面试官：你工作中做过 JVM 调优吗？怎么做的？

点击关注 面试专栏 2024-01-15 14:04 发表于广东

最近很多小伙伴跟我说，自己学了不少JVM的调优知识，但是在实际工作中却不知道何时对JVM进行调优。今天，我就为大家介绍几种JVM调优的场景。

> 在阅读本文时，假定大家已经了解了运行时的数据区域和常用的垃圾回收算法，也了解了Hotspot支持的垃圾回收器。

cpu占用过高

cpu占用过高要分情况讨论，是不是业务上在搞活动，突然有大批的流量进来，而且活动结束后cpu占用率就下降了，如果是这种情况其实可以不用太关心，因为请求越多，需要处理的线程数越多，这是正常的现象。话说回来，如果你的服务器配置本身就差，cpu也只有一个核心，这种情况，稍微多一点流量就真的能够把你的cpu资源耗尽，这时应该考虑先把配置提升吧。

第二种情况，cpu占用率**长期过高**，这种情况下可能是你的程序有那种循环次数超级多的代码，甚至是出现死循环了。排查步骤如下：

（1）用top命令查看cpu占用情况

```
top - 05:03:36 up  5:41,  4 users,  load average: 0.38, 0.12, 0.07
Tasks: 178 total,   1 running, 177 sleeping,   0 stopped,   0 zombie
%Cpu(s): 49.6 us,  0.2 sy,  0.0 ni, 50.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  2895296 total,  1681848 free,   743052 used,   470396 buff/cache
KiB Swap:  2097148 total,  2097148 free,        0 used.  1967216 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 7268 root      20   0 3289540 264704  13480 S 100.0  9.1   0:29.47 java
    9 root      20   0       0      0      0 S   0.3  0.0   0:33.45 rcu_sched
    1 root      20   0  128228   6920   4200 S   0.0  0.2   0:03.15 systemd
```

这样就可以定位出cpu过高的进程。在linux下，top命令获得的进程号和jps工具获得的vmid是相同的：

```
[root@localhost ~]# jps -l
7268 jvm-0.0.1-SNAPSHOT.jar
7531 sun.tools.jps.Jps
```

（2）用top -Hp命令查看线程的情况

```
[root@localhost ~]# top -Hp 7268
top - 05:37:05 up  6:15,  4 users,  load average: 1.00, 1.01, 0.90
Threads:  28 total,   1 running,  27 sleeping,   0 stopped,   0 zombie
%Cpu(s): 50.1 us,  0.2 sy,  0.0 ni, 49.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  2895296 total,  1670872 free,   754036 used,   470388 buff/cache
KiB Swap:  2097148 total,  2097148 free,        0 used.  1956248 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 7287 root      20   0 3289540 278632  13480 R 99.9  9.6  33:43.79 java
 7268 root      20   0 3289540 278632  13480 S  0.0  9.6   0:00.09 java
 7269 root      20   0 3289540 278632  13480 S  0.0  9.6   0:04.82 java
 7270 root      20   0 3289540 278632  13480 S  0.0  9.6   0:00.48 java
 7271 root      20   0 3289540 278632  13480 S  0.0  9.6   0:00.50 java
 7272 root      20   0 3289540 278632  13480 S  0.0  9.6   0:00.59 java
 7273 root      20   0 3289540 278632  13480 S  0.0  9.6   0:00.01 java
 7274 root      20   0 3289540 278632  13480 S  0.0  9.6   0:00.01 java
 7275 root      20   0 3289540 278632  13480 S  0.0  9.6   0:00.00 java
 7276 root      20   0 3289540 278632  13480 S  0.0  9.6   0:05.10 java
 7277 root      20   0 3289540 278632  13480 S  0.0  9.6   0:01.47 java
 7278 root      20   0 3289540 278632  13480 S  0.0  9.6   0:00.00 java
 7279 root      20   0 3289540 278632  13480 S  0.0  9.6   0:02.04 java
 7282 root      20   0 3289540 278632  13480 S  0.0  9.6   0:00.06 java
 7283 root      20   0 3289540 278632  13480 S  0.0  9.6   0:00.37 java
 7284 root      20   0 3289540 278632  13480 S  0.0  9.6   0:00.01 java
 7285 root      20   0 3289540 278632  13480 S  0.0  9.6   0:00.23 java
 7286 root      20   0 3289540 278632  13480 S  0.0  9.6   0:00.17 java
```

可以看到是线程id为7287这个线程一直在占用cpu

（3）把线程号转换为16进制

```
[root@localhost ~]# printf "%x" 7287
1c77
```

记下这个16进制的数字，下面我们要用

（4）用jstack工具查看线程栈情况

```
[root@localhost ~]# jstack 7268 | grep 1c77 -A 10
"http-nio-8080-exec-2" #16 daemon prio=5 os_prio=0 tid=0x00007fb66ce81000 nid=0x1c77 runnable
   java.lang.Thread.State: RUNNABLE
 at com.spareyaya.jvm.service.EndlessLoopService.service(EndlessLoopService.java:19)
 at com.spareyaya.jvm.controller.JVMController.endlessLoop(JVMController.java:30)
 at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
 at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
 at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
 at java.lang.reflect.Method.invoke(Method.java:498)
 at org.springframework.web.method.support.InvocableHandlerMethod.doInvoke(InvocableHandlerMe
 at org.springframework.web.method.support.InvocableHandlerMethod.invokeForRequest(InvocableH
 at org.springframework.web.servlet.mvc.method.annotation.ServletInvocableHandlerMethod.invok
```

通过jstack工具输出现在的线程栈，再通过grep命令结合上一步拿到的线程16进制的id定位到这个线程的运行情况，其中jstack后面的7268是第（1）步定位到的进程号，grep后面的是（2）、（3）步定位到的线程号。

从输出结果可以看到这个线程处于运行状态，在执行`com.spareyaya.jvm.service.EndlessLoopService.service`这个方法，代码行号是19行，这样就可以去到代码的19行，找到其所在的代码块，看看是不是处于循环中，这样就定位到了问题。

死锁

死锁并没有第一种场景那么明显，web应用肯定是多线程的程序，它服务于多个请求，程序发生死锁后，死锁的线程处于等待状态（WAITING或TIMED_WAITING），等待状态的线程不占用cpu，消耗的内存也很有限，而表现上可能是请求没法进行，最后超时了。在死锁情况不多的时候，这种情况不容易被发现。

可以使用jstack工具来查看

（1）jps查看java进程

```
[root@localhost ~]# jps -l
8737 sun.tools.jps.Jps
8682 jvm-0.0.1-SNAPSHOT.jar
```

（2）jstack查看死锁问题

由于web应用往往会有很多工作线程，特别是在高并发的情况下线程数更多，于是这个命令的输出内容会十分多。jstack最大的好处就是会把产生死锁的信息（包含是什么线程产生的）输出到最后，所以我们只需要看最后的内容就行了

```
Java stack information for the threads listed above:
===================================================
"Thread-4":
 at com.spareyaya.jvm.service.DeadLockService.service2(DeadLockService.java:35)
 - waiting to lock <0x00000000f5035ae0> (a java.lang.Object)
 - locked <0x00000000f5035af0> (a java.lang.Object)
 at com.spareyaya.jvm.controller.JVMController.lambda$deadLock$1(JVMController.java:41)
 at com.spareyaya.jvm.controller.JVMController$$Lambda$457/1776922136.run(Unknown Source)
 at java.lang.Thread.run(Thread.java:748)
"Thread-3":
 at com.spareyaya.jvm.service.DeadLockService.service1(DeadLockService.java:27)
 - waiting to lock <0x00000000f5035af0> (a java.lang.Object)
 - locked <0x00000000f5035ae0> (a java.lang.Object)
 at com.spareyaya.jvm.controller.JVMController.lambda$deadLock$0(JVMController.java:37)
 at com.spareyaya.jvm.controller.JVMController$$Lambda$456/474286897.run(Unknown Source)
 at java.lang.Thread.run(Thread.java:748)

Found 1 deadlock.
```

发现了一个死锁，原因也一目了然。

内存泄漏

我们都知道，java和c++的最大区别是前者会自动收回不再使用的内存，后者需要程序员手动释放。在c++中，如果我们忘记释放内存就会发生内存泄漏。但是，不要以为jvm帮我们回收了内存就不会出现内存泄漏。

程序发生内存泄漏后，进程的可用内存会慢慢变少，最后的结果就是抛出OOM错误。发生OOM错误后可能会想到是内存不够大，于是把-Xmx参数调大，然后重启应用。这么做的结果就是，过了一段时间后，OOM依然会出现。最后无法再调大最大堆内存了，结果就是只能每隔一段时间重启一下应用。

内存泄漏的另一个可能的表现是请求的响应时间变长了。这是因为频繁发生的GC会暂停其它所有线程（Stop The World）造成的。

为了模拟这个场景，使用了以下的程序

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class Main {
    public static void main(String[] args) {
        Main main = new Main();
        while (true) {
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            main.run();
        }
    }

    private void run() {
        ExecutorService executorService = Executors.newCachedThreadPool();
        for (int i = 0; i < 10; i++) {
            executorService.execute(() -> {
                // do something...
            });
        }
    }
}
```

运行参数是`-Xms20m -Xmx20m -XX:+PrintGC`，把可用内存调小一点，并且在发生gc时输出信息，运行结果如下

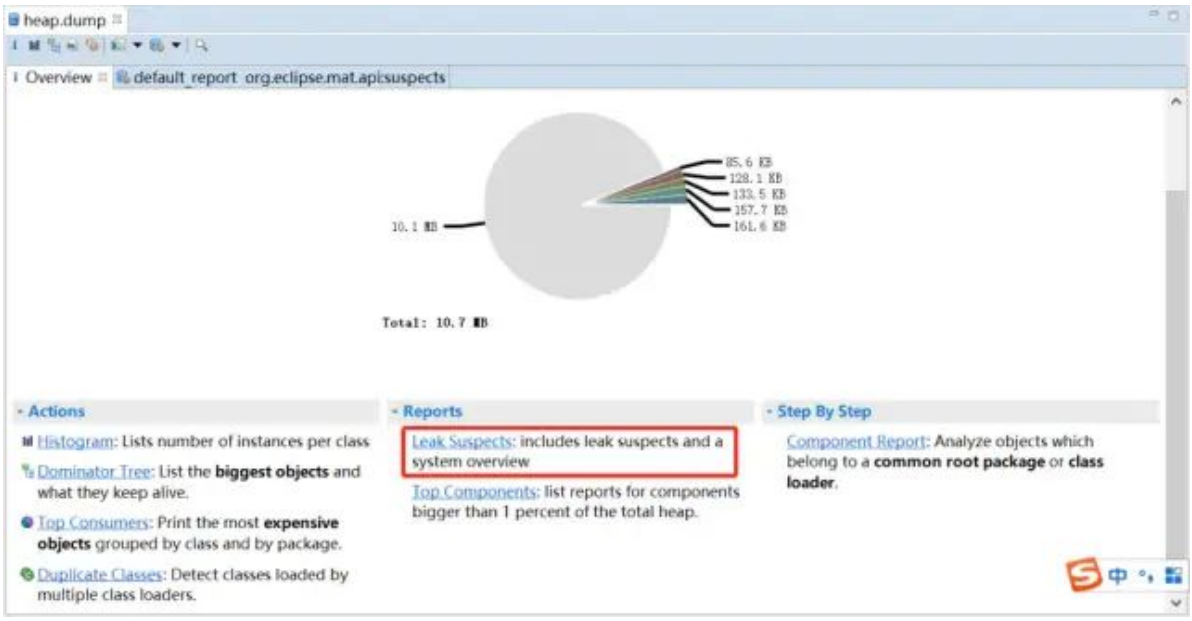
```
```

```
...
[GC (Allocation Failure)  12776K->10840K(18432K), 0.0309510 secs]
[GC (Allocation Failure)  13400K->11520K(18432K), 0.0333385 secs]
[GC (Allocation Failure)  14080K->12168K(18432K), 0.0332409 secs]
[GC (Allocation Failure)  14728K->12832K(18432K), 0.0370435 secs]
[Full GC (Ergonomics)  12832K->12363K(18432K), 0.1942141 secs]
[Full GC (Ergonomics)  14923K->12951K(18432K), 0.1607221 secs]
[Full GC (Ergonomics)  15511K->13542K(18432K), 0.1956311 secs]
...
[Full GC (Ergonomics)  16382K->16381K(18432K), 0.1734902 secs]
[Full GC (Ergonomics)  16383K->16383K(18432K), 0.1922607 secs]
[Full GC (Ergonomics)  16383K->16383K(18432K), 0.1824278 secs]
[Full GC (Allocation Failure)  16383K->16383K(18432K), 0.1710382 secs]
[Full GC (Ergonomics)  16383K->16382K(18432K), 0.1829138 secs]
[Full GC (Ergonomics) Exception in thread "main"  16383K->16382K(18432K), 0.1406222 secs]
[Full GC (Allocation Failure)  16382K->16382K(18432K), 0.1392928 secs]
[Full GC (Ergonomics)  16383K->16382K(18432K), 0.1546243 secs]
[Full GC (Ergonomics)  16383K->16382K(18432K), 0.1755271 secs]
[Full GC (Ergonomics)  16383K->16382K(18432K), 0.1699080 secs]
[Full GC (Allocation Failure)  16382K->16382K(18432K), 0.1697982 secs]
[Full GC (Ergonomics)  16383K->16382K(18432K), 0.1851136 secs]
[Full GC (Allocation Failure)  16382K->16382K(18432K), 0.1655088 secs]
java.lang.OutOfMemoryError: Java heap space
```

可以看到虽然一直在gc，占用的内存却越来越多，说明程序有的对象无法被回收。但是上面的程序对象都是定义在方法内的，属于局部变量，局部变量在方法运行结果后，所引用的对象在gc时应该被回收啊，但是这里明显没有。

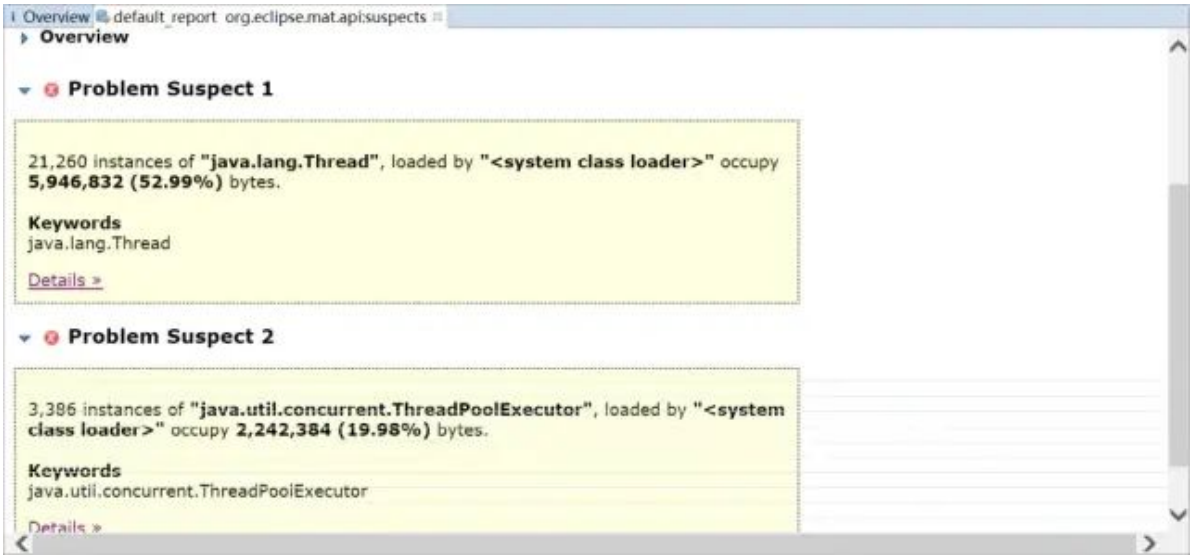
为了找出到底是哪些对象没能被回收，我们加上运行参数`-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=heap.bin`，意思是发生OOM时把堆内存信息dump出来。运行程序直至异常，于是得到heap.dump文件，然后我们借助eclipse的MAT插件来分析，如果没有安装需要先安装。

然后File->Open Heap Dump...，然后选择刚才dump出来的文件，选择Leak Suspects



图片

MAT会列出所有可能发生内存泄漏的对象



可以看到居然有21260个Thread对象，3386个ThreadPoolExecutor对象，如果你去看一下`java.util.concurrent.ThreadPoolExecutor`的源码，可以发现线程池为了复用线程，会不断地等待新的任务，线程也不会回收，需要调用其`shutdown()`方法才能让线程池执行完任务后停止。

其实线程池定义成局部变量，好的做法是设置成单例。

上面只是其中一种处理方法

在线上的应用，内存往往会设置得很大，这样发生OOM再把内存快照dump出来的文件就会很大，可能大到在本地的电脑中已经无法分析了（因为内存不足够打开这个dump文件）。这里介绍另一种处理办法：

（1）用jps定位到进程号

```
C:\Users\spareyaya\IdeaProjects\maven-project\target\classes\org\example\net>jps -l
24836 org.example.net.Main
62520 org.jetbrains.jps.cmdline.Launcher
129980 sun.tools.jps.Jps
136028 org.jetbrains.jps.cmdline.Launcher
```

因为已经知道了是哪个应用发生了OOM，这样可以直接用jps找到进程号135988

（2）用jstat分析gc活动情况

jstat是一个统计java进程内存使用情况和gc活动的工具，参数可以有很多，可以通过`jstat -help`查看所有参数以及含义

```
```

```
C:\Users\spareyaya\IdeaProjects\maven-project\target\classes\org\example\net>jstat -gcutil -t
Timestamp         S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT     GC
           29.1  32.81   0.00  23.48  85.92  92.84  84.13     14    0.339     0    0.000    0
           30.1  32.81   0.00  78.12  85.92  92.84  84.13     14    0.339     0    0.000    0
           31.1   0.00   0.00  22.70  91.74  92.72  83.71     15    0.389     1    0.233    0
```

上面是命令意思是输出gc的情况，输出时间，每8行输出一个行头信息，统计的进程号是24836，每1000毫秒输出一次信息。

输出信息是Timestamp是距离jvm启动的时间，S0、S1、E是新生代的两个Survivor和Eden，O是老年代区，M是Metaspace，CCS使用压缩比例，YGC和YGCT分别是新生代gc的次数和时间，FGC和FGCT分别是老年代gc的次数和时间，GCT是gc的总时间。虽然发生了gc，但是老年代内存占用率根本没下降，说明有的对象没法被回收（当然也不排除这些对象真的是有用）。

（3）用jmap工具dump出内存快照

jmap可以把指定java进程的内存快照dump出来，效果和第一种处理办法一样，不同的是它不用等OOM就可以做到，而且dump出来的快照也会小很多。

```
jmap -dump:live,format=b,file=heap.bin 24836
```

这时会得到heap.bin的内存快照文件，然后就可以用eclipse来分析了。

总结

以上三种严格地说还算不上jvm的调优，只是用了jvm工具把代码中存在的问题找了出来。我们进行jvm的主要目的是尽量减少停顿时间，提高系统的吞吐量。

但是如果我们没有对系统进行分析就盲目去设置其中的参数，可能会得到更坏的结果，jvm发展到今天，各种默认的参数可能是实验室的人经过多次的测试来做平衡的，适用大多数的应用场景。

如果你认为你的jvm确实有调优的必要，也务必要取样分析，最后还得慢慢多次调节，才有可能得到更优的效果。

好了，今天就分享这么多。

MySQL 开发规范，非常详细，建议收藏！

16k面试中的**10**个问题

从**0**开始搭建公司技术栈，**yyds**

全程面试辅导，保驾护航！

喜欢此内容的人还喜欢

Redis多规则限流和防重复提交思考与实践

面试专栏

图说Redis持久化 RDB和AOF，我终于全明白了！

面试专栏