

图文 085、动手实验：JVM堆内存溢出的时候，应该如何解决？

506 人次阅读 2019-09-30 07:00:00

详情 评论

动手实验：
JVM堆内存溢出的时候，应该如何解决？

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：



相关频道



从 0 开
战高手
已更新1

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)



狸猫技术窝

进店逛

1、前文回顾

上一篇文章已经给大家分析了栈内存溢出是如何来解决的，这篇文章我们给大家分析一下最常见的堆内存溢出是如何来解决的。

2、示例代码

我们还是沿用之前的示例代码：

```
public class Demo3 {  
  
    public static void main(String[] args) {  
        long counter = 0;  
        List<Object> list = new ArrayList<Object>();  
        while(true) {  
            list.add(new Object());  
            System.out.println("当前创建了第" + (++counter) + "个对象");  
        }  
    }  
}
```

采用的JVM参数如下：

```
-Xms10m  
-Xmx10m  
-XX:+PrintGCDetails  
-Xloggc:gc.log  
-XX:+HeapDumpOnOutOfMemoryError  
-XX:HeapDumpPath=/  
-XX:+UseParNewGC  
-XX:+UseConcMarkSweepGC
```

接着我们运行上述程序。

3、运行后的观察

其实堆内存溢出现象也是很简单的，在系统运行一段时间之后，直接会发现系统崩溃了，然后登录到线上机器检查日志文件

先看到底为什么崩溃：

```
java.lang.OutOfMemoryError: Java heap space  
Dumping heap to ./java_pid1023.hprof ...  
Heap dump file created [13409210 bytes in 0.033 secs]
```

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
```

这个就很明显告诉我们，是Java堆内存溢出了，而且他还给我们导出了一份内存快照。

所以此时我们GC日志都不用分析了，因为堆内存溢出往往对应着大量的GC日志，所以你分析起来是很麻烦的。

此时直接将线上自动导出的内存快照拷贝回本地笔记本电脑，然后用MAT分析即可。

4、用MAT分析内存快照

采用MAT打开内存快照之后会看到下图：

▼ **Problem Suspect 1**

The thread **java.lang.Thread @ 0x7bf6a9a98 main** keeps local variables with total size **7,203,536 (92.03%)** bytes.

The memory is accumulated in one instance of "**java.lang.Object[]**" loaded by "**<system class loader>**".

The stacktrace of this Thread is available. [See stacktrace.](#)

Keywords
java.lang.Object[]
[Details >](#)

这次MAT非常简单，直接在内存泄漏报告中告诉我们，内存溢出原因只有一个！那就是这个问题，因为他没提示任何其他的问题。

我们这次来给大家仔细分析一下MAT给我们的分析报告。

首先看下面的句子：The thread java.lang.Thread @ 0x7bf6a9a98 main keeps local variables with total size 7,203,536 (92.03%) bytes。











这个意思就是main线程通过局部变量引用了7230536个字节的对象，大概就是7MB左右。

考虑到我们总共就给堆内存10MB，所以7MB基本上个已经到极限了，是差不多的。

我们接着看：The memory is accumulated in one instance of "java.lang.Object[]" loaded by "<system class loader>"。

这句话的意思就是内存都被一个实例对象占用了，就是java.lang.Object[]。

我们肯定不知道这个是什么东西，所以得往下看，点击**Details**

Class Name	Shallow Heap	Retained Heap	Percentage
 java.lang.Thread @ 0x7bf6a9a98 main	120	7,203,536	92.03%
 java.lang.Object[360145] @ 0x7bf9f3d58	1,440,600	7,202,920	92.02%
 java.lang.Object @ 0x7bf606928	16	16	0.00%
 java.lang.Object @ 0x7bf606938	16	16	0.00%
 java.lang.Object @ 0x7bf606948	16	16	0.00%
 java.lang.Object @ 0x7bf606958	16	16	0.00%
 java.lang.Object @ 0x7bf606968	16	16	0.00%
 java.lang.Object @ 0x7bf606978	16	16	0.00%
 java.lang.Object @ 0x7bf606988	16	16	0.00%
 java.lang.Object @ 0x7bf606998	16	16	0.00%

在Details里我们能看到这个东西，也就是占用了7MB内存的的java.lang.Object[]，他里面的每个元素在这里都有，我们看到是一大堆的java.lang.Object。那么这些java.lang.Object不就是我们在代码里创建的吗？

至此真相大白，我们已经知道，就是一大堆的Object对象占用了7MB的内存导致了内存溢出。

接着下一个任务就是知道这些对象是怎么创建出来的，那么我们怎么找呢？回到之前的上一级页面，各位看下图。

▼ Problem Suspect 1

The thread **java.lang.Thread @ 0x7bf6a9a98 main** keeps local variables with total size **7,203,536 (92.03%)** bytes.

The memory is accumulated in one instance of "**java.lang.Object[]**" loaded by "**<system class loader>**".

The stacktrace of this Thread is available. [See stacktrace.](#)

Keywords

java.lang.Object[]

[Details »](#)

这个就是说可以看看创建那么多对象的线程，他的一个执行栈，这样我们就知道这个线程执行什么方法的时候创建了一大堆的对象。

[Leak Suspects](#) » [Leaks](#) » [Problem Suspect 1](#) » [Description](#) » [Thread Stack](#)

▼ Thread Stack

```
main
  at java.lang.OutOfMemoryError.<init>()V (OutOfMemoryError.java:48)
  at java.util.Arrays.copyOf([Ljava/lang/Object;ILjava/lang/Class;)[Ljava/lang/Object; (
  at java.util.Arrays.copyOf([Ljava/lang/Object;I)[Ljava/lang/Object; (Arrays.java:3181)
  at java.util.ArrayList.grow(I)V (ArrayList.java:265)
  at java.util.ArrayList.ensureExplicitCapacity(I)V (ArrayList.java:239)
  at java.util.ArrayList.ensureCapacityInternal(I)V (ArrayList.java:231)
  at java.util.ArrayList.add(Ljava/lang/Object;)Z (ArrayList.java:462)
  at com.limao.demo.jvm.Demo3.main([Ljava/lang/String;)V (Demo3.java:12)
```

大家看上面的调用栈，其实说的很明显了，在Demo3.main()方法中，一直在调用ArrayList.add()方法，然后此时直接引发了内存溢出。所以我们只要在对应该代码里看一下，立马就知道怎么回事了。

接下来优化对应的代码即可，就不会发生内存溢出了。

5、本文总结

今天的文章带着大家分析了一下，堆内存溢出的问题如何分析和定位

其实很简单，一个是必须在JVM参数中加入自动导出内存快照，一个是到线上看一下日志文件里的报错，如果是堆溢出，立马用MAT分析内存快照。

MAT分析的时候一些顺序和技巧，还有思路，也教给大家了，先看占用内存最多的对象是谁，然后分析那个线程的调用栈，接着就可以看到是哪个方法引发的内存溢出了。接着优化代码即可。

到这周为止，我们已经彻底学会了JVM OOM的原理以及现象，还有解决问题的思路和方法。

下周开始我们用最后两周给大家演示多个各种各样的奇怪的OOM问题，都是从我们之前经历过的大量真实案例，带着各种案例的业务背景给大家分析。

这样大家积累丰富的OOM处理经验后，就能彻底成为JVM优化实战的高手了。

End

专栏版权归公众号**狸猫技术窝**所有

未经许可不得传播，如有侵权将追究法律责任

如何加群?

添加微信号: Lvgu0715_ (微信名: 绿小九), 狸猫技术窝管理员

发送 Jvm专栏的购买截图

由于是人工操作, 发送截图后请耐心等待被拉群

最后再次提醒: 通过其他专栏加过群的同学, 就不要重复加了