小心踩雷，一次Java内存泄漏排查实战

程序员私房菜 1周前

> 前些日子小组内安排值班，轮流看顾我们的服务，主要做一些报警邮件处理、Bug 排查、运营 issue 处理的事。工作日还好，无论干什么都要上班的，若是轮到周末，那这一天算是毁了。

不知道是公司网络广了就这样还是网络运维组不给力，网络总有问题，不是这边交换机脱网了，就是那边路由器坏了，还偶发地各种超时，而我们灵敏的服务探测服务总能准确地抓住偶现的小问题，给美好的工作加点料。

好几次值班组的小伙伴们一起吐槽，商量着怎么避过服务保活机制，偷偷停了探测服务而不让人发现（虽然也并不敢）。前些天我就在周末处理了一次探测服务的锅。

问题出现

问题出现

晚上七点多开始，我就开始不停地收到报警邮件，邮件显示探测的几个接口有超时情况。

多数执行栈都在：

```
java.io.BufferedReader.readLine(BufferedReader.java:371)
java.io.BufferedReader.readLine(BufferReader.java:389)
java_io_BufferedReader$readLine.call(Unknown Source)
com.domain.detect.http.HttpClient.getResponse(HttpClient.groovy:122)
com.domain.detect.http.HttpClient.this$2$getResponse(HttpClient.groovy)
```

这个线程栈的报错我见得多了，我们设置的 HTTP DNS 超时是 1s，connect 超时是 2s，read 超时是 3s。

这种报错都是探测服务正常发送了 HTTP 请求，服务器也在收到请求正常处理后正常响应了，但数据包在网络层层转发中丢失了，所以请求线程的执行栈会停留在获取接口响应的地方。

这种情况的典型特征就是能在服务器上查找到对应的日志记录。而且日志会显示服务器响应完全正常。

与它相对的还有线程栈停留在 Socket connect 处的，这是在建连时就失败了，服务端完全无感知。

我注意到其中一个接口报错更频繁一些，这个接口需要上传一个 4M 的文件到服务器，然后经过一连串的业务逻辑处理，再返回 2M 的文本数据。

而其他的接口则是简单的业务逻辑，我猜测可能是需要上传下载的数据太多，所以超时导致丢包的概率也更大吧。

根据这个猜想，群登上服务器，使用请求的 request_id 在近期服务日志中搜索一下，果不其然，就是网络丢包问题导致的接口超时了。

当然这样 leader 是不会满意的，这个结论还得有人接锅才行。于是赶紧联系运维和网络组，向他们确认一下当时的网络状态。

网络组同学回复说是我们探测服务所在机房的交换机老旧，存在未知的转发瓶颈，正在优化，这让我更放心了，于是在部门群里简单交待一下，算是完成任务。

问题爆发

本以为这次值班就起这么一个小波浪，结果在晚上八点多，各种接口的报警邮件蜂拥而至，打得准备收拾东西过周日单休的我措手不及。

这次几乎所有的接口都在超时，而我们那个大量网络 I/O 的接口则是每次探测必超时，难道是整个机房故障了么？

我再次通过服务器和监控看到各个接口的指标都很正常，自己测试了下接口也完全 OK，既然不影响线上服务，我准备先通过探测服务的接口把探测任务停掉再慢慢排查。

结果给暂停探测任务的接口发请求好久也没有响应，这时候我才知道没这么简单。

解决问题

内存泄漏

于是赶快登陆探测服务器，首先是 top free df 三连，结果还真发现了些异常：

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
17737 root      20   0 11.401g 3.574g  12148 S 887.5 30.9   3984:59 java
    1 root      20   0   56768   3528   1708 S   0.0  0.0  13:57.08 systemd
```

我们的探测进程 CPU 占用率特别高，达到了 900%。

我们的 Java 进程，并不做大量 CPU 运算，正常情况下，CPU 应该在 100~200% 之间，出现这种 CPU 飙升的情况，要么走到了死循环，要么就是在做大量的 GC。

使用 jstat -gc pid [interval] 命令查看了 Java 进程的 GC 状态，果然，FULL GC 达到了每秒一次：

```
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC     MU    CCSC   CCSU   YGC     YGCT    FGC    FGCT     GCT
16896.0 28160.0  0.0    0.0   650240.0 650240.0 2022400.0  2022036.4  90368.0 83233.5 11264.0 8943.1  13337  205.910 15718 22584.871 22790.781
16896.0 28160.0  0.0    0.0   650240.0 650240.0 2022400.0  2022036.4  90368.0 83233.5 11264.0 8943.1  13337  205.910 15719 22586.145 22792.055
16896.0 28160.0  0.0    0.0   650240.0 650240.0 2022400.0  2022036.4  90368.0 83233.5 11264.0 8943.1  13337  205.910 15720 22587.434 22793.344
16896.0 28160.0  0.0    0.0   650240.0 650240.0 2022400.0  2022036.4  90368.0 83233.5 11264.0 8943.1  13337  205.910 15721 22588.696 22794.606
16896.0 28160.0  0.0    0.0   650240.0 650240.0 2022400.0  2022036.4  90368.0 83233.5 11264.0 8943.1  13337  205.910 15722 22589.882 22795.792
```

这么多的 FULL GC，应该是内存泄漏没跑了，于是使用 jstack pid > jstack.log 保存了线程栈的现场。

使用 jmap -dump:format=b,file=heap.log pid 保存了堆现场，然后重启了探测服务，报警邮件终于停止了。

jstat

jstat 是一个非常强大的 JVM 监控工具，一般用法是：

```
jstat [-options] pid interval
```

它支持的查看项有：

- **class 查看类加载信息。**
- **compile 编译统计信息。**
- **gc 垃圾回收信息。**
- **gcXXX 各区域 GC 的详细信息，如 -gcold。**

使用它，对定位 JVM 的内存问题很有帮助。

排查问题

问题虽然解决了，但为了防止它再次发生，还是要把根源揪出来。

分析栈

栈的分析很简单，看一下线程数是不是过多，多数栈都在干嘛：

```
> grep 'java.lang.Thread.State' jstack.log  | wc -l
> 464
```

才四百多线程，并无异常：

```
> grep -A 1 'java.lang.Thread.State' jstack.log  | grep -v 'java.lang.Thread.State' | s

     10     at java.lang.Class.forName0(Native Method)
     10     at java.lang.Object.wait(Native Method)
     16     at java.lang.ClassLoader.loadClass(ClassLoader.java:404)
     44     at sun.nio.ch.EPollArrayWrapper.epollWait(Native Method)
    344     at sun.misc.Unsafe.park(Native Method)
```

线程状态好像也无异常，接下来分析堆文件。

下载堆 dump 文件

堆文件都是一些二进制数据，在命令行查看非常麻烦，Java 为我们提供的工具都是可视化的，Linux 服务器上又没法查看，那么首先要把文件下载到本地。

由于我们设置的堆内存为 4G，所以 dump 出来的堆文件也很大，下载它确实非常费事，不过我们可以先对它进行一次压缩。

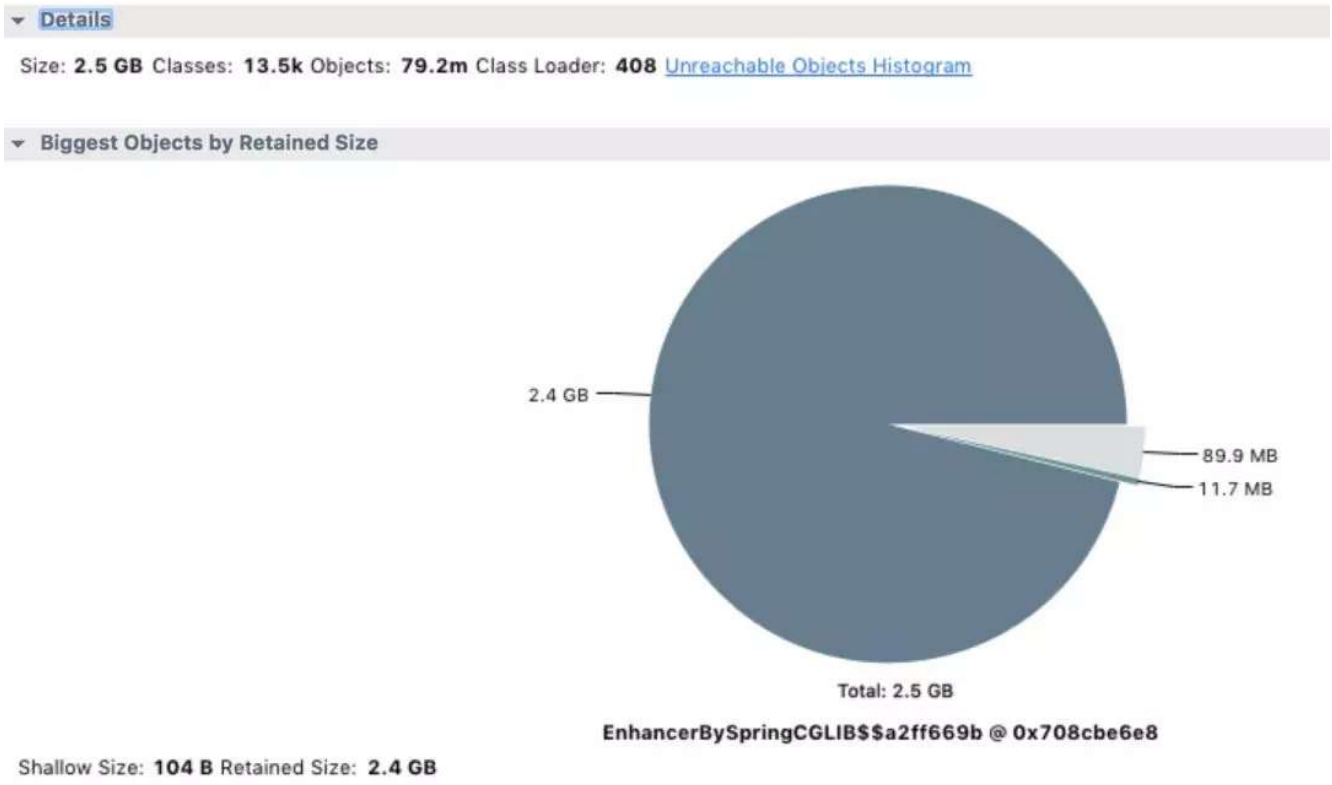
gzip 是个功能很强大的压缩命令，特别是我们可以设置 -1~-9 来指定它的压缩级别。

数据越大压缩比率越大，耗时也就越长，推荐使用 -6~7，-9 实在是太慢了，且收益不大，有这个压缩的时间，多出来的文件也下载好了。

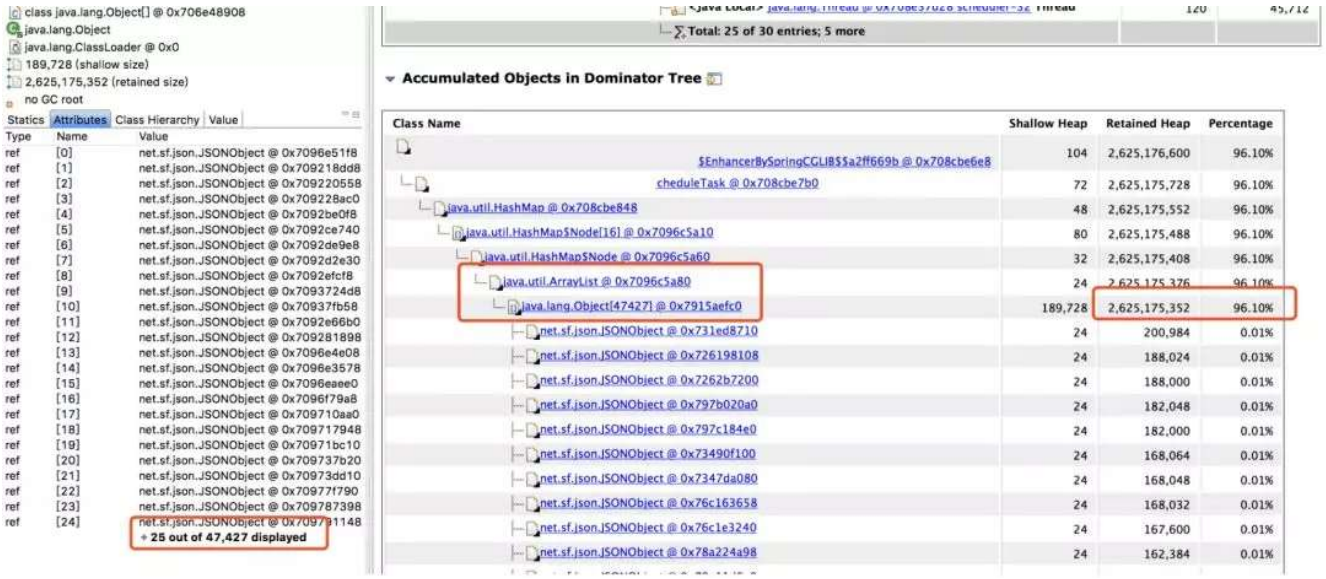
使用 MAT 分析 jvm heap

MAT 是分析 Java 堆内存的利器，使用它打开我们的堆文件（将文件后缀改为 .hprof），它会提示我们要分析的种类。

对于这次分析，果断选择 memory leak suspect：



从上面的饼图中可以看出，绝大多数堆内存都被同一个内存占用了，再查看堆内存详情，向上层追溯，很快就发现了罪魁祸首。



分析代码

找到内存泄漏的对象了，在项目里全局搜索对象名，它是一个 Bean 对象，然后定位到它的一个类型为 Map 的属性。

这个 Map 根据类型用 ArrayList 存储了每次探测接口响应的结果，每次探测完都塞到 ArrayList 里去分析。

由于 Bean 对象不会被回收，这个属性又没有清除逻辑，所以在服务十来天没有上线重启的情况下，这个 Map 越来越大，直至将内存占满。

内存满了之后，无法再给 HTTP 响应结果分配内存了，所以一直卡在 readLine 那里。而我们那个大量 I/O 的接口报警次数特别多，估计跟响应太大需要更多内存有关。

给代码 owner 提了 PR，问题圆满解决。

小结

其实还是要反省一下自己的，一开始报警邮件里还有这样的线程栈：

```
groovy.json.internal.JsonParserCharArray.decodeValueInternal(JsonParserCharArray.java:1
groovy.json.internal.JsonParserCharArray.decodeJsonObject(JsonParserCharArray.java:132)
groovy.json.internal.JsonParserCharArray.decodeValueInternal(JsonParserCharArray.java:1
groovy.json.internal.JsonParserCharArray.decodeJsonObject(JsonParserCharArray.java:132)
groovy.json.internal.JsonParserCharArray.decodeValueInternal(JsonParserCharArray.java:1
```

看到这种报错线程栈却没有细想，要知道 TCP 是能保证消息完整性的，况且消息没有接收完也不会把值赋给变量。

这种很明显的是内部错误，如果留意后细查是能提前查出问题所在的，查问题真是差了哪一环都不行啊。

作者：枕边书

编辑：陶家龙、孙淑娟

出处：https://zhenbianshu.github.io

往 期 推 荐

阿里面试，我挂在了第四轮.....

程序员最讨厌的9句话，你可有补充？

@程序员，你已掉入能力陷阱！

同样是程序员，为什么别人比你更优秀？

我是如何从通信转到Java软件开发工程师的？

程序员面试IT公司，这些地方你要注意！

有了这些习惯，你离成功会越来越近！

关注我

每天进步一点点

点赞是最大的支持

绝影

堆内存满了后无法给http请求结果分配内存

如果一个没有大量io操作也没有大量需要Cpu计算的接口请求后响应非常慢甚至超时，数据库连接数不大正常时候这个接口的sql执行很快，那么这种情况是指我的请求结果从数据库返回来后由于虚拟机堆内存不足，无法及时给http请求结果分配内存就会导致响应很慢嘛

文耀

跟前段时间踩的Jboss的雷一样