大家好，在前面两个小节中，我们主要讲述了JVM内存机制的基础知识点，垃圾回收算法和垃圾收集器的工作方式等。本小节在前面的基础上进一步介绍JVM内存调优相关命令，这些命令对于我们排查线上故障相当有帮助。本节中还会介绍Java中的类加载机制相关技术知识点，希望大家可以有效理解与掌握。

（1）JVM常用内存调优命令：（重点掌握）

答： JVM在内存调优方面，提供了几个常用的命令，分别为**jps，jinfo，jstack，jmap以及jstat命令**。分别介绍如下：

- **jps：**主要用来输出JVM中运行的进程状态信息，一般使用jps命令来查看进程的状态信息，包括JVM启动参数等。
- **jinfo：**主要用来观察进程运行环境参数等信息。
- **jstack：**主要用来查看某个Java进程内的线程堆栈信息。jstack pid 可以看到当前进程中各个线程的状态信息，包括其持有的锁和等待的锁。
- **jmap：**用来查看堆内存使用状况。jmap -heap pid可以看到当前进程的堆信息和使用的GC收集器，包括年轻代和老年代的大小分配等
- **jstat：**进行实时命令行的监控，包括堆信息以及实时GC信息等。可以使用jstat -gcutil pid1000来每隔一秒来查看当前的GC信息。

这些常见的命令均为JDK提供，在这个如下的位置，各位可以自行查看。

解析：

上述命令都属于JVM提供的内存查看并且调优的常用命令，每一个命令都是极其重要的，需要大家学习与熟练掌握，对于日常开发工作有很大帮助。接下来，我们对每一个命令都给出对应的示例展示吧。

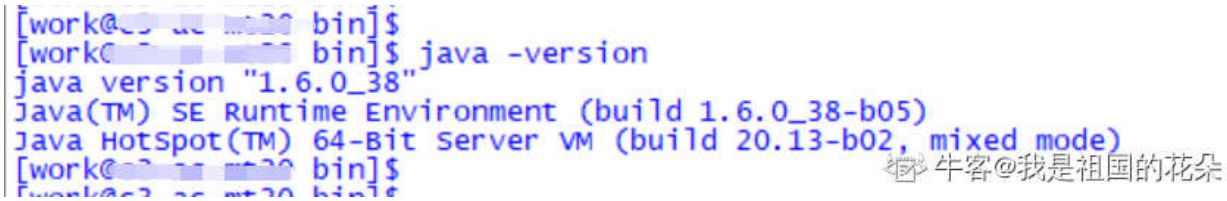
jps 用于显示当前所有java进程pid，jps经常使用的参数如下：

```
-q：仅输出VM标识符，不包括class name,jar name,arguments in main method
-m：输出main method的参数
-l：输出完全的包名，应用主类名，jar的完全路径名
-v：输出jvm参数
```

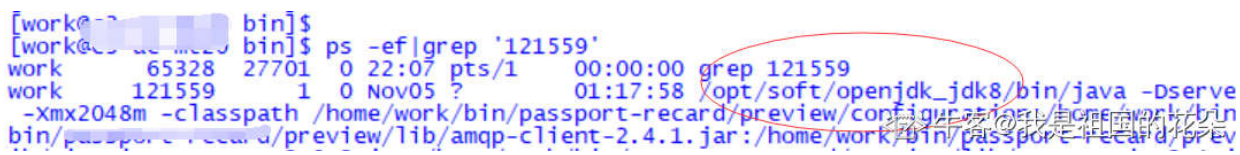
示例如下：

jinfo 可以观察进程运行环境参数，通过jinfo pid可以查看指定进程的运行环境参数，如下所示：

很遗憾，有错误发生，这个错误是指我们使用JDK版本和当前进程启动时候使用的JDK版本不一致。接着查看java -version 如下所示：



再来接着看我们进程121559所使用的java版本：



所以，我们需要使用同样版本的JDK才可以查看该进程运行时的环境参数，如下所示：

```
[work@ bin]$
[work@ bin]$ /opt/soft/openjdk1.8.0/bin/jinfo 121559
Attaching to process ID 121559, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.202-b08
Java System Properties:

client.service.level = 5
java.runtime.name = OpenJDK Runtime Environment
java.vm.version = 25.202-b08
sun.boot.library.path = /opt/soft/openjdk_jdk8/jre/lib/amd64
java.vendor.url = https://adoptopenjdk.net/
java.vm.vendor = Oracle Corporation
path.separator = :
file.encoding.pkg = sun.io
java.vm.name = OpenJDK 64-Bit Server VM
sun.os.patch.level = unknown
sun.java.launcher = SUN_STANDARD
user.country = US
xiaomi.db.connpool.maxsize = 50
user.dir = /home/work/bin/passport-recard/preview
java.vm.specification.name = Java Virtual Machine Specification
java.runtime.version = 1.8.0_202-b08
java.awt.graphicsenv = sun.awt.X11GraphicsEnvironment
basedir = /home/work/bin/passport-recard/preview
os.arch = amd64
java.endorsed.dirs = /opt/soft/openjdk_jdk8/jre/lib/endorsed
line.separator =
```


jstack 用于显示jvm中当前所有线程的运行情况和线程当前状态，一般使用的参数为-l，表示长列表，并且打印锁的相关附加信息。jstack的输出中还可以看到每一个线程当前所处的状态以及其当前所占用的锁和等待的锁，还可以检测是否存在死锁。如下所示：

jstack -l 121599

```
        at java.lang.Thread.run(Thread.java:748)

   Locked ownable synchronizers:
        - <0x0000000080a039d0> (a java.util.concurrent.ThreadPoolExecutor$Worker)

"nifty-client-worker-2" #50 daemon prio=5 os_prio=0 tid=0x00007fc8cddbd800 nid=0x1db2f runnable [0x00007fc99aaf3000]
   java.lang.Thread.State: RUNNABLE
        at sun.nio.ch.EPollArrayWrapper.epollWait(Native Method)
        at sun.nio.ch.EPollArrayWrapper.poll(EPollArrayWrapper.java:269)
        at sun.nio.ch.EPollSelectorImpl.doSelect(EPollSelectorImpl.java:93)
        at sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:86)
        - locked <0x000000008094c418> (a sun.nio.ch.Util$3)
        - locked <0x000000008094c408> (a java.util.Collections$UnmodifiableSet)
        - locked <0x000000008094c428> (a sun.nio.ch.EPollSelectorImpl)
        at sun.nio.ch.SelectorImpl.select(SelectorImpl.java:97)
        at org.jboss.netty.channel.socket.nio.SelectorUtil.select(SelectorUtil.java:68)
        at org.jboss.netty.channel.socket.nio.AbstractNioSelector.select(AbstractNioSelector.java:409)
        at org.jboss.netty.channel.socket.nio.AbstractNioSelector.run(AbstractNioSelector.java:206)
        at org.jboss.netty.channel.socket.nio.AbstractNioWorker.run(AbstractNioWorker.java:90)
        at org.jboss.netty.channel.socket.nio.NioWorker.run(NioWorker.java:178)
        at org.jboss.netty.util.ThreadRenamingRunnable.run(ThreadRenamingRunnable.java:108)
        at org.jboss.netty.util.internal.DeadLockProofWorker$1.run(DeadLockProofWorker.java:42)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
        at java.lang.Thread.run(Thread.java:748)

   Locked ownable synchronizers:
        - <0x000000008094c4f0> (a java.util.concurrent.ThreadPoolExecutor$Worker)

"nifty-client-worker-1" #49 daemon prio=5 os_prio=0 tid=0x00007fc8cddbf000 nid=0x1db2e runnable [0x00007fc99abf4000]
   java.lang.Thread.State: RUNNABLE
        at sun.nio.ch.EPollArrayWrapper.epollWait(Native Method)
        at sun.nio.ch.EPollArrayWrapper.poll(EPollArrayWrapper.java:269)
        at sun.nio.ch.EPollSelectorImpl.doSelect(EPollSelectorImpl.java:93)
        at sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:86)
        - locked <0x0000000080ad0660> (a sun.nio.ch.Util$3)
        - locked <0x0000000080ad0650> (a java.util.Collections$UnmodifiableSet)
        - locked <0x0000000080850360> (a sun.nio.ch.EPollSelectorImpl)
```


jmap 用来打印内存映射以及查看堆内存细节等，一般情况下使用-heap参数来 打印堆内存的概要信息，GC使用的算法以及堆的一些配置等信息，如下所示：

```
[work          bin]$
[work          bin]$ /opt/soft/openjdk1.8.0/bin/jmap -heap 121599
Attaching to process ID 121599, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.202-b08

using thread-local object allocation.
Parallel GC with 18 thread(s)

Heap Configuration:
   MinHeapFreeRatio         = 0
   MaxHeapFreeRatio         = 100
   MaxHeapSize              = 2147483648 (2048.0MB)
   NewSize                  = 357564416 (341.0MB)
   MaxNewSize               = 715653120 (682.5MB)
   OldSize                  = 716177408 (683.0MB)
   NewRatio                 = 2
   SurvivorRatio            = 8
   MetaspaceSize            = 21807104 (20.796875MB)
   CompressedClassSpaceSize = 1073741824 (1024.0MB)
   MaxMetaspaceSize         = 17592186044415 MB
   G1HeapRegionSize         = 0 (0.0MB)

Heap Usage:
PS Young Generation
Eden Space:
   capacity = 347078656 (331.0MB)
   used     = 288586232 (275.21727752685547MB)
   free     = 58492424 (55.782722473144 53MB)
   83.1472137543 37% used
From Space:
   capacity = 5242880 (5.0MB)
   used     = 3091520 (2.94830322265625MB)
   free     = 2151360 (2.05169677734375MB)
   58.966064453125% used
To Space:
   capacity = 5242880 (5.0MB)
   used     = 0 (0.0MB)
   free     = 5242880 (5.0MB)
```


jstat 一般用来观察GC情况，并且进行实时的分析与监控，可以使用的参数如下：

```
-class 显示ClassLoad的相关信息
-compiler 显示JIT编译的相关信息
-gc 显示和gc相关的堆信息
-gccapacity 显示各个代的容量以及使用情况
-gcmetacapacity 显示metaspace的大小
-gcnew 显示新生代信息
-gcnewcapacity 显示新生代大小和使用情况
-gcold 显示老年代和永久代的信息
-gcoldcapacity 显示老年代的大小
-gcutil 显示垃圾收集信息
-gccause 显示垃圾回收的相关信息（通-gcutil）,同时显示最后一次或当前正在发生的垃圾回收的诱因
-printcompilation 输出JIT编译的方法信息
```

我们来看一个示例 jstat -gc 121559 1000，表示每隔1000ms周期性打印该进程的GC情况，如下所示：

```
[work          bin]$
[work          bin]$ jstat -gc 121559 1000
Warning: Unresolved Symbol: sun.gc.generation.2.space.0.capacity substituted NaN
Warning: Unresolved Symbol: sun.gc.generation.2.space.0.used substituted NaN
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       PC     PU    YGC     YGCT    FGC    FGCT     GCT
5120.0 5120.0  0.0    3019.1 338944.0 281828.6  699392.0    85500.6     锟     锟     1009    4.898   2      0.070    4.969
5120.0 5120.0  0.0    3019.1 338944.0 281828.6  699392.0    85500.6     锟     锟     1009    4.898   2      0.070    4.969
^C[work          bin]$
[work          bin]$
```


好了，介绍了这么多相关命令。那么，遇到一个线上服务异常问题，我们该如何排查呢？

如何排查一个线上的服务异常?

- 首先查看当前进程的JVM启动参数，查看内存设置是否存在明显问题。
- 查看GC日志，看GC频率和时间是否明显异常。
- 查看当前进程的状态信息top -Hp pid，包括线程个数等信息。
- jstack pid查看当前的线程状态，是否存在死锁等关键信息。
- jstat -gcutil pid查看当前进程的GC情况。
- jmap -heap pid查看当前进程的堆信息，包括使用的垃圾收集器等信息。
- 用jvisiual工具打开dump二进制文件，分析是什么对象导致了内存泄漏，定位到代码处，进行code review。

一般情况下，我们在测试环境上线新服务的时候，应该重点关注并且查看当前新服务的内存使用以及回收情况，避免新服务种出现内存异常导致服务崩溃的现象发生。

（2）JDK8中在内存管理上的变化：

答： JDK8中出现了**元空间代替了永久代**。元空间和永久代类似，都是对JVM规范中方法区的实现。区别在于元空间并不在虚拟机中，而是使用**本地内存，**默认情况下元空间的大小仅受本地内存限制，也可以通过**-XX：MetaspaceSize指定元空间大小。**

为什么要使用元空间代替永久代?

字符串在永久代中，容易出现性能问题和内存溢出的问题。类和方法的信息等比较难确定大小，因此对于永久代的大小指定比较困难，太小容易出现永久代溢出。使用元空间则使用了本地内存。

（3）Java中的类加载机制有了解吗？（重点掌握）

答： Java中的类加载机制指虚拟机**把描述类的数据从 Class 文件加载到内存**，并对数据进行**校验、转换、解析和初始化**，最终形成可以被虚拟机直接使用的 Java 类型。

类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括了：**加载、验证、准备、解析、初始化、使用、卸载七个阶段**。类加载机制的保持则包括前面五个阶段。

- **加载：**

加载是指**将类的.class文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个java.lang.Class对象，用来封装类在方法区内的数据结构。**

- **验证：**

验证的作用是确保被加载的类的正确性，包括文件格式验证，元数据验证，字节码验证以及符号引用验证。

- **准备：**

准备阶段为类的静态变量分配内存，并将其初始化为默认值。假设一个类变量的定义为public static int val = 3;那么变量val在准备阶段过后的初始值不是3而是0。

- **解析：**

解析阶段将类中**符号引用转换为直接引用**。符号引用以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能够无歧义的定位到目标即可。

- **初始化：**

初始化阶段**为类的静态变量赋予正确的初始值**，JVM负责对类进行初始化，主要对类变量进行初始化。

解析：

关于类加载机制的考察也是JVM知识点的重中之重，上边我们介绍了类加载机制的过程以及其基本作用。接下来，我们看下类加载器有哪几种呢？类加载器的职责又是什么呢？

类加载器的分类：

- **启动类加载器（Bootstrap ClassLoader）：**

启动类加载器负责加载存放在JDK\jre\lib(JDK代表JDK的安装目录，下同)下，或被-Xbootclasspath参数指定的路径中的类。

- **扩展类加载器（ExtClassLoader）：**

扩展类加载器负责加载JDK\jre\lib\ext目录中，或者由java.ext.dirs系统变量指定的路径中的所有类库（如javax.*开头的类）。

- **应用类加载器（AppClassLoader）：**

应用类加载器负责加载用户类路径（ClassPath）所指定的类，开发者可以直接使用该类加载器。

类加载器的职责：

- **全盘负责：**

当一个类加载器负责加载某个Class时，该Class所依赖的和引用的其他Class也将由该类加载器负责载入，除非显式使用另外一个类加载器来载入。

- **父类委托：**

类加载机制会先让父类加载器试图加载该类，只有在父类加载器无法加载该类时才尝试从自己的类路径中加载该类。

父类委托机制是为了**防止内存中出现多份同样的字节码，**保证java程序安全稳定运行。

- **缓存机制：**

缓存机制将会保证所有加载过的Class都会被缓存，当程序中需要使用某个Class时，先从缓存区寻找该Class，只有缓存区不存在，系统才会读取该类对应的二进制数据，并将其转换成Class对象，存入缓存区。这就是为什么修改了Class后，必须重启JVM，程序的修改才会生效。

总结：

本小节中我们主要介绍了JVM内存相关的调优命令，并且较为详细的阐述了Java的类加载机制。通过三个小节的学习，我们基本将工作中常见的内存相关知识以及高频面试考点阐述了一遍。JVM是Java面试中必考的知识点，**考察可深可浅**，要求不一样。如果我们想拿下知名大厂的Offer，那么JVM相关知识点也需要较为深入的研究与学习。这里强烈推荐周志明老师的**《深入理解Java虚拟机：JVM高级特性与最佳实践》**。

限于作者水平，文章中难免会有不妥之处。大家在学习过程中遇到我没有表达清楚或者表述有误的地方，欢迎随时在文章下边指出，我会及时关注，随时改正。另外，大家有任何话题都可以在下边留言，我们一起交流探讨。

5

讨论 评论

柳杰201905011049420 1#

类加载机制这块不太懂

发表于 2020-01-01 16:33:04 赞(0) 回复(0)

刘畅201904211606816 2#

打卡

发表于 2020-01-03 20:33:57 赞(0) 回复(0)

牧水s 3#

打卡

发表于 2020-01-18 21:37:02 赞(0) 回复(0)

爱生活的猿 4#

类加载这里错了，加载来的Class对象存放到方法区中，不会在堆中创建。详见深入理解Java虚拟机P215最下面。书内有详细说明

发表于 2020-03-11 13:48:40 赞(1) 回复(1)

我是祖国的花朵 作者 ： 你好，你看的是周老师的深入理解虚拟机第二版，目前该书已经出版了第三版，P268中间，明确说了是在堆区中实例化的该Class类对象，这个对象作为程序访问方法区中的类型（类或者接口）数据的外部接口。

2020-03-15 17:30:14 赞(4) 回复(0)

请输入你的观点

回复

sky爱吃青菜 5#

笔记：

java有两种对象：实例对象和Class对象。每个类的运行时的**类型信息**就是用Class对象表示的。它包含了与类有关的信息。实例对象就通过Class对象来创建的。

有三种获得Class对象的方式：

1. Class.forName(“类的全限定名”)
2. 实例对象.getClass()

3. 类名.class （类字面常量）

发表于 今天 10:30:56 赞(0) 回复(0)

5