图文 086、案例实战：每秒仅仅上百请求的系统为什么会因为OOM而崩溃？

详情 评论

案例实战：
每秒仅仅上百请求的系统，为什么会因为OOM而崩溃？

狸猫技术窝专栏上新，基于**真实订单系统**的消息中间件（mq）实战，重磅推荐：

未来3个月，我的好朋友原子弹大侠将带你一起，全程实战，360度死磕MQ

（点击下方蓝字进行试听）

从 0 开始带你成为消息中间件实战高手

重要说明：

如何提问：每篇文章都有评论区，大家可以尽情在评论区留言提问，我都会逐一答疑

（**ps**：评论区还精选了一些小伙伴对**专栏每日思考题的作答**，有的答案真的非常好！大家可以通过看别人的思路，启发一下自己，从而加深理解）

如何加群：购买了狸猫技术窝专栏的小伙伴都可以加入**狸猫技术交流群**。

（群里有不少**一二线互联网大厂的助教**，大家可以一起讨论交流各种技术）

具体**加群方式**请参见文末。

（**注**：以前通过其他专栏加过群的同学就不要重复加了）

1、案例背景介绍

本文将会给大家分析一个线上的真实生产案例，在这个案例中，一个每秒仅仅只有100+请求的系统却频繁的因为OOM而崩溃。

在对这个案例进行深度分析的时候，大家会看到一个OOM问题是如何牵扯到Tomcat底层工作原理、Tomcat内核参数的设置、服务请求超时时间等问题的。

2、系统发生OOM的生产现场

我们从系统发生OOM的生产现场开始说起，某一天突然收到线上系统的一个报警，说这个系统发生了异常

一看异常信息立马让人大惊失色，居然是大名鼎鼎的OOM问题！这个问题可不得了，因为谁都知道，一旦JVM发生了OOM，将会导致系统直接崩溃掉，因为OOM代表的是JVM本身已经没办法继续运行了！

因此我们立马登录到系统的线上机器去查看对应的日志，在这里插一句，大家记住一点，一旦你收到系统OOM的报警，或者是有人突然反馈说你负责的线上系统崩溃了，**第一件事情：**一定是登录到线上机器去看日志，而不是做别的事情。

当时在机器的日志文件中看到类似下面的一句话：

Exception in thread "http-nio-8080-exec-1089" java.lang.OutOfMemoryError: Java heap space

但凡对Tomcat的底层工作原理有一定了解的朋友，应该立马就会反映过来了，这里的“http-nio-8080-exec-1089”说的其实就是Tomcat的工作线程。

而后面的“java.lang.OutOfMemoryError: Java heap space”大家如果认真学习了之前的内容，都很清楚，指的是堆内存溢出的问题。

所以连起来看，这段日志的意思，就是Tomcat的工作线程在处理请求的时候需要在堆内存里分配对象，但是发现堆内存塞满了，而且根本没办法回收任何一点多余的对象，此时实在没办法在堆内存里放下更多对象了，报了这个异常。

3、初步看看Tomcat的底层原理

说到这里，我们先通过一步一图的方式，带大家看看Tomcat的基本工作原理，以及发生这个OOM异常的基本的原因。

首先，相信每个学过Java Web的朋友应该都知道，我们写的系统都是部署在Tomcat中的。

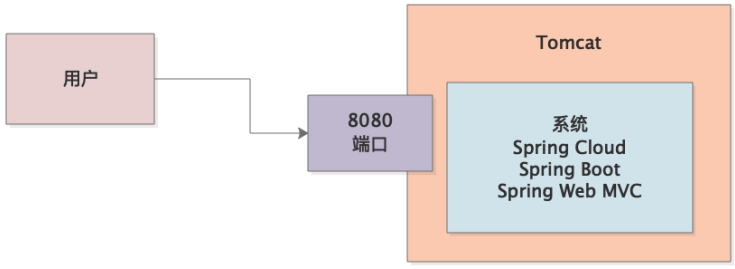
最早我们会在Eclipse / Intellij IDEA这种开发工具里写一堆的Servlet，然后打包之后放入Tomcat，再启动Tomcat

接着我们访问Tomcat监听的一个端口号（一般是8080），然后系统的功能就可以运行起来了。

当然，后来随着技术发展，我们不再写Servlet这么原始的东西了，有一些类似Spring MVC之类的框架把Servlet封装起来了，我们就基于Spring MVC之类的框架去开发。

再到后来，越来越先进了，出现了Spring Boot，我们可以把Tomcat之类的Web容器都内嵌在系统里。再到后来甚至是基于Spring Cloud去开发分布式的系统。

当然，先别扯远了，先看一张图，这是一个最基本的Web系统部署在Tomcat中运行的一个图。



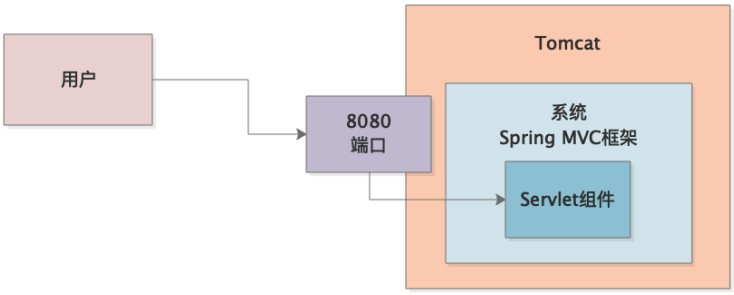
上面这个图相信大家都可以理解，我们基于Spring Cloud、Spring Boot、Spring Web MVC等技术，写好一套系统，给打包之后，放入线上服务器中部署的Tomcat目录下，然后启动Tomcat就可以了。

Tomcat会监听一个默认8080的端口号，然后接着我们我们就通过浏览器就可以对这个机器上的Tomcat发起请求了，类似下面这种请求：

http://192.168.31.109:8080/order?userid=100

接着Tomcat会监听8080端口收到这个请求，通常来说他会把请求交给Spring Web MVC之类的框架去处理。

这类框架一般底层都封装了Servlet/Filter之类的组件，他也是用这类组件去处理请求的，如下图所示。

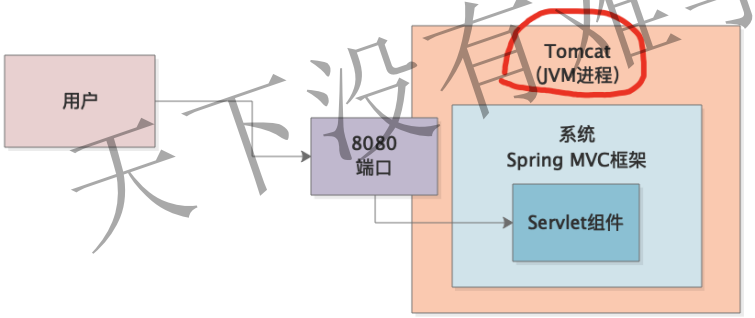


然后类似Spring MVC的框架的Servlet组件，就会根据你的请求路径，比如“/order”这种东西，去找到你代码中用来处理这个请求的Controller组件，这个相信学过Java Web的朋友都能理解。

那么现在我们来思考一个问题，稍微牵扯到一些Tomcat的底层工作原理：Tomcat是个什么东西？

不知道大家有没有思考过，如果我们是把写好的系统放入Tomcat目录中，然后启动Tomcat，此时我们启动的Tomcat本身就是一个JVM进程，因为Tomcat自己也是Java写的。

所以看下图，大家首先明确一个概念，就是**Tomcat自己就是一个JVM进程**，我们写好的系统只不过是一些代码而已，这些代码就是一个一个的类，这些类被Tomcat加载到内存里去，然后由Tomcat来执行我们写的类。

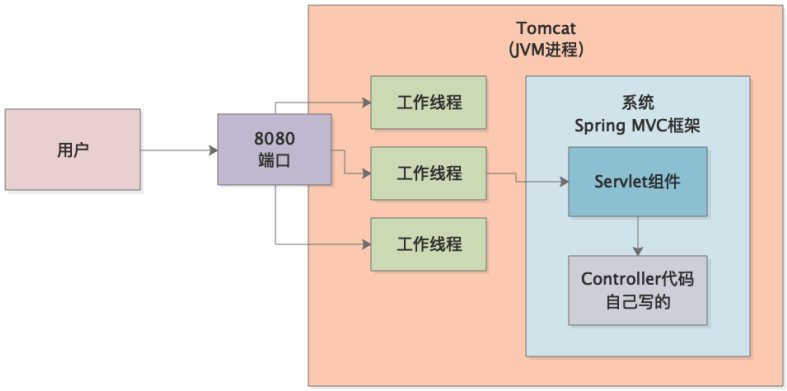


既然如此，**Tomcat本身是如何去监听8080端口上收到的请求的？**

很简单，Tomcat有自己的工作线程，大家务必要对Tomcat的工作线程这个概念有一个认识，即Tomcat有很多自己的工作线程，少则一两百个，多则三四百个也是可以的。

然后从8080端口上收到的请求都会均匀的分配给这些工作线程去进行处理，而这些工作线程收到请求之后，就负责调用Spring MVC框架的代码，Spring MVC框架又负责调用我们自己写好的代码，比如一些Controller类之类的

所以最终运行起来的原理如下图所示，大家看一下。



大家看上图中的箭头运行的顺序，是不是就上面我们说的那个顺序。

好了，关于Tomcat的底层原理就初步说到这里，毕竟这不是讲Tomcat的专栏。主要为了方便各位理解这个案例，因此简单阐述一下，不然大家很难理解透彻下面的内容。

当然后续如果有需要，也会考虑专门写专栏讲解Tomcat，毕竟搞 java 的，谁又离得开这只Tom猫呢?

4、回头看看异常日志

好！言归正传，接着我们回过头看看在当时的线上系统的日志中发现的异常：

Exception in thread "http-nio-8080-exec-1089" java.lang.OutOfMemoryError: Java heap space

这时候大家再来理解就很简单了，“http-nio-8080-exec-1089”

这个东西说白了就是上图中的Tomcat工作线程，因为是他负责调用Spring MVC以及我们写的Controller、Service、DAO等一大堆的代码的，所以他发现运行的时候堆内存不够了，就会抛出来堆内存溢出的异常了。

5、不要忘了一个关键的JVM参数

一旦我们发现线上系统发生了内存溢出的异常，第一步一定是看日志

具体是看什么？主要有两点：

1、看看到底是堆内存溢出？还是栈内存溢出？或者是Metaspace内存溢出？首先得确定一下具体的溢出类型

2、看看是哪个线程运行代码的时候内存溢出了。因为Tomcat运行的时候不光有自己的工作线程，我们写的代码也可能会创建一些线程出来

万一是我们自己启动的线程遇到了内存溢出呢？所以这两个要点是首先要关注的。

接着看完了这两个东西之后，就得记得**每个系统上线，务必得设置一个参数：**

-XX:+HeapDumpOnOutOfMemoryError

这个参数会在系统内存溢出的时候导出来一份内存快照到我们指定的位置。

接着排查和定位内存溢出问题，主要就得依托这个自动导出来的内存快照了。

6、对内存快照进行分析

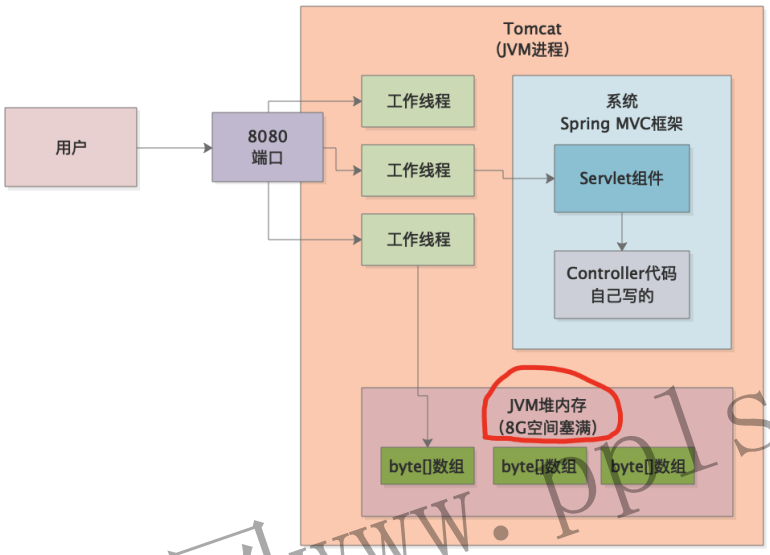
相信大家还记得之前我们详细讲解的MAT分析内存快照的技巧，通常来说在内存溢出的时候分析内存快照也并没那么的复杂，主要其实就是通过MAT来找到那些占据内存最大的对象。

首先我们会发现占据内存最大的是大量的“byte[]”数组，一大堆的byte[]数组就占据了大约8G左右的内存空间。而我们当时线上机器给Tomcat的JVM堆内存分配的也就是8G左右的内存而已。

因此我们直接可以得出第一个结论：Tomcat工作线程在处理请求的时候会创建大量的byte[]数组，大概有8G左右，直接把JVM堆内存占满了。

此时导致继续处理新请求的时候，没办法继续在堆中分配新对象了，所以就内存溢出了。

大家看下面的图，我们在图中体现出来了这个意思。



接着我们当然是想分析一下，到底是哪些byte[]数组在这里了，因此我们就继续通过MAT深入查看，发现大概是类似下面的一大堆byte[]数组：

```
byte[10008192] @ 0x7aa800000 GET /order/v2 HTTP/1.0-forward...
byte[10008192] @ 0x7aa800000 GET /order/v2 HTTP/1.0-forward...
byte[10008192] @ 0x7aa800000 GET /order/v2 HTTP/1.0-forward...
byte[10008192] @ 0x7aa800000 GET /order/v2 HTTP/1.0-forward...
```

上面只是写出来几个示例的，其实当时这里看到了很多类似这样的数组，而且数组的大小都是一致的10MB。

大概清点了一下，类似上面那样的数组，大概有800个左右，也就对应了8G的空间。

接着我们进一步思考，这种数组应该是谁创建的?

首先负责写代码的工程师一口咬定，绝对不是自己写出来的，那么在MAT上可以继续查看一下这个数组是被谁引用的，大致可以发现是Tomcat的类引用的，具体来说是类似下面的这个类：

org.apache.tomcat.util.threads.TaskThread

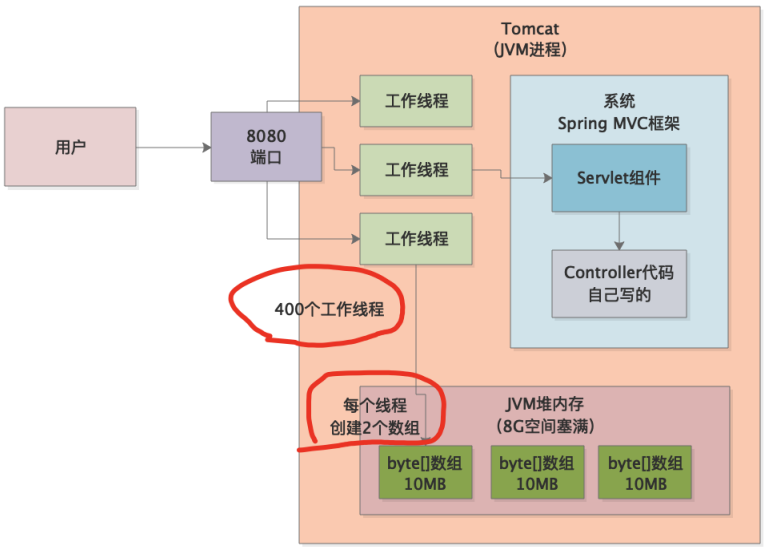
这个类一看就是Tomcat自己的线程类，因此可以认为是Tomcat的线程创建了大量的byte[]数组，占据了8G的内存空间。

之前的文章给大家提到过，MAT中可以查看具体有哪些线程存在，其实MAT使用很方便，也并不难，大家在实际使用的时候，自己多摸索，多点一点，多尝试，就会逐渐摸熟他的使用方式 。最关键的，是掌握分析的思路。

此时我们发现Tomcat的工作线程大致有400个左右，也就是说每个Tomcat的工作线程会创建2个byte[]数组，每个byte[]数组是10MB左右

最终就是400个Tomcat工作线程同时在处理请求，结果创建出来了8G内存的byte[]数组，进而导致了内存溢出。

大家看下面的图，我们把上述分析推断逻辑都反应在图里，大家看一下就一目了然了。



7、系统每秒QPS才只有100？！！

此时我们大致结合上图，脑海中已经可以出现一幅流动的图形了，一秒钟之内瞬间来了400个请求，导致Tomcat的400个工作线程全部上阵处理请求，每个工作线程在处理一个请求的时候，会创建2个数组，每个数组是10MB，结果导致瞬间就让8G的内存空间被占满了。

看起来生产系统的情况就是如此，那么，**这里有什么问题吗?**

当然，我们来具体分析一下，首先，我们检查了一下系统的监控，发现每秒请求并不是400，而是100！

也就是说明明每秒才100个请求，**怎么可能Tomcat的400个线程都处于工作状态?**

此时我们不禁会倒吸一口凉气，出现这种情况只有一种可能，那就是每个请求处理需要4秒钟的时间！

如果每秒来100个请求，但是每个请求处理完毕需要4秒的时间，那么在4秒内瞬间会导致有400个请求同时在处理，也就会导致Tomcat的400个工作线程都在工作！接着就会导致上述的情况。

好，第一个问题解决了，那么第二个问题来了，为什么Tomcat工作线程在处理一个请求的时候会创建2个10MB的数组?

这里我们可以到Tomcat的配置文件里搜索一下，发现了如下的一个配置：

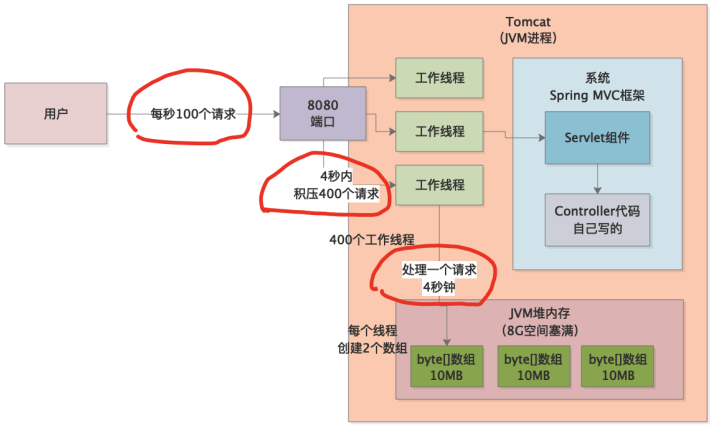
max-http-header-size: 10000000

有了，就是这个东西，导致Tomcat工作线程在处理请求的时候会创建2个数组，每个数组的大小如上面配置就是10MB。

因此，在这里，我们来继续梳理一遍系统运行时候的场景：

每秒100个请求，每个请求处理需要4秒，导致4秒内有400个请求同时被400个线程在处理，每个线程会根据配置创建2个数组，每个数组是10MB，占满了8G的内存。

我们把上述结论在下面的图中继续反应出来，大家来看看。



8、为什么处理一个请求需要4秒钟?

问题全部分析清楚了，接着当然是需要解决问题了

第一个问题，为什么处理一个请求需要4秒钟?

这个问题绝对是偶发性的，不是平时每次处理请求都如此，因为负责这个系统的工程师说了，平时处理一个请求也就几百毫秒的时间而已。

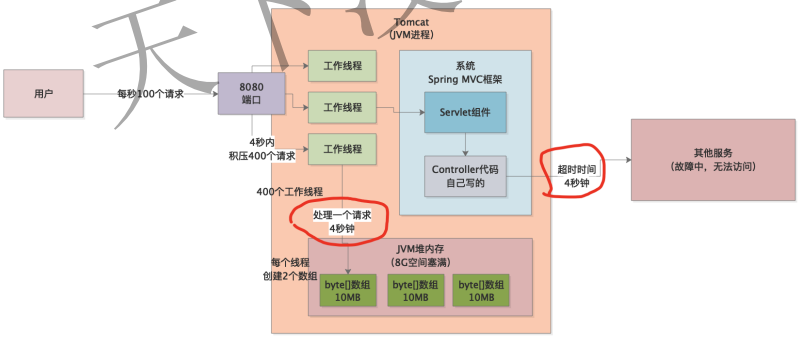
好，那么既然如此，唯一的办法只能是在日志里去找问题了，继续翻看事故发生时的日志，发现日志中除了OOM以外，其实有大量的服务请求调用超时的异常，类似下面那样子:

Timeout Exception....

也就是说，我们的这个系统在通过RPC调用其他系统的时候突然出现了大量的请求超时，立马翻看一下系统的RPC调用超时的配置，惊讶的发现，负责这个系统的工程师居然将服务RPC调用超时时间设置为了刚好是4秒!

也就是说，一定是在这个时间里，远程服务自己故障了，导致我们的系统RPC调用他的时候是访问不通的，然后就会在配置好的4秒超时时间之后抛出异常，在这4秒内，工作线程会直接卡死在无效的网络访问上。

我们看下面的图，在图中我们表述出来了服务间调用超时的问题。



在上图中已经表述的很清楚了，之所以每个请求需要处理4秒钟，是因为下游服务故障了，网络请求都是失败的，此时会按照设置好的4秒超时时间一直卡住4秒钟之后才会抛出Timeout异常，然后请求处理结束。

这就是一个请求处理需要4秒钟的根本原因，进而导致每秒100个请求的压力下，4秒内积压400个请求同时在处理，导致400个工作线程创建了800个数组，每个数组10MB内存，耗尽了8G的内存，最终导致内存溢出!

9、对系统进行优化

其实要解决上述问题，只要分析清楚了原因，还是非常简单的，对症下药即可。

最核心的问题就是那个超时时间设置的实在太长了，因此立马将超时时间改为1秒即可。

这样的话，每秒100个请求过来，也就只有200个数组，占据2G内存，远远不会把堆内存塞满的，然后1秒内这100个请求会全部超时，请求就处理结束了。

下一秒再来100个请求又是新的一轮处理，不会每秒积压100个请求，4秒积压400个请求同时处理了。

另外一个，对Tomcat的那个参数，max-http-header-size，可以适当调节的小一些就可以了，这样Tomcat工作线程自身为请求创建的数组，不会占据太大的内存空间的。

10、本文小结

本文实际上是一个有一定复杂度的综合性的OOM故障排查的案例，但是跟后续大量的其他OOM案例一样，大家会发现，解决这类问题的思路都是一致的，需要你一步一步去分析，但是实际面对各种各样的千奇百怪的问题，需要你举一反三，能够对你系统涉及的种种技术本身有较为深入的了解才可以。

End
