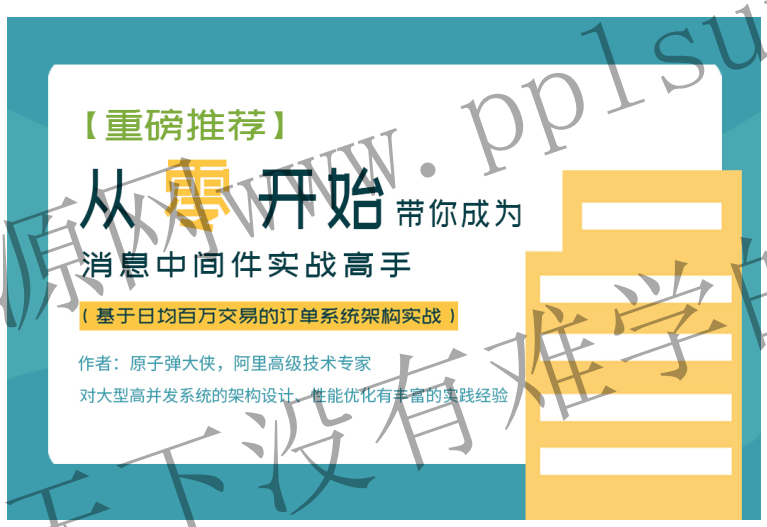
图文 088、案例实战：一次微服务架构下的RPC调用引发的OOM故障排查实践

详情 评论

案例实战：
一次微服务架构下的RPC调用引发的OOM故障排查实践

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OOM的案例是发生在微服务架构下的一次RPC调用过程中的，也是一次非常奇怪的故障案例

大家应该还记得昨天的文章里讲的一个解决系统OOM故障的核心能力积累，你必须对你线上系统使用的各种技术，从服务框架，到第三方框架，到Tomcat/Jetty等Web服务器，再到各种底层的中间件系统，对他们的源码最好都要有深入的理解。

因为一般线上系统OOM，都不是简单的由你的代码导致的，可能是因为你系统使用的某个开源技术的内核代码有一定的故障和缺陷，这时你要解决OOM问题，就必须深入到开源技术的源码中去分析。

对于我们的专栏而言，最后两周大量的OOM案例，主要带给大家的就是各种真实的生产案例以及分析思路

主要帮助大家掌握思路，但是真正到了具体的OOM故障时，每个人的故障可能都是不同的，都需要你去分析对应的技术的底层，然后找到故障原因，最后解决故障。

2、系统架构介绍

这个系统是比较早的一个系统，在进行服务间的RPC通信时，采用的是基于Thrift框架自己封装出来的一个RPC框架，可能很多朋友对Thrift之类的东西都没概念，那就不要过多的关注这个了。

大家只需要知道当时我们自己封装了一个RPC框架就可以了，然后在一个系统中，就是基于这个RPC框架去进行通信。

我们看一下下面的图，图中就是一个最基本的服务间RPC调用的示例。

3、故障发生现场

就当时而言，平时是服务A通过RPC框架去调用服务B，但是有一天，负责服务A的工程师更新了一些代码，然后将服务A重新上线之后，服务A自己倒是没什么，结果反而是服务B却突然宕机了！

这可真是奇怪和诡异的场景，因为明明修改代码的是服务A，重新部署的也是服务A，结果怎么成了服务B却挂掉了？

别急，让我们一步一步慢慢分析。我们先看一下下面的图，里面反映出了当时的生产情况。

当时我们立马登录到服务B的机器去查看服务B的日志，结果在里面赫然发现了OOM异常：

java.lang.OutOfMemoryError Java heap space

直接告诉你堆内存溢出了！当时我们就很奇怪，服务B为什么会OOM？

难道是服务B自己的问题？那不如重启一下服务B？

于是我们尝试重启了服务B，结果发现服务B重启过后很快又宕机了，而且原因还是OOM。

这个就奇怪了，因为在服务A修改代码部署之前，服务B从来没出现过这种情况！都是服务A修改了代码部署之后才导致服务B出现了这种情况的！

4、初步查找内存溢出的故障发生点

一般内存溢出的时候，大家务必要先找一下故障的发生点，其实看日志就可以了，因为在日志中都会有详细的异常栈信息

我们会发现，引发OutOfMemory异常的，居然就是我们自研的那个RPC框架！

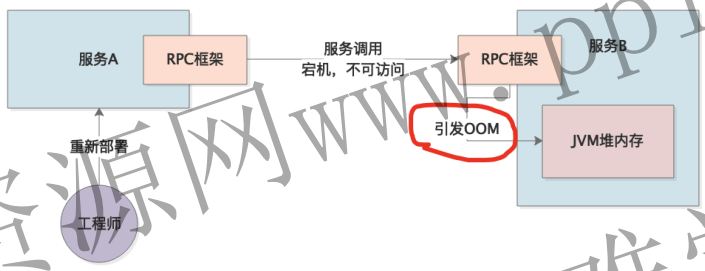
当时大致来说你可以看到的一个异常信息如下所示：

java.lang.OutOfMemoryError: Java heap space
xx.xx.xx.rpc.xx.XXXClass.read()
xx.xx.xx.rpc.xx.XXXClass.xxMethod()
xx.xx.xx.rpc.xx.XXXClass.xxMethod()

这里就不给出当时详细的异常信息截图了，因为当时也没截图保存下来。但是去除掉大量无效的信息后，其实核心的日志大致就类似上述那样子。

这里我们初步可以确定，就是自研的RPC框架在接收请求的时候引发了OOM！

我们看看下图，在里面反映出来了RPC框架运行的时候引发OOM的场景。



5、分析内存快照找到占用内存最大的对象

既然已经定位到OOM故障发生的位置了，也就是谁引发的OOM，接下来肯定得用MAT分析一下发生OOM的时候，对内存占用最大的是哪个对象了。

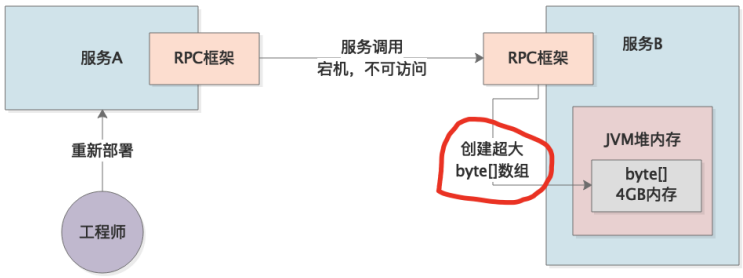
此时把OOM的时候自动导出的内存快照打开分析，发现占用内存最大的是一个超大的byte[]数组！

当时我们一台机器给的堆内存也不过就是4GB而已，而在内存快照中发现，居然一个超大的byte[]数组就占据了差不多4GB的空间

这个byte[]数组是哪儿来的?

通过分析这个byte[]数组的引用者（这个在MAT里有对应的功能，大家右击一个对象会出现一个菜单，里面会有很多选项，自己多尝试尝试），会发现这个数组就是RPC框架内部的类引用的。

我们在下面的图里给大家反映出来当前的一个场景。



6、通过分析源代码找出原因

一般分析到这一步，答案几乎就快要揭晓了，因为我们通过日志第一步已经定位到是谁导致的OOM，往往可能就是某个技术，比如Tomcat？Jetty？或者是RPC框架？

第一步我们先得定位到这个主体人

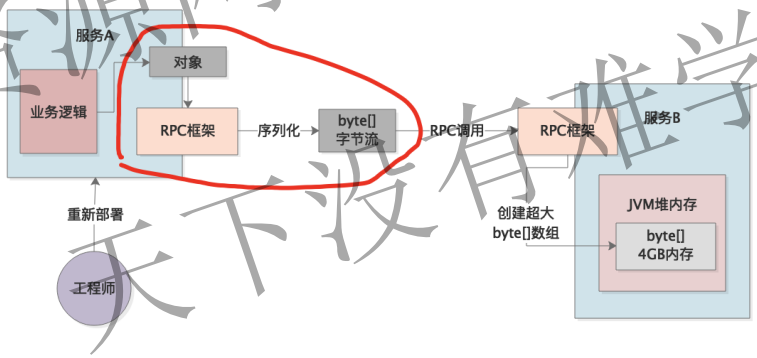
第二步一般就是用MAT之类的工具去分析内存快照，找到当时占用内存最大的对象是谁，可以找找都是谁在引用他，当然一般第一步通过看日志就大概知道导致内存溢出的类是谁，日志的异常栈里都会告诉你的。

第三步，很简单，你得对那个技术的源代码进行分析，比如对Tomcat、Jetty、RPC框架的源代码去进行追踪分析

通过代码分析找到故障发生的原因，如果可能的话，最好是在本地搭建一个类似的环境，把线上问题给复现出来。

我们这里当时就结合日志里的异常栈分析了一下自己写的RPC框架的源代码，先给大家简单说一下，这个框架在接收请求的时候的一个流程。

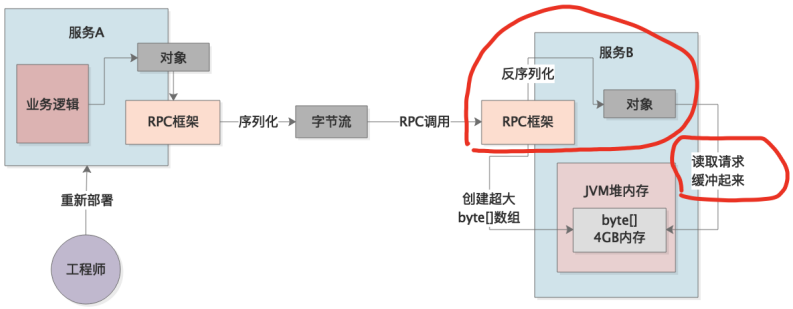
首先在服务A发送请求的时候，会对你传输过来的对象进行序列化。这个序列化，简单来说，就是把你的类似Request的对象变成一个byte[]数组，大概可以理解为这个意思，我们在下图里给大家反映出来。



然后对服务B而言，他首先会根据我们自定义的序列化协议（当时用的是Protobuf，很多人可能不了解这个，先别管这个了），对发送过来的数据进行反序列化

接着把请求数据读取到一个byte[]缓存中去，然后调用业务逻辑代码处理请求，最后请求处理完毕，清理byte[]缓存。

我们也在下面的图中反映出来服务B的处理流程。



想必大家都已经看明白上面RPC框架运行的原理了，接着我们自然在源码中要寻找一下，为什么用来缓冲请求的byte[]数组会搞成几个GB那么大？正常情况下，这个数组应该最多不超过1MB的。

7、铺垫一个关键知识点：RPC框架的类定义

通过源码的分析，我们最终总算搞清楚了，原来当时有这么一个特殊的情况，因为RPC框架要进行对象传输，就必须得让服务A和服务B都知道有这么一个对象

给大家举个例子，比如服务A要把一个Request对象传输给服务B，那么首先需要使用一种特定的语法定义一个对象文件，当时用的是ProtoBuf支持的语法，大家不理解的可以直接忽略，直接看看大概类似下面这样的一个文件就可以了：

```
syntax = "proto3";

message Request {
  string query = 1;
  int32 page_number = 2;
  int32 result_per_page = 3;
}
```

然后会通过上面那个特殊语法写的文件反向生成一个对应的Java类出来，此时会生成一个Java语法的Request类，类似下面这样：

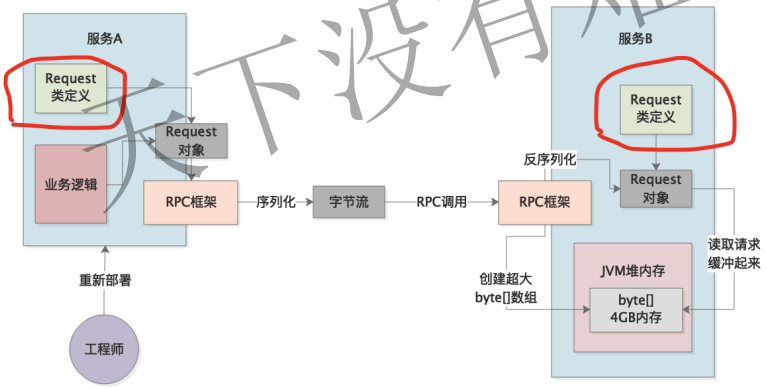
```
public class Requst {

// 这里是一大堆的自动生成的代码

}
```

接着这个Request类你需要在服务A和服务B的工程里都要引入，他们俩就知道，把Request交给服务A，他会自己进行序列化成字节流，然后到服务B的时候，他会把字节流反序列化成一个Request对象。

我们看下面的图，里面就引入了Request类的概念。



服务A和服务B都必须知道有Request类的存在，然后才能把Request对象序列化成字节流，也才能从字节流反序列化出来一个Request类的对象。

这是继续分析案例之前，大家务必清楚理解的一个概念。

8、RPC框架的一个bug：过大的默认值！

明白了这个，咱们就可以来继续看了，在上面的图中，服务B在接收到请求之后，会先反序列化，接着把请求读出来放入一个byte[]数组。

但是这里RPC框架我们有一个bug，就是一旦发现对方发送过来的字节流反序列化的时候失败了，这个往往是因为服务A对Request类做了修改，但是服务B不知道这次修改，Request还是以前的版本。

结果比如服务A的Request类有15个字段，序列化成字节流给你发送过来了，服务B的Request类只有10个字段，有的字段名字还不一样，那么反序列化的时候就会失败。

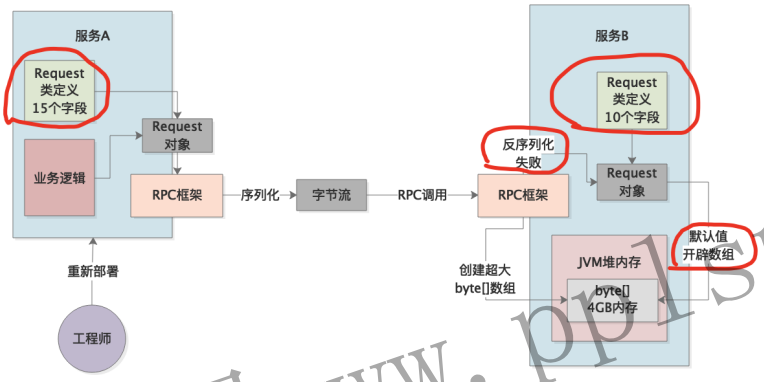
当时代码中写的逻辑是，一旦反序列化失败了，此时就会开辟一个byte[]数组，默认大小是4GB，然后把对方的字节流原封不动的放进去。

所以最终的问题就出在这里了，当时服务A的工程师修改了很多Request类的字段，结果没告诉服务B的工程师。

所以服务A上线之后，序列化的Request对象到服务B那里是没办法反序列化的，此时服务B就会直接开辟一个默认4GB的byte[]数组。

然后就会直接导致OOM了。

我们在下面的图中，把两边Request字段不一致的情况，反序列化失败的情况，以及开辟大数组的情况，都反应出来了，大家结合图可以看一下。



9、最终的解决方案

肯定很多人会疑惑，当时那个工程师为什么要把异常情况下的数组默认大小设置为几个GB那么大？

这个其实也没办法，因为当时写这段代码的刚好是才毕业的应届工程师，当时他考虑的是万一反序列化失败了，那么就原封不动的封装字节流到数组里去，让我们来自行处理。

但是他又不知道对方字节流里数据到底有多少，所以就干脆开辟一个特别大的数组，保证一定能放下字节流就可以了。

而且一般测试的时候都不会测到这种异常情况，所以之前一直没发现这个问题。

其实解决这个问题的办法很简单，把RPC框架中那个数组的默认值从4GB调整为4MB即可，一般请求都不会超过4MB，不需要开辟那么大的数组。

另外就是让服务A和服务B的Request类定义保持一致即可。上述问题就解决了。

End
