次put后都会remove, verificationResults只会put, 不会remove

```

final class JoeSecurity {
    static final SecureRandom RANDOM = new SecureRandom();
    private static CryptoPermissions defaultPolicy = null;
    private static CryptoPermissions exemptPolicy = null;
    private static final Map verificationResults = new IdentityHashMap();
    private static final Map verifyingProviders = new IdentityHashMap();
    private static boolean isRestricted = true;
    private static final Object PROVIDER_VERIFIED;
    private static final URL NULL_URL;
    private static final Map codeBaseCacheRef;

    private JoeSecurity() {

```

看到verificationResults是一个static的map, 即属于JoeSecurity类的。所以每次运行到加解密都会向这个map put一个对象, 而这个map属于类的维度, 所以不会被GC回收。这就导致了大量的new的对象不被回收。

5、代码改进

将有问题的对象置为static, 每个类持有有一个, 不会多次新建。

总结:

遇到线上问题不要慌, 首先确认排查问题的思路:

- 查看日志
- 查看CPU情况
- 查看TCP情况

- 查看java线程, jstack
- 查看java堆, jmap
- 通过MAT分析堆文件, 寻找无法被回收的对象

jstack和jmap的区别

jstack主要用来查看某个Java进程内的线程堆栈信息

jmap用来查看堆内存使用状况, 一般结合jhat使用。

https://blog.csdn.net/sinat_29581293/article/details/70214436