记一次在生产环境排查 OutOfMemoryError (OOM)的真实经历

我们都知道JVM的内存管理是自动化的，Java语言的程序指针也不需要开发人员手工释放，JVM的GC会自动的进行回收，但是，如果编程不当，JVM仍然会发生内存泄露，导致Java程序产生了 OutOfMemoryError（ OOM ）错误。

产生OutOfMemoryError错误的原因包括：

> 1. java.lang.OutOfMemoryError: Java heap space
> 2. java.lang.OutOfMemoryError: PermGen space及其解决方法
> 3. java.lang.OutOfMemoryError: unable to create new native thread
> 4. java.lang.OutOfMemoryError ： GC overhead limit exceeded

对于第1种异常，表示Java堆空间不够，当应用程序申请更多的内存，而Java堆内存已经无法满足应用程序对内存的需要，将抛出这种异常。

GitChat

对于第2种异常，表示Java永久带（方法区）空间不够，永久带用于存放类的字节码和长常量池，类的字节码加载后存放在这个区域，这和存放对象实例的堆区是不同的，大多数JVM的实现都不会对永久带进行垃圾回收，因此，只要类加载的过多就会出现这个问题。一般的应用程序都不会产生这个错误，然而，对于Web服务器来讲，会产生有大量的JSP，JSP在运行时被动态的编译成Java Servlet类，然后加载到方法区，因此，太多的JSP的Web工程可能产生这个异常。

对于第3种异常，本质原因是创建了太多的线程，而能创建的线程数是有限制的，导致了这种异常的发生。

对于第4种异常，是在并行或者并发回收器在GC回收时间过长、超过98%的时间用来做GC并且回收了不到2%的堆内存，然后抛出这种异常进行提前预警，用来避免内存过小造成应用不能正常工作。

下面两个异常与OOM有关系，但是，又没有绝对关系。

> 1. java.lang.StackOverflowError …
> 2. java.net.SocketException: Too many open files

对于第1种异常，是JVM的线程由于递归或者方法调用层次太多，占满了线程堆栈而导致的，线程堆栈默认大小为1M。

对于第2种异常，是由于系统对文件句柄的使用是有限制的，而某个应用程序使用的文件句柄超过了这个限制，就会导致这个问题。

上面介绍了OOM相关的基础知识，接下来我们开始讲述笔者经历的一次OOM问题的定位和解决的过程。

1. 产生问题的现象

在某一段时间内，我们发现不同的业务服务开始偶发的报OOM的异常，有的时候是白天发生，有的时候是晚上发生，有的时候是基础服务A发生，有的时候是上层服务B发生，有的时候是上层服务C发生，有的时候是下层服务D发生，丝毫看不到一点规律。

产生问题的异常如下：

```
Caused by: java.lang.OutOfMemoryError: unable to create new
native thread at java.lang.Thread.start0(Native Method)
at java.lang.Thread.start(Thread.java:597)
at java.util.Timer.<init>(Timer.java:154)
```

2. 解决问题的思路和过程

经过细心观察发现，产生问题虽然在不同的时间发生在不同的服务池，但是，晚上0点发生的时候概率较大，也有其他时间偶发，但是都在整点。

GitChat

这个规律很重要，虽然不是一个时间，但是基本都在整点左右发生，并且晚上0点居多。从这个角度思考，整点或者0点系统是否有定时，与出问题的每个业务系统技术负责人核实，0点没有定时任务，其他时间的整点有定时任务，但是与发生问题的时间不吻合，这个思路行不通。

到现在为止，从现象的规律上我们已经没法继续分析下去了，那我们回顾一下错误本身：

> java.lang.OutOfMemoryError: unable to create new native thread

顾名思义，错误产生的原因就是应用不能创建线程了，但是，应用还需要创建线程。为什么程序不能创建线程呢？

有两个具体原因造成这个异常:

> 1. 由于线程使用的资源过多，操作系统已经不能再提供给应用资源了。
> 2. 操作系统设置了应用创建线程的最大数量，并且已经达到了最大允许数量。

上面第1条资源指的是内存，而第2条中，在Linux下线程使用轻量级进程实现的，因此线程的最大数量也是操作系统允许的进程的最大数量。

内存计算

操作系统中的最大可用内存除去操作系统本身使用的部分，剩下的都可以为某一个进程服务，在JVM进程中，内存又被分为堆、本地内存和栈等三大块，Java堆是JVM自动管理的内存，应用的对象的创建和销毁、类的装载等都发生在这里，本地内存是Java应用使用的一种特殊内存，JVM并不直接管理其生命周期，每个线程也会有一个栈，是用来存储线程工作过程中产生的方法局部变量、方法参数和返回值的，每个线程对应的栈的默认大小为1M。

Linux和JVM的内存管理示意图如下：

因此，从内存角度来看创建线程需要内存空间，如果JVM进程正当一个应用创建线程，而操作系统没有剩余的内存分配给此JVM进程，则会抛出问题中的OOM异常：unable to create new native thread。

如下公式可以用来从内存角度计算允许创建的最大线程数：

> 最大线程数 =（操作系统最大可用内存 - JVM内存 - 操作系统预留内存）/ 线程栈大小

GitChat

根据这个公式，我们可以通过剩余内存计算可以创建线程的数量。

下面是问题出现的时候，从生产机器上执行前面小节介绍的Linux命令free的输出：

```
free -m >> /tmp/free.log
             total       used       free     shared    buffers
cached
Mem:          7872       7163        709          0         31
3807
-/+ buffers/cache:       3324       4547
Swap:         4095        173       3922
Tue Jul 5 00:27:51 CST 2016
```

从上面输出可以得出，生产机器8G内存，使用了7G，剩余700M可用，其中操作系统cache使用3.8G。操作系统cache使用的3.8G是用来缓存IO数据的，如果进程内存不够用，这些内存是可以释放出来优先分配给进程使用。然而，我们暂时并不需要考虑这块内存，剩余的700M空间完全可以继续用来创建线程数：

> 700M / 1M = 700个线程

因此，根据内存可用计算，当OOM异常：unable to create new native thread问题发生的时候，还有700M可用内存，可以创建700个线程。

到现在为止可以证明此次OOM异常不是因为线程吃光所有的内存而导致的。

线程数对比

上面提到，有两个具体原因造成这个异常，我们上面已经排除了第1个原因，那我们现在从第2个原因入手，评估是否操作系统设置了应用创建线程的最大数量，并且已经达到了最大允许数量。

在问题出现的生产机器上使用ulimit -a来显示当前的各种系统对用户使用资源的限制：

```
robert@robert-ubuntu1410: ~$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 62819
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 65535
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 10240
cpu time               (seconds, -t) unlimited
max user processes              (-u) 1024
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

GitChat

这里面我们看到生产机器设置的允许使用的最大用户进程数为1024：

```
max user processes              (-u) 1024
```

现在，我们必须获得问题出现的时候，用户下创建的线程情况。

在问题产生的时候，我们使用前面小结介绍的JVM监控命令jstack命令打印出了Java线程情况,jstack命令的示例输出如下：

```
robert@robert-ubuntu1410:~$ jstack 2743
2017-04-09 12:06:51
Full thread dump Java HotSpot(TM) Server VM (25.20-b23 mixed
mode):

"Attach Listener" #23 daemon prio=9 os_prio=0 tid=0xc09adc00
nid=0xb4c waiting on condition [0x00000000]
   java.lang.Thread.State: RUNNABLE

"http-nio-8080-Acceptor-0" #22 daemon prio=5 os_prio=0
```

```
tid=0xc3341000 nid=0xb02 runnable [0xbf1bd000]
   java.lang.Thread.State: RUNNABLE
    at sun.nio.ch.ServerSocketChannelImpl.accept0(Native Method)
    at
sun.nio.ch.ServerSocketChannelImpl.accept(ServerSocketChannelImpl
.java:241)
    - locked <0xcf8938d8> (a java.lang.Object)
    at
org.apache.tomcat.util.net.NioEndpoint$Acceptor.run(NioEndpoint.j
ava:688)
    at java.lang.Thread.run(Thread.java:745)

"http-nio-8080-ClientPoller-1" #21 daemon prio=5 os_prio=0
tid=0xc35bc400 nid=0xb01 runnable [0xbf1fe000]
   java.lang.Thread.State: RUNNABLE
    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:79)
    at
sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:86)
    - locked <0xcf99b100> (a sun.nio.ch.Util$2)
    - locked <0xcf99b0f0> (a
java.util.Collections$UnmodifiableSet)
    - locked <0xcf99aff8> (a sun.nio.ch.EPollSelectorImpl)
    at sun.nio.ch.SelectorImpl.select(SelectorImpl.java:97)
    at
org.apache.tomcat.util.net.NioEndpoint$Poller.run(NioEndpoint.jav
a:1052)
    at java.lang.Thread.run(Thread.java:745)
......
```

GitChat

从jstack命令的输出并统计后，我们得知，JVM一共创建了904个线程，但是，这还没有到最大的进程限制1024。

```
robert@robert-ubuntu1410:~$ grep "Thread " js.log | wc -l
     904
```

这是我们思考，除了JVM创建的应用层线程，JVM本身可能会有一些管理线程存在，而且操作系统内用户下可能也会有守护线程在运行。

我们继续从操作系统的角度来统计线程数，我们使用上面小结介绍的Linux操作系统命令pstack，并得到如下的输出：

```
PID   LWP USER     %CPU %MEM CMD
    1     1 root      0.0  0.0 /sbin/init
    2     2 root      0.0  0.0 [kthreadd]
```

```
    3     3 root      0.0  0.0 [migration/0]
    4     4 root      0.0  0.0 [ksoftirqd/0]
    5     5 root      0.0  0.0 [migration/0]
    6     6 root      0.0  0.0 [watchdog/0]
    7     7 root      0.0  0.0 [migration/1]
    8     8 root      0.0  0.0 [migration/1]
    9     9 root      0.0  0.0 [ksoftirqd/1]
   10    10 root      0.0  0.0 [watchdog/1]
   11    11 root      0.0  0.0 [migration/2]
   12    12 root      0.0  0.0 [migration/2]
   13    13 root      0.0  0.0 [ksoftirqd/2]
   14    14 root      0.0  0.0 [watchdog/2]
   15    15 root      0.0  0.0 [migration/3]
   16    16 root      0.0  0.0 [migration/3]
   17    17 root      0.0  0.0 [ksoftirqd/3]
   18    18 root      0.0  0.0 [watchdog/3]
   19    19 root      0.0  0.0 [events/0]
   20    20 root      0.0  0.0 [events/1]
   21    21 root      0.0  0.0 [events/2]
   22    22 root      0.0  0.0 [events/3]
   23    23 root      0.0  0.0 [cgroup]
   24    24 root      0.0  0.0 [khelper]

 ......

 7257  7257 zabbix    0.0  0.0
/usr/local/zabbix/sbin/zabbix_agentd: active checks #2 [idle 1
sec]
 7258  7258 zabbix    0.0  0.0
/usr/local/zabbix/sbin/zabbix_agentd: active checks #3 [idle 1
sec]
 7259  7259 zabbix    0.0  0.0
/usr/local/zabbix/sbin/zabbix_agentd: active checks #4 [idle 1
sec]

 ......

 9040  9040 app       0.0 30.5 /apps/prod/jdk1.6.0_24/bin/java -
Dnop -
Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager
-Ddbconfigpath=/apps/dbconfig/ -Djava.io.tmpdir=/apps/data/java-
tmpdir -server -Xms2048m -Xmx2048m -XX:PermSize=128m -
XX:MaxPermSize=512m -Dcom.sun.management.jmxremote -
Djava.rmi.server.hostname=192.168.10.194 -
Dcom.sun.management.jmxremote.port=6969 -
Dcom.sun.management.jmxremote.ssl=false -
Dcom.sun.management.jmxremote.authenticate=false -
XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/tmp -Xshare:off
-Dhostname=sjsa-trade04 -Djute.maxbuffer=41943040 -
Djava.net.preferIPv4Stack=true -Dfile.encoding=UTF-8 -
Dworkdir=/apps/data/tomcat-work -
Djava.endorsed.dirs=/apps/product/tomcat-trade/endorsed -
```

GitChat

```
classpath commonlib:/apps/product/tomcat-
trade/bin/bootstrap.jar:/apps/product/tomcat-trade/bin/tomcat-
juli.jar -Dcatalina.base=/apps/product/tomcat-trade -
Dcatalina.home=/apps/product/tomcat-trade -
Djava.io.tmpdir=/apps/data/tomcat-temp/
org.apache.catalina.startup.Bootstrap start
 9040  9041 app       0.0 30.5 /apps/prod/jdk1.6.0_24/bin/java -
Dnop -
Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager
-Ddbconfigpath=/apps/dbconfig/ -Djava.io.tmpdir=/apps/data/java-
tmpdir -server -Xms2048m -Xmx2048m -XX:PermSize=128m -
XX:MaxPermSize=512m -Dcom.sun.management.jmxremote -
Djava.rmi.server.hostname=192.168.10.194 -
Dcom.sun.management.jmxremote.port=6969 -
Dcom.sun.management.jmxremote.ssl=false -
Dcom.sun.management.jmxremote.authenticate=false -
XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/tmp -Xshare:off
-Dhostname=sjsa-trade04 -Djute.maxbuffer=41943040 -
Djava.net.preferIPv4Stack=true -Dfile.encoding=UTF-8 -
Dworkdir=/apps/data/tomcat-work -
Djava.endorsed.dirs=/apps/product/tomcat-trade/endorsed -
classpath commonlib:/apps/product/tomcat-
trade/bin/bootstrap.jar:/apps/product/tomcat-trade/bin/tomcat-
juli.jar -Dcatalina.base=/apps/product/tomcat-trade -
Dcatalina.home=/apps/product/tomcat-trade -
Djava.io.tmpdir=/apps/data/tomcat-temp/
org.apache.catalina.startup.Bootstrap start

......
```

GitChat

通过命令统计用户下已经创建的线程数为1201。

```
$ grep app pthreads.log | wc -l
    1201
```

现在我们确定，1201的数字已经相当的接近1204的最大进程数了，正如前面我们提到，在Linux操作系统里，线程是通过轻量级的进程实现的，因此，限制用户的最大进程数，就是限制用户的最大线程数，至于为什么没有精确达到1024这个最大值就已经报出异常，应该是系统的自我保护功能，在还剩下3个线程的前提下，就开始报错。

到此为止，我们已经通过分析来找到问题的原因，但是，我们还是不知道为什么会创建这么多的线程，从第一个输出得知，JVM已经创建的应用线程有907个，那么他们都在做什么事情呢？

于是，在问题发生的时候，我们又使用JVM的jstack命令，查看输出得知，每个线程都阻塞在打印日志的语句上，log4j中打印日志的代码实现如下：

```
public void callAppenders(LoggingEvent event) {
    int writes = 0;
    for(Category c = this; c != null; c=c.parent) {
        // Protected against simultaneous call to addAppender,
removeAppender,...
        synchronized(c) {
            if(c.aai != null) {
                writes += c.aai.appendLoopOnAppenders(event);
            }
            if(!c.additive) {
                break;
            }
        }
    }
    if(writes == 0) {
        repository.emitNoAppenderWarning(this);
    }
}
```

在log4j中，打印日志有一个锁，锁的作用是让打印日志可以串行，保证日志在日志文件中的正确性和顺序性。

那么，新的问题又来了，为什么只有凌晨0点会出现打印日志阻塞，其他时间会偶尔发生呢？这时，我们带着新的线索又回到问题开始的思路，凌晨12点应用没有定时任务，系统会不会有其他的IO密集型的任务，比如说归档日志、磁盘备份等？

GitChat

经过与运维部门碰头，基本确定是每天凌晨0点日志切割导致磁盘IO被占用，于是堵塞打印日志，日志是每个工作任务都必须的，日志阻塞，线程池就阻塞，线程池阻塞就导致线程池被撑大，线程池里面的线程数超过1024就会报错。

到这里，我们基本确定了问题的原因，但是还需要对日志切割导致IO增大进行分析和论证。

首先我们使用前面小结介绍的vmstat查看问题发生时IO等待数据：

```
vmstat 2 1 >> /tmp/vm.log
procs -----------memory---------- ---swap-- -----io---- --system-
- -----cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs
us sy id wa st
 3  0 177608 725636  31856 3899144    0    0     2    10    0
0  39  1 1  59  0
Tue Jul 5 00:27:51 CST 2016
```

可见，问题发生的时候，CPU的IO等待为59%，同时又与运维部门同事复盘，运维同事确认，脚本切割通过cat命令方法，先把日志文件cat后，通过管道打印到另外一个文件，再清空原文件，因此，一定会导致IO的上升。

其实，问题的过程中，还有一个疑惑，我们认为线程被IO阻塞，线程池被撑开，导致线程增多，于是，我们查看了一下Tomcat线程池的设置，我们发现Tomcat线程池设置了800，按理说，永远不会超过1024。

```
<Connector port="8080"
               maxThreads="800" minSpareThreads="25"
maxSpareThreads="75"
               enableLookups="false" redirectPort="8443"
acceptCount="100"
               debug="0" connectionTimeout="20000"
               disableUploadTimeout="true" />
```

关键在于，笔者所在的支付平台服务化架构中，使用了两套服务化框架，一个是基于dubbo的框架，一个是点对点的RPC，用来紧急情况下dubbo服务出现问题，服务降级使用。

每个服务都配置了点对点的RPC服务，并且独享一个线程池：

```
<Connector port="8090"
               maxThreads="800" minSpareThreads="25"
maxSpareThreads="75"
               enableLookups="false" redirectPort="8443"
acceptCount="100"
               debug="0" connectionTimeout="20000"
               disableUploadTimeout="true" />
```

GitChat

由于我们在对dubbo服务框架进行定制化的时候，设计了自动降级原则，如果dubbo服务负载变高，会自动切换到点对点的RPC框架，这也符合微服务的失效转移原则，但是设计中没有进行全面的考虑，一旦一部分服务切换到了点对点的RPC，而一部分的服务没有切换，就导致两个现场池都被撑满，于是超过了1024的限制，就出了问题。

到这里，我们基本可以验证，问题的根源是日志切割导致IO负载增加，然后阻塞线程池，最后发生OOM：unable to create new native thread。

剩下的任务就是最小化重现的问题，通过实践来验证问题的原因。我们与性能压测部门沟通，提出压测需求：

> 1. Tomcat线程池最大设置为1500.
> 2. 操作系统允许的最大用户进程数1024.
> 3. 在给服务加压的过程中，需要人工制造繁忙的IO操作，IO等待不得低于50%。

经过压测压测部门的一下午努力，环境搞定，结果证明完全可以重现此问题。

最后，与所有相关部门讨论和复盘，应用解决方案，解决方案包括：

1. 全部应用改成按照小时切割，或者直接使用log4j的日志滚动功能。
2. Tomcat线程池的线程数设置与操作系统的线程数设置不合理，适当的减少Tomcat线程池线程数量的大小。
3. 升级log4j日志，使用logback或者log4j2。

这次OOM问题的可以归结为“多个因、多个果、多台机器、多个服务池、不同时间”，针对这个问题，与运维部、监控部和性能压测部门的同事奋斗了几天几夜，终于通过在线上抓取信息、分析问题、在性能压测部门同事的帮助下，最小化重现问题并找到问题的根源原因，最后，针对问题产生的根源提供了有效的方案。

3. 与监控同事现场编写的脚本

本节提供一个笔者在实践过程中解决OOM问题的一个简单脚本，这个脚本是为了解决OOM（unable to create native thread)的问题而在问题机器上临时编写，并临时使用的，脚本并没有写的很专业，笔者也没有进行优化，保持原汁原味的风格，这样能让读者有种身临其境的感觉，只是为了抓取需要的信息并解决问题，但是在线上问题十分火急的情况下，这个脚本会有大用处。

```
#!/bin/bash

ps -Leo pid,lwp,user,pcpu,pmem,cmd >> /tmp/pthreads.log
echo "ps -Leo pid,lwp,user,pcpu,pmem,cmd >> /tmp/pthreads.log" >>
/tmp/pthreads.log
echo `date` >> /tmp/pthreads.log
echo 1

pid=`ps aux|grep tomcat|grep cwh|awk -F ' ' '{print $2}'`
echo 2

echo "pstack $pid >> /tmp/pstack.log" >> /tmp/pstack.log
pstack $pid >> /tmp/pstack.log
echo `date` >> /tmp/pstack.log
echo 3

echo "lsof >> /tmp/sys-o-files.log" >> /tmp/sys-o-files.log
lsof >> /tmp/sys-o-files.log
echo `date` >> /tmp/sys-o-files.log
echo 4

echo "lsof -p $pid >> /tmp/service-o-files.log" >> /tmp/service-
o-files.log
lsof -p $pid >> /tmp/service-o-files.log
echo `date` >> /tmp/service-o-files.log
echo 5

echo "jstack -l $pid  >> /tmp/js.log" >> /tmp/js.log
jstack -l -F $pid  >> /tmp/js.log
echo `date` >> /tmp/js.log
```

GitChat

```
echo 6

echo "free -m >> /tmp/free.log" >> /tmp/free.log
free -m >> /tmp/free.log
echo `date` >> /tmp/free.log
echo 7

echo "vmstat 2 1 >> /tmp/vm.log" >> /tmp/vm.log
vmstat 2 1 >> /tmp/vm.log
echo `date` >> /tmp/vm.log
echo 8

echo "jmap -dump:format=b,file=/tmp/heap.hprof 2743" >>
/tmp/jmap.log
jmap -dump:format=b,file=/tmp/heap.hprof >> /tmp/jmap.log
echo `date` >> /tmp/jmap.log
echo 9

echo end
```

如果读者在线上已经遇到了OOM的问题，可以顺着这个看似简陋而又信息满满的Java服务的监控脚本的思路，利用本文提供的各种脚本和命令来深挖问题的根本原因。

GitChat