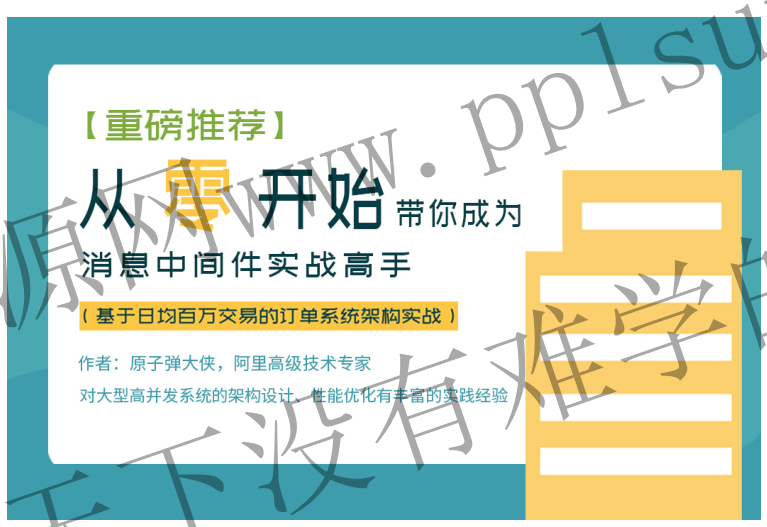
图文 087、案例实战：Jetty 服务器的 NIO 机制是如何导致堆外内存溢出的?

详情 评论

案例实战：
Jetty服务器的NIO机制是如何导致堆外内存溢出的?

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、案例背景引入

这个案例是我曾经带过的一个项目，使用Jetty作为Web服务器的时候在某个非常罕见的场景下发生的一次堆外内存溢出的场景。

这种场景其实并不多见，但还是准备用来作为一个案例，给大家介绍一下这种场景的排查方法。

2、案例发生现场

有一天突然收到线上的一个报警：某台机器部署的一个服务突然之间就不可以访问了。

此时第一反应当然是立马登录上机器去看一下日志，因为服务挂掉，很可能是OOM导致的崩溃，当然也可能是其他原因导致的问题。

这个时候在机器的日志中发现了如下的一些信息：

nio handle failed java.lang.OutOfMemoryError: Direct buffer memory
at org.eclipse.jetty.io.nio.xxxx
at org.eclipse.jetty.io.nio.xxxx
at org.eclipse.jetty.io.nio.xxxx

过多的日志信息给省略掉了，因为都是非常杂乱的一些信息，也没太大意义，大家关注比较核心的一些信息就可以

上述日志中，最主要的就是告诉我们有OOM异常，但是是哪个区域导致的呢？

居然是我们没见过的一块区域：**Direct buffer memory**，而且在下面我们还看到一大堆的Jetty相关的方法调用栈

到此为止，仅仅看到这些日志，我们基本就可以分析出来这次OOM发生的原因了。

3、初步分析事故发生的原因

先给大家解释一个东西：Direct buffer memory

这个东西其实就是堆外内存，顾名思义，他是JVM堆内存之外的一块内存空间，这块内存空间不是JVM管理的，因此之前我们也没怎么多讲这块内容，但是你的Java代码确实是可以在JVM堆之外使用一些内存空间的。

这些空间就叫做Direct buffer memory，如果按英文字面理解，他的意思是直接内存，其实你要这么叫也没问题，但是从字面可以看出来，这块内存并不是JVM管理的，而是“直接”被操作系统管理。

正因为这样，所以其英文名叫做Direct buffer memory，就是直接内存的意思。但是如果我们叫他直接内存，又显得非常的奇怪，所以通常更好的称呼就是“堆外内存”。

另外再给大家解释一个东西：Jetty。这个是什么？

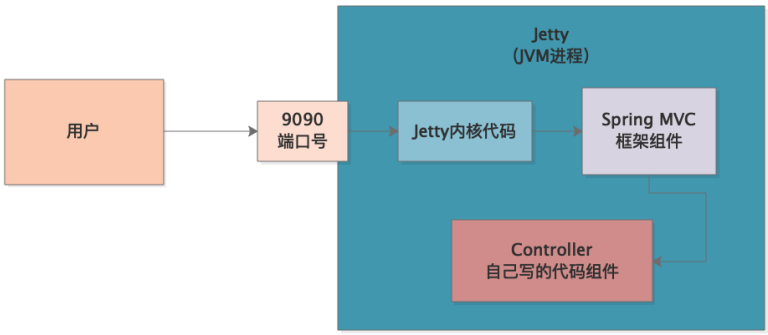
其实你大致可以理解为跟Tomcat一样的东西，就是Web容器

Jetty本身也是Java写的，我们如果写好了一个系统，可以打包后放入Jetty，启动Jetty即可。

Jetty启动之后，本身就是一个JVM进程，他会监听一个端口号，比如说9090

然后你就向Jetty监听的9090端口发送请求，Jetty会把请求转交给你用的Spring MVC之类的框架，Spring MVC之类的框架再去调用写好的Controller之类的代码。

我们先看下面一个图，简单看看Jetty作为一个JVM进程运行我们写好的系统的一个流程：



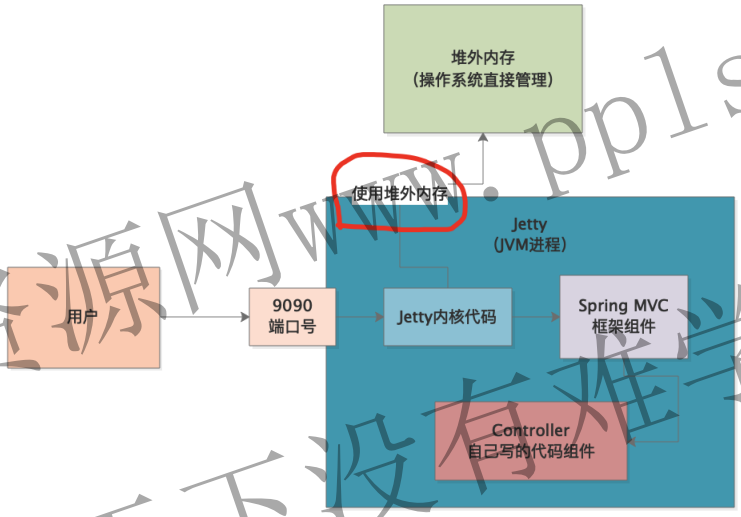
首先我们可以明确一点，这次OOM是Jetty这个Web服务器在使用堆外内存的时候导致的

也就是说，基本可以推测出来，Jetty服务器可能在不停的使用堆外内存，然后堆外内存空间不足了，没法使用更多的堆外内存了，此时就会抛出内存溢出的异常。

至于为什么Jetty要不停的使用堆外内存空间，大家就暂时先别管那么多了，那涉及到Jetty作为一个Web服务器的底层源码细节。

我们只要知道他肯定会不停的去使用堆外内存，然后用着用着堆外内存不够了，就内存溢出了。

我们接着看下面的图，下面的图里，就体现出来了Jetty不停使用堆外内存的场景。



4、关于解决OOM问题的底层技术修为的一点建议

讲到这里，我们肯定会有一点很疑惑了：Jetty既然是用Java写的，那么他是如何通过Java代码去在JVM堆之外申请一块堆外内存来使用的？然后这个堆外内存空间又是如何释放掉的呢？

这个东西其实涉及到Java的NIO的底层技术细节，如果大家之前对NIO没什么了解的话，突然看到这个异常，估计是没法继续分析下去了。

因此这里也给大家额外插一句话，其实JVM的性能优化相对还是较为容易一些的，而且基本上整个套路之前也已经给大家都说的很清楚了。

但是如果是解决OOM问题，那么除了一些特别弱智和简单的，比如有人在代码里不停的创建对象最后导致内存溢出这种。其他的很多生产环境下的OOM问题，都是有点技术难度的。

大家如果把整个专栏最后两周的内容都看完了，就会有一个很深的感触，那就是1000个工程师可能会遇到1000种不同的OOM问题。

可能排查的思路是类似的，或者解决问题的思路是类似的，但是如果你要解决各种OOM问题，是需要对各种技术都有一定的了解，换句话说，需要有较为扎实的技术内功修为。

比如昨天的那个案例，就需要你对Tomcat的一些工作原理有一定的了解，你才能分析清楚那个案例。

同样，今天的这个案例，就要求你对Java NIO技术的工作原理有一定的了解才能分析清楚。

因此这里也说句题外话：希望大家在学习JVM技术本身之余，多去对其他的核心主流技术做一些深层次的甚至源码级别的研究

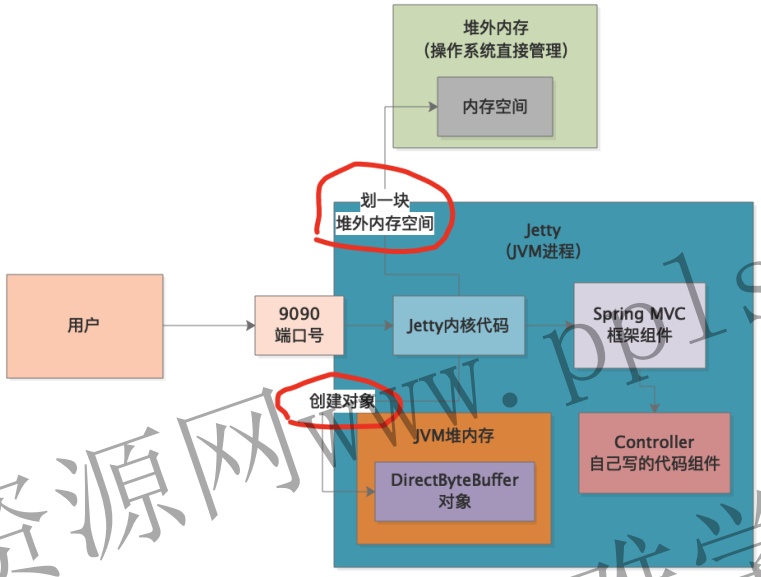
这些底层技术积累将会在你的线上系统出现问题的时候，迅速帮助你分析和解决问题。

5、堆外内存是如何申请的，又是如何释放的?

接着我们继续来看看，这个堆外内存是如何申请和释放的?

简单来说，如果在Java代码里要申请使用一块堆外内存空间，是使用DirectByteBuffer这个类，你可以通过这个类构建一个DirectByteBuffer的对象，这个对象本身是在JVM堆内存里的。

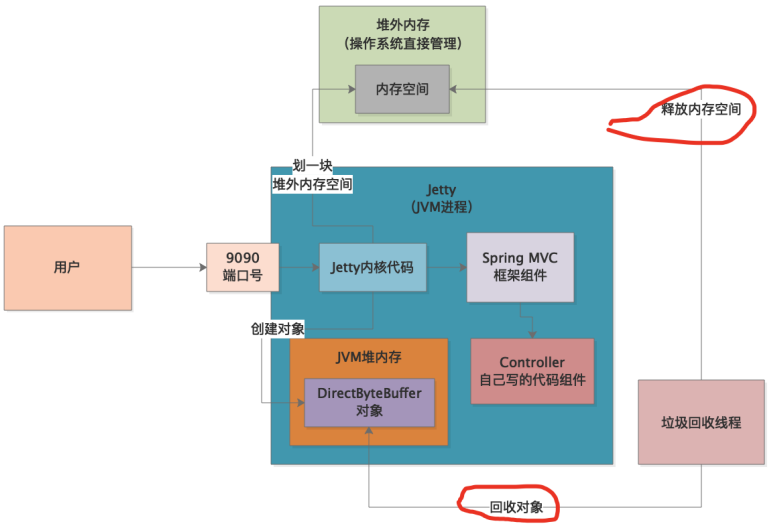
但是你在构建这个对象的同时，就会在堆外内存中划出来一块内存空间跟这个对象关联起来，我们看看下面的图，你就对他们俩的关系很清楚了。



因此在分配堆外内存的时候大致就是这个思路，那么堆外内存是如何释放的呢?

很简单，当你的DirectByteBuffer对象没人引用了，成了垃圾对象之后，自然会在某一次young gc或者是full gc的时候把DirectByteBuffer对象回收掉。

只要回收一个DirectByteBuffer对象，就会自然释放掉他关联的那块堆外内存，我们看看下面的图就知道了。



6、为什么会出现堆外内存溢出的情况?

那么大家现在应该很清楚了，一般什么情况下会出现堆外内存的溢出？

很简单，如果你创建了很多的DirectByteBuffer对象，占用了大量的堆外内存，然后这些DirectByteBuffer对象还没有GC线程来回收掉，那么就不会释放堆外内存！

久而久之，当堆外内存都被大量的DirectByteBuffer对象关联使用了，如果你再要使用更多的堆外内存，那么就会报内存溢出了！

那么，**什么情况下会出现大量的DirectByteBuffer对象一直存活着，导致大量的堆外内存无法释放呢？**

有一种可能，就是系统承载的是超高并发，复杂压力很高，瞬时大量请求过来，创建了过多的DirectByteBuffer占用了大量的堆外内存，此时再继续想要使用堆外内存，就会内存溢出了！

但是这个系统是这种情况吗？

明显不是！因为这个系统的负载其实没有想象中的那么高，不会有瞬时大量的请求过来。

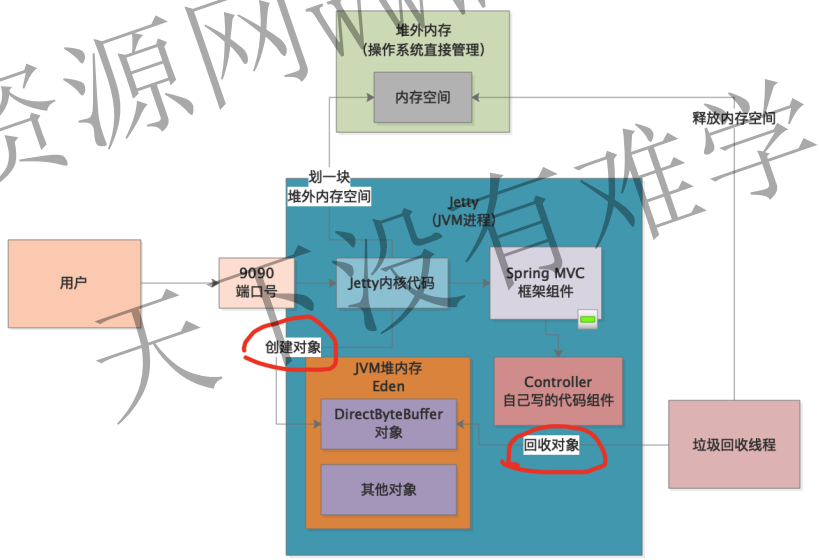
7、真正的堆外内存溢出原因分析

这个时候你就得思路活跃起来了，我们完全可以去用jstat等工具观察一下线上系统的实际运行情况，同时根据日志看看一些请求的处理耗时，综合性的分析一下。

当时我们通过jstat工具分析jvm运行情况，同时分析了过往的gc日志，另外还看了一下系统各个接口的调用耗时之后，分析出了如下的思路。

首先看了一下接口的调用耗时，这个系统并发量不高，但是他每个请求处理较为耗时，平均在每个请求需要一秒多的时间去处理。

然后我们通过jstat发现，随着系统不停的被调用会一直创建各种对象，包括Jetty本身会不停的创建DirectByteBuffer对象去申请堆外内存空间，接着直到年轻代的Eden区满了，就会触发young gc，如下图所示。

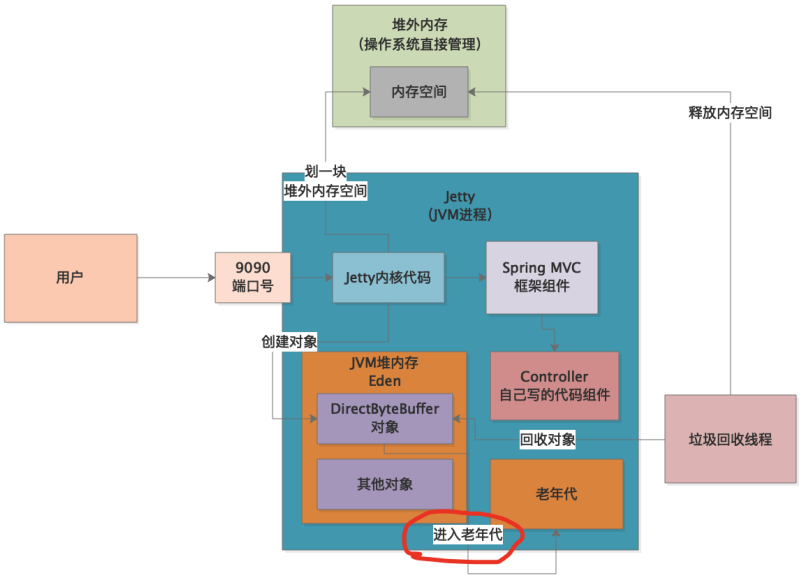


但是往往在进行垃圾回收的一瞬间，可能有的请求还没处理完毕，此时就会有不少DirectByteBuffer对象处于存活状态，不能被回收掉，当然之前不少DirectByteBuffer对象对应的请求可能处理完毕了，他们就可以被回收了。

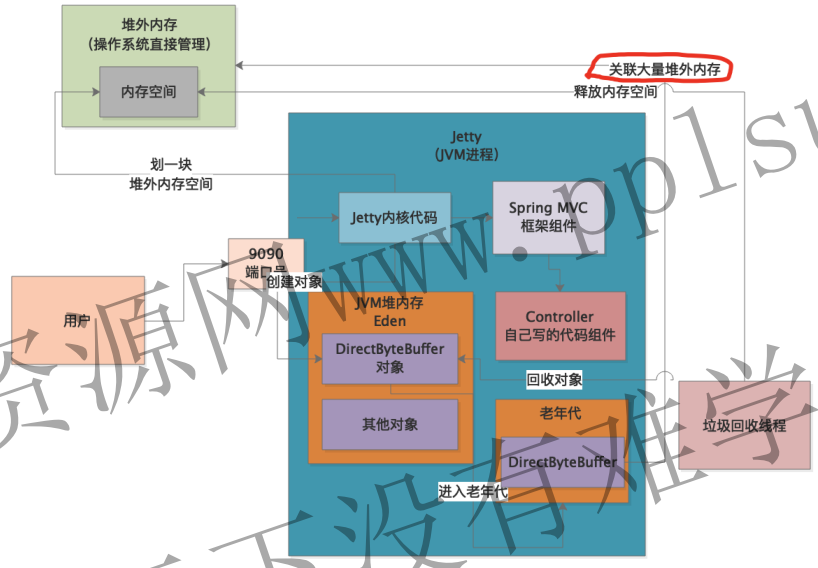
此时肯定会有一些DirectByteBuffer对象以及一些其他的对象是处于存活状态的，那么就需要转入Survivor区域中。

但是大家注意了，这个系统当时在上线的时候，内存分配的极为不合理，在当时而言，大概就给了年轻代一两百MB的空间，老年代反而给了七八百MB的空间，进而导致年轻代中的Survivor区域只有10MB左右的空间。

因此往往在young gc过后，一些存活下来的对象（包括了一些DirectByteBuffer在内）会超过10MB，没法放入Survivor中，就会直接进入老年代，我们看下图就表现出了这个过程。



因此上述的过程就这么反复的执行，必然会慢慢的导致一些DirectByteBuffer对象慢慢的进入老年代中，老年代中的DirectByteBuffer对象会越来越多，而且这些DirectByteBuffer都是关联了很多堆外内存的，如下图所示。



这些老年代里的DirectByteBuffer其实很多都是可以回收的状态了，但是因为老年代一直没塞满，所以没触发full gc，也就自然不会回收老年代里的这些DirectByteBuffer了！当然老年代里这些没有被回收的DirectByteBuffer就一直关联占据了大量的堆外内存空间了！

直到最后，当你要继续使用堆外内存的时候，结果所有堆外内存都被老年代里大量的DirectByteBuffer给占用了，虽然他们可以被回收，但是无奈因为始终没有触发老年代的full gc，所以堆外内存也始终无法被回收掉。

最后就会导致内存溢出问题的发生！

8、难道Java NIO就没考虑过这个问题吗?

所以这里我们先不说如何解决这个问题，先说一点，难道Java NIO就从没考虑过会有上述问题的产生过吗?

当然不是了，Java NIO是考虑到的！他知道可能很多DirectByteBuffer对象也许没人用了，但是因为没有触发gc就导致他们一直占据着堆外内存。

所以在**Java NIO的源码中会做如下处理**，他每次分配新的堆外内存的时候，都会调用System.gc()去提醒JVM去主动执行以下gc去回收掉一些垃圾没人引用的DirectByteBuffer对象，释放堆外内存空间。

只要能触发垃圾回收去回收掉一些没人引用的DirectByteBuffer，就会释放一些堆外内存，自然就可以分配更多的对象到堆外内存去了。

但是我们又在JVM中设置了如下参数：

-XX:+DisableExplicitGC

导致这个System.gc()是不生效的，因此就会导致上述的情况。

9、最终对问题的优化

其实项目问题有两个，一个是内存设置不合理，导致DirectByteBuffer对象一直慢慢进入老年代，导致堆外内存一直释放不掉

另外一个是设置了-XX:+DisableExplicitGC导致Java NIO没法主动提醒去回收掉一些垃圾DIrectByteBuffer对象，同样导致堆外内存无法释放。

因此最终对这个项目做的事情就是：

一个是合理分配内存，给年轻代更多内存，让Survivor区域有更大的空间

另外一个就是放开-XX:+DisableExplicitGC这个限制，让System.gc()生效。

做完优化之后，DirectByteBuffer一般就不会不断进入老年代了。只要他停留在年轻代，随着young gc就会正常回收释放堆外内存了。

另外一个，只要你放开-XX:+DisableExplicitGC的限制，Java NIO发现堆外内存不足了，自然会通过System.gc()提醒JVM去主动垃圾回收，可以回收掉一些DirectByteBuffer释放一些堆外内存。

End
