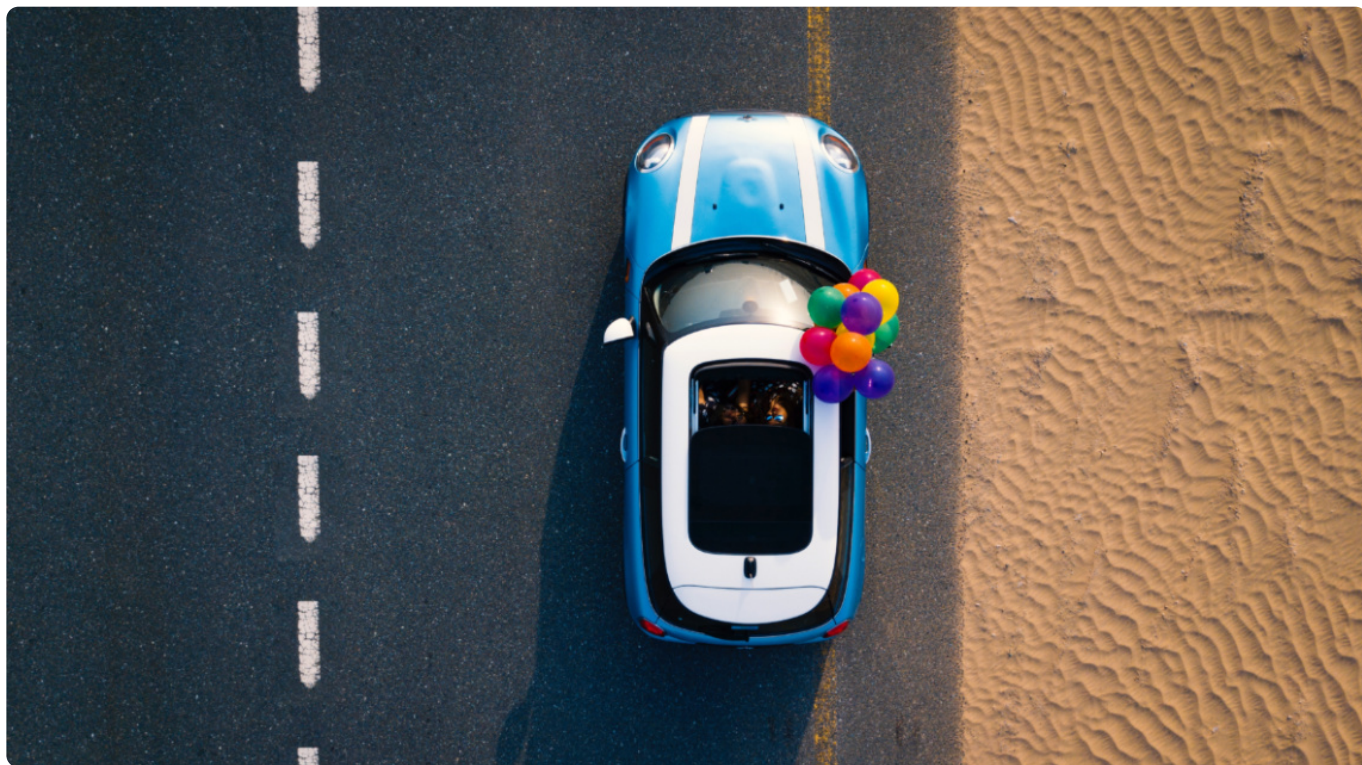


37 | Tomcat内存溢出的原因分析及调优

2019-08-06 李号双

深入拆解Tomcat & Jetty

[进入课程 >](#)



讲述：李号双

时长 10:26 大小 9.57M



作为 Java 程序员，我们几乎都会碰到 `java.lang.OutOfMemoryError` 异常，但是你知道有哪些原因可能导致 JVM 抛出 `OutOfMemoryError` 异常吗？

JVM 在抛出 `java.lang.OutOfMemoryError` 时，除了会打印出一行描述信息，还会打印堆栈跟踪，因此我们可以通过这些信息来找到导致异常的原因。在寻找原因前，我们先来看看有哪些因素会导致 `OutOfMemoryError`，其中内存泄漏是导致 `OutOfMemoryError` 的一个比较常见的原因，最后我们通过一个实战案例来定位内存泄漏。

内存溢出场景及方案

`java.lang.OutOfMemoryError: Java heap space`

JVM 无法在堆中分配对象时，会抛出这个异常，导致这个异常的原因可能有三种：

1. 内存泄漏。Java 应用程序一直持有 Java 对象的引用，导致对象无法被 GC 回收，比如对象池和内存池中的对象无法被 GC 回收。
2. 配置问题。有可能是我们通过 JVM 参数指定的堆大小（或者未指定的默认大小），对于应用程序来说是不够的。解决办法是通过 JVM 参数加大堆的大小。
3. finalize 方法的过度使用。如果我们想在 Java 类实例被 GC 之前执行一些逻辑，比如清理对象持有的资源，可以在 Java 类中定义 finalize 方法，这样 JVM GC 不会立即回收这些对象实例，而是将对象实例添加到一个叫“java.lang.ref.Finalizer.ReferenceQueue”的队列中，执行对象的 finalize 方法，之后才会回收这些对象。Finalizer 线程会和主线程竞争 CPU 资源，但由于优先级低，所以处理速度跟不上主线程创建对象的速度，因此 ReferenceQueue 队列中的对象就越来越多，最终会抛出 OutOfMemoryError。解决办法是尽量不要给 Java 类定义 finalize 方法。

java.lang.OutOfMemoryError: GC overhead limit exceeded

出现这种 OutOfMemoryError 的原因是，垃圾收集器一直在运行，但是 GC 效率很低，比如 Java 进程花费超过 98% 的 CPU 时间来进行一次 GC，但是回收的内存少于 2% 的 JVM 堆，并且连续 5 次 GC 都是这种情况，就会抛出 OutOfMemoryError。

解决办法是查看 GC 日志或者生成 Heap Dump，确认一下是不是内存泄漏，如果不是内存泄漏可以考虑增加 Java 堆的大小。当然你还可以通过参数配置来告诉 JVM 无论如何也不要抛出这个异常，方法是配置 `-XX:-UseGCOverheadLimit`，但是我并不推荐这么做，因为这只是延迟了 OutOfMemoryError 的出现。

java.lang.OutOfMemoryError: Requested array size exceeds VM limit

从错误消息我们也能猜到，抛出这种异常的原因是“请求的数组大小超过 JVM 限制”，应用程序尝试分配一个超大的数组。比如应用程序尝试分配 512MB 的数组，但最大堆大小为 256MB，则将抛出 OutOfMemoryError，并且请求的数组大小超过 VM 限制。

通常这这也是一个配置问题（JVM 堆太小），或者是应用程序的一个 Bug，比如程序错误地计算了数组的大小，导致尝试创建一个大小为 1GB 的数组。

java.lang.OutOfMemoryError: MetaSpace

如果 JVM 的元空间用尽，则会抛出这个异常。我们知道 JVM 元空间的内存存在本地内存中分配，但是它的大小受参数 `MaxMetaSpaceSize` 的限制。当元空间大小超过 `MaxMetaSpaceSize` 时，JVM 将抛出带有 `MetaSpace` 字样的 `OutOfMemoryError`。解决办法是加大 `MaxMetaSpaceSize` 参数的值。

java.lang.OutOfMemoryError: Request size bytes for reason. Out of swap space

当本地堆内存分配失败或者本地内存快要耗尽时，Java HotSpot VM 代码会抛出这个异常，VM 会触发“致命错误处理机制”，它会生成“致命错误”日志文件，其中包含崩溃时线程、进程和操作系统的有用信息。如果碰到此类型的 `OutOfMemoryError`，你可以根据 JVM 抛出的错误信息来进行诊断；或者使用操作系统提供的 DTrace 工具来跟踪系统调用，看看是什么样的程序代码在不断地分配本地内存。

java.lang.OutOfMemoryError: Unable to create native threads

抛出这个异常的过程大概是这样的：

1. Java 程序向 JVM 请求创建一个新的 Java 线程。
2. JVM 本地代码（Native Code）代理该请求，通过调用操作系统 API 去创建一个操作系统级别的线程 Native Thread。
3. 操作系统尝试创建一个新的 Native Thread，需要同时分配一些内存给该线程，每一个 Native Thread 都有一个线程栈，线程栈的大小由 JVM 参数 `-Xss` 决定。
4. 由于各种原因，操作系统创建新的线程可能会失败，下面会详细谈到。
5. JVM 抛出“`java.lang.OutOfMemoryError: Unable to create new native thread`”错误。

因此关键在于第四步线程创建失败，JVM 就会抛出 `OutOfMemoryError`，那具体有哪些因素会导致线程创建失败呢？

1. 内存大小限制：我前面提到，Java 创建一个线程需要消耗一定的栈空间，并通过 `-Xss` 参数指定。请你注意的是栈空间如果过小，可能会导致 `StackOverflowError`，尤其是在递归调用的情况下；但是栈空间过大会占用过多内存，而对于一个 32 位 Java 应用来说，用户进程空间是 4GB，内核占用 1GB，那么用户空间就剩下 3GB，因此它能创建的线程数大致可以通过这个公式算出来：

```
1 Max memory (3GB) = [-Xmx] + [-XX:MaxMetaSpaceSize] + number_of_threads * [-Xss]
```

不过对于 64 位的应用，由于虚拟进程空间近乎无限大，因此不会因为线程栈过大而耗尽虚拟地址空间。但是请你注意，64 位的 Java 进程能分配的最大内存数仍然受物理内存大小的限制。

2.ulimit 限制，在 Linux 下执行 `ulimit -a`，你会看到 `ulimit` 对各种资源的限制。

```
#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) 31876
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) 8192 线程栈大小
cpu time                (seconds, -t) unlimited
max user processes      (-u) 1024 线程数
virtual memory          (kbytes, -v) unlimited
file locks              (-x) unlimited
```

其中的 “max user processes” 就是一个进程能创建的最大线程数，我们可以修改这个参数：

```
#ulimit -u 65535
```

3.参数`sys.kernel.threads-max`限制。这个参数限制操作系统全局的线程数，通过下面的命令可以查看它的值。

```
#cat /proc/sys/kernel/threads-max
63752
```

这表明当前系统能创建的总的线程是 63752。当然我们调整这个参数，具体办法是：

在`/etc/sysctl.conf`配置文件中，加入`sys.kernel.threads-max = 999999`。


4.参数`sys.kernel.pid_max`限制，这个参数表示系统全局的 PID 号数值的限制，每一个线程都有 ID，ID 的值超过这个数，线程就会创建失败。跟`sys.kernel.threads-max`参数一样，我们也可以将`sys.kernel.pid_max`调大，方法是在`/etc/sysctl.conf`配置文件中，加入`sys.kernel.pid_max = 999999`。

对于线程创建失败的 `OutOfMemoryError`，除了调整各种参数，我们还需要从程序本身找找原因，看看是否真的需要这么多线程，有可能是程序的 Bug 导致创建过多的线程。

内存泄漏定位实战

我们先创建一个 Web 应用，不断地 new 新对象放到一个 List 中，来模拟 Web 应用中的内存泄漏。然后通过各种工具来观察 GC 的行为，最后通过生成 Heap Dump 来找到泄漏点。

内存泄漏模拟程序比较简单，创建一个 Spring Boot 应用，定义如下所示的类：

 复制代码

```
1 import org.springframework.scheduling.annotation.Scheduled;
2 import org.springframework.stereotype.Component;
3
4 import java.util.LinkedList;
5 import java.util.List;
6
7 @Component
8 public class MemLeaker {
9
10     private List<Object> objs = new LinkedList<>();
```


```

11
12     @Scheduled(fixedRate = 1000)
13     public void run() {
14
15         for (int i = 0; i < 50000; i++) {
16             objs.add(new Object());
17         }
18     }
19 }

```


这个程序做的事情就是每隔 1 秒向一个 List 中添加 50000 个对象。接下来运行并通过工具观察它的 GC 行为：

1. 运行程序并打开 verbosegc，将 GC 的日志输出到 gc.log 文件中。

 复制代码

```
1 java -verbose:gc -Xloggc:gc.log -XX:+PrintGCDetails -jar mem-0.0.1-SNAPSHOT.jar
```

2. 使用 jstat 命令观察 GC 的过程：

 复制代码

```
1 jstat -gc 94223 2000 1000
```

94223 是程序的进程 ID，2000 表示每隔 2 秒执行一次，1000 表示持续执行 1000 次。下面是命令的输出：

S0C	S1C	S0U	S1U	EC	EU	OC	OU	MC	MU	CCSC	CCSU	YGC	YGCT	FGC	FGCT	GCT
10752.0	10752.0	10736.0	0.0	65536.0	10064.0	175104.0	11480.0	18304.0	17418.8	2432.0	2216.1	2	0.075	0	0.000	0.075
10752.0	10752.0	10736.0	0.0	65536.0	14089.6	175104.0	11480.0	18304.0	17418.8	2432.0	2216.1	2	0.075	0	0.000	0.075
10752.0	10752.0	10736.0	0.0	65536.0	18115.2	175104.0	11480.0	18304.0	17418.8	2432.0	2216.1	2	0.075	0	0.000	0.075

其中每一列的含义是：

S0C：第一个 Survivor 区总的大小；

S1C：第二个 Survivor 区总的大小；

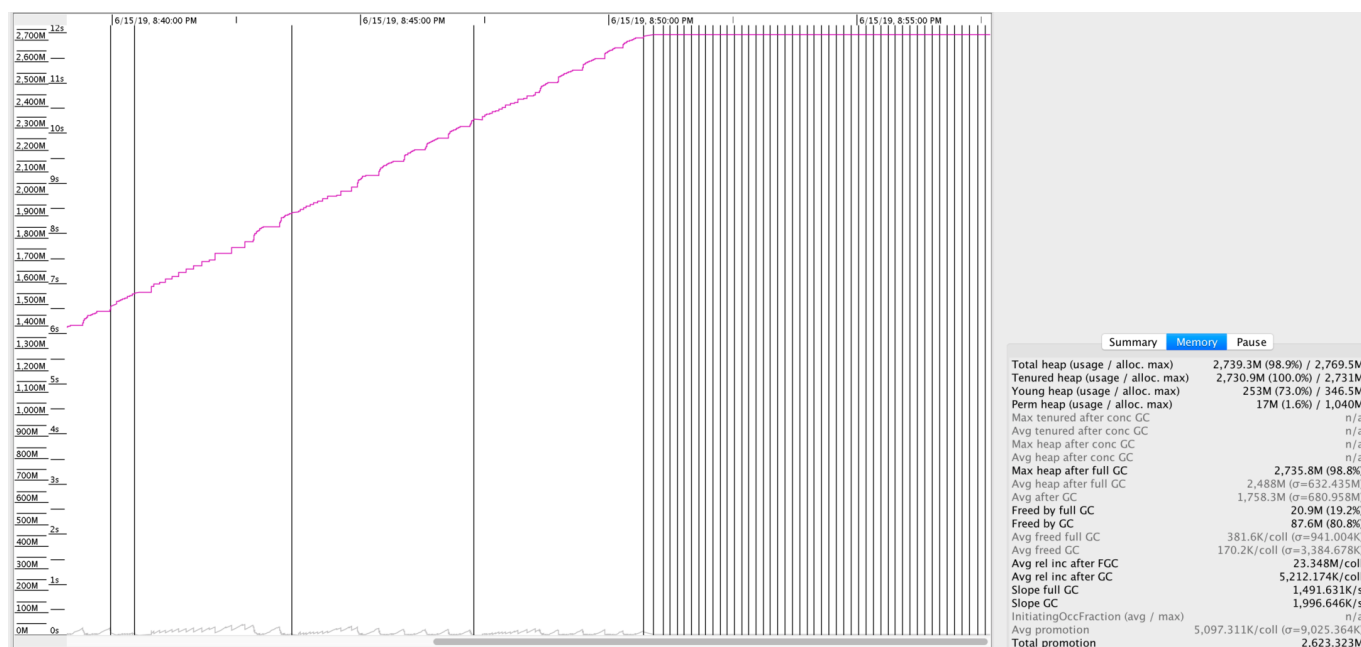
S0U：第一个 Survivor 区已使用内存的大小；

S1U：第二个 Survivor 区已使用内存的大小。

后面的列相信从名字你也能猜出是什么意思了，其中 E 代表 Eden，O 代表 Old，M 代表 Metadata；YGC 表示 Minor GC 的总时间，YGCT 表示 Minor GC 的次数；FGC 表示 Full GC。

通过这个工具，你能大概看到各个内存区域的大小、已经 GC 的次数和所花的时间。
verbosegc 参数对程序的影响比较小，因此很适合在生产环境现场使用。

3. 通过 GCViewer 工具查看 GC 日志，用 GCViewer 打开第一步产生的 gc.log，会看到这样的图：



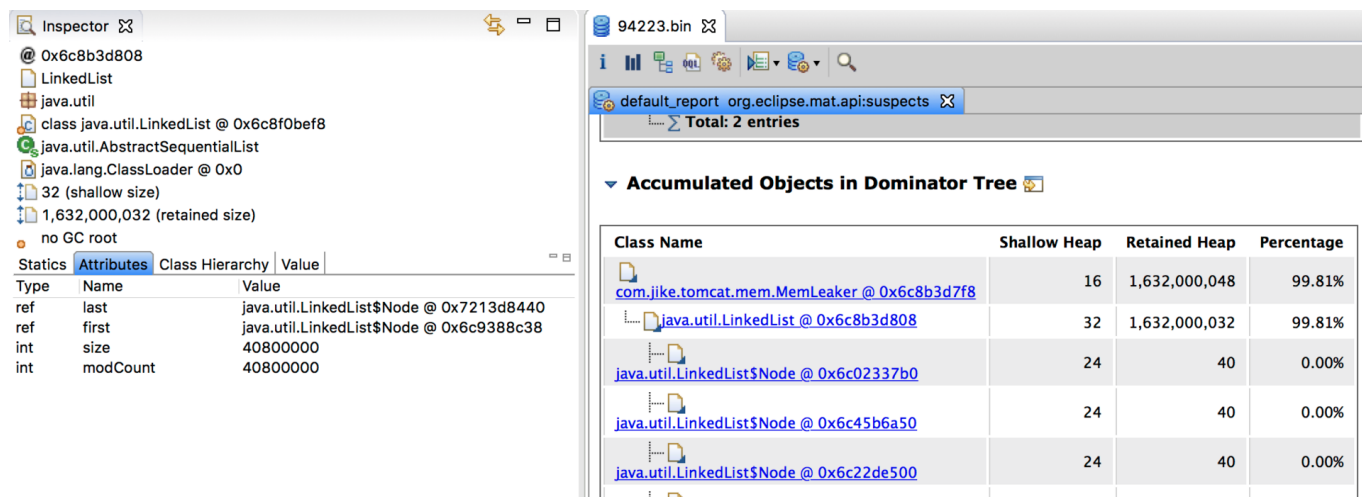
图中红色的线表示年老代占用的内存，你会看到它一直在增加，而黑色的竖线表示一次 Full GC。你可以看到后期 JVM 在频繁地 Full GC，但是年老代的内存并没有降下来，这是典型的内存泄漏的特征。

除了内存泄漏，我们还可以通过 GCViewer 来观察 Minor GC 和 Full GC 的频次，已及每次的内存回收量。

4. 为了找到内存泄漏点，我们通过 jmap 工具生成 Heap Dump：

```
1 jmap -dump:live,format=b,file=94223.bin 94223
```

5. 用 Eclipse Memory Analyzer 打开 Dump 文件，通过内存泄漏分析，得到这样一个分析报告：



从报告中可以看到，JVM 内存中有一个长度为 4000 万的 List，至此我们也就找到了泄漏点。

本期精华

今天我讲解了常见的 `OutOfMemoryError` 的场景以及解决办法，我们在实际工作中要根据具体的错误信息去分析背后的原因，尤其是 Java 堆内存不够时，需要生成 Heap Dump 来分析，看是不是内存泄漏；排除内存泄漏之后，我们再调整各种 JVM 参数，否则根本的问题原因没有解决的话，调整 JVM 参数也无济于事。

课后思考

请你分享一下平时在工作中遇到了什么样的 `OutOfMemoryError`，以及你是怎么解决的。

不知道今天的内容你消化得如何？如果还有疑问，请大胆的在留言区提问，也欢迎你把你的课后思考和心得记录下来，与我和其他同学一起讨论。如果你觉得今天有所收获，欢迎你把它分享给你的朋友。

深入拆解 Tomcat & Jetty

从源码角度深度探索 Java 中间件

李号双

eBay 技术主管



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 36 | Tomcat I/O和线程池的并发调优

下一篇 38 | Tomcat拒绝连接原因分析及网络优化

精选留言 (4)

写留言



QQ怪

2019-08-06

之前在使用es的时候想用线程池来优化频繁获取连接造成的资源浪费，但因为自己粗心，使用的过程中错误的操作获取连接都去new线程池，而不是从线程池获取线程，导致内存老是到顶，那次内存泄露还是自己的基本功不扎实导致的，最后也是通过一些jvm工具找到了问题，当时画了不少时间在上面，挺感慨的

展开 ∨



2



neohope

2019-08-06

YGC 表示 Minor GC 的总时间，YGCT 表示 Minor GC 的次数。这两个是写反了吗？



1



广训

2019-08-07

刚入职现在的公司，发现线上某个实例会不定期不提供服务，进程还在，但是不再接受请求。每次都重启就恢复，后来一直观察。线上没写权限，也没什么监控工具，就是那种突然出问题，还要临时申请写权限去机器上执行jvm的相关命令。有时候也抱怨权限这东西，就跟站着茅坑不拉屎一样搞笑，有权限的人不做，要做的人做不了。有次用eclipse的工具看了下，dubbo对象特别多，都超过spring boot的了，看了dubbo的代码，发觉居然是...
展开 ∨



酱油君

2019-08-06

赞 自己手还是比较生 了解到的知识面比较窄 还是要多记 多理解 多联系 无奈平时的增删改查太多了

