JVM第十四天-JVM调优-调优实战和常用工具

**调优前的基础概念**

1、吞吐量：用户代码时间 /（用户代码执行时间 + 垃圾回收时间）

2、响应时间：STW越短，响应时间越好

所谓调优，首先确定，追求啥？吞吐量优先，还是响应时间优先？还是在满足一定的响应时间的情况下，要求达到多大的吞吐量…

**场景分析选择**

像类似于科学计算，数据挖掘，这种追求吞吐量优先的，一般选用（PS + PO）垃圾回收器组合

对于对响应时间要求比较高的，类似于网站，GUI渲染， API调用等，如果是JDK1.8之前，可以选用(PAR NEW+CMS)组合，在JDK1.8开始，可以选用 G1。

**什么是调优？**

1、根据需求进行JVM规划和预调优

2、优化运行JVM运行环境（慢，卡顿）

3、解决JVM运行过程中出现的各种问题(OOM)

**1、调优，从规划开始**

调优，从业务场景开始，没有业务场景的调优都是耍流氓

无监控（压力测试，能看到结果），不调优

步骤：

1、熟悉业务场景（没有最好的垃圾回收器，只有最合适的垃圾回收器，看你到底追求的是什么？）

如果追求响应时间、停顿时间 [CMS G1 ZGC] （需要给用户作响应）

如果追求吞吐量（吞吐量 = 用户时间 /( 用户时间 + GC时间)） [PS]

2、选择回收器组合

3、计算内存需求（经验值 1.5G 16G，但内存也不是越大越好）

4、选定CPU（越高越好）

5、设定年代大小、升级年龄

6、设定GC日志参数

|  |
| --- |
| -Xloggc:/opt/xxx/logs/xxx-xxx-gc-%t.log //设定日志路径  -XX:+UseGCLogFileRotation //循环使用GC日志文件 也就是说，假如设定了5个20M的日志文件，当都满了时，接下来的日志将覆盖第一个的日志继续开始写。  -XX:NumberOfGCLogFiles=5 //生成GC日志文件数  -XX:GCLogFileSize=20M //每个GC日志文件  -XX:+PrintGCDetails //打印GC详情  -XX:+PrintGCDateStamps 打印GC时间戳  -XX:+PrintGCCause //打印GC原因 |

**2、优化环境**

**1、系统CPU经常100%，如何调优？ CPU100%那么一定有线程在占用系统资源。**

找出哪个进程cpu高（top）

该进程中的哪个线程cpu高（top -Hp）

导出该线程的堆栈 (jstack)

查找哪个方法（栈帧）消耗时间 (jstack)

工作线程占比高 | 垃圾回收线程占比高

**2、系统内存飙高，如何查找问题？**

导出堆内存 (jmap)

分析 (jhat jvisualvm mat jprofiler … )

**3、如何监控JVM**

jstat

jvisualvm

jprofiler

arthas

top…

**3、解决JVM运行中的问题**

一个案例理解常用工具

测试代码：

|  |
| --- |
| package com.peng.jvm.gc;  import java.math.BigDecimal;  import java.util.ArrayList;  import java.util.Date;  import java.util.List;  import java.util.concurrent.ScheduledThreadPoolExecutor;  import java.util.concurrent.ThreadPoolExecutor;  import java.util.concurrent.TimeUnit;  /\*\*  \* 从数据库中读取信用数据，套用模型，并把结果进行记录和传输  \*/  public class T15\_FullGC\_Problem01 {  private static class CardInfo {  BigDecimal price = new BigDecimal(0.0);  String name = "张三";  int age = 5;  Date birthdate = new Date();  public void m() {}  }  private static ScheduledThreadPoolExecutor executor = new ScheduledThreadPoolExecutor(50,  new ThreadPoolExecutor.DiscardOldestPolicy());  public static void main(String[] args) throws Exception {  executor.setMaximumPoolSize(50);  for (;;){  modelFit();  Thread.sleep(100);  }  }  private static void modelFit(){  List<CardInfo> taskList = getAllCardInfo();  taskList.forEach(info -> {  // do something  executor.scheduleWithFixedDelay(() -> {  //do sth with info  info.m();  }, 2, 3, TimeUnit.SECONDS);  });  }  private static List<CardInfo> getAllCardInfo(){  List<CardInfo> taskList = new ArrayList<>();  for (int i = 0; i < 100; i++) {  CardInfo ci = new CardInfo();  taskList.add(ci);  }  return taskList;  }  } |

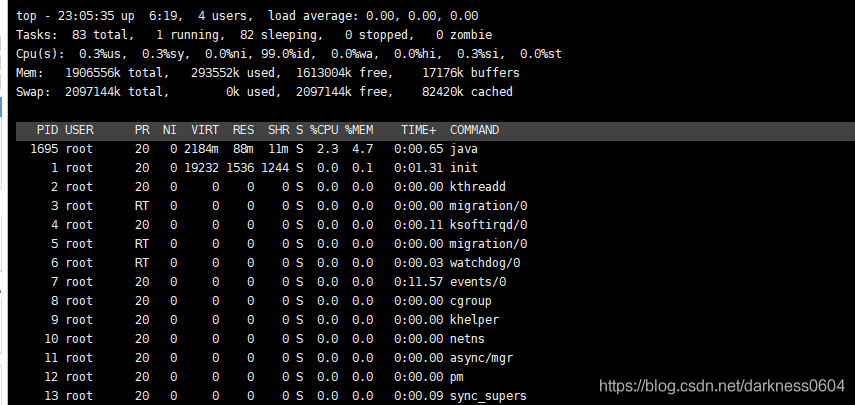
运行程序

|  |
| --- |
| java -Xms200M -Xmx200M -XX:+PrintGC com.peng.jvm.gc.T15\_FullGC\_Problem01 |

**定位线程维度的问题**

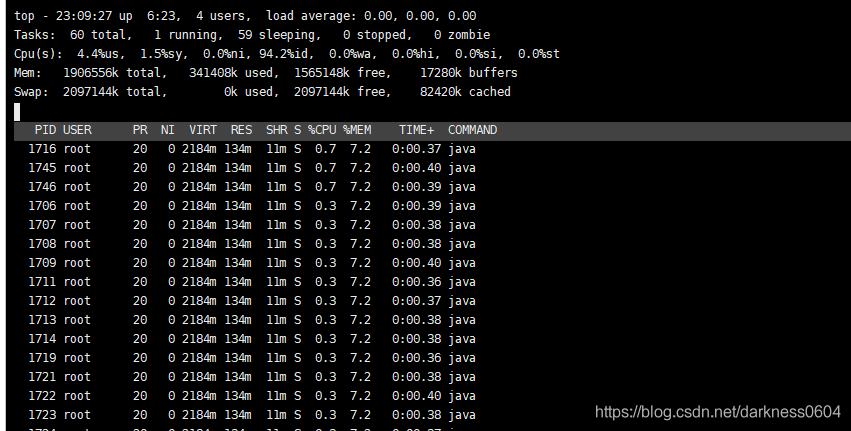
1、首先通过top命令观察到问题：内存不断增长 CPU占用率居高不下

|  |
| --- |
| top |



2、发现是PID为1695的java进程的CPU和内存一直在增长。使用top -Hp 观察该进程中的线程，哪个线程CPU和内存占比高

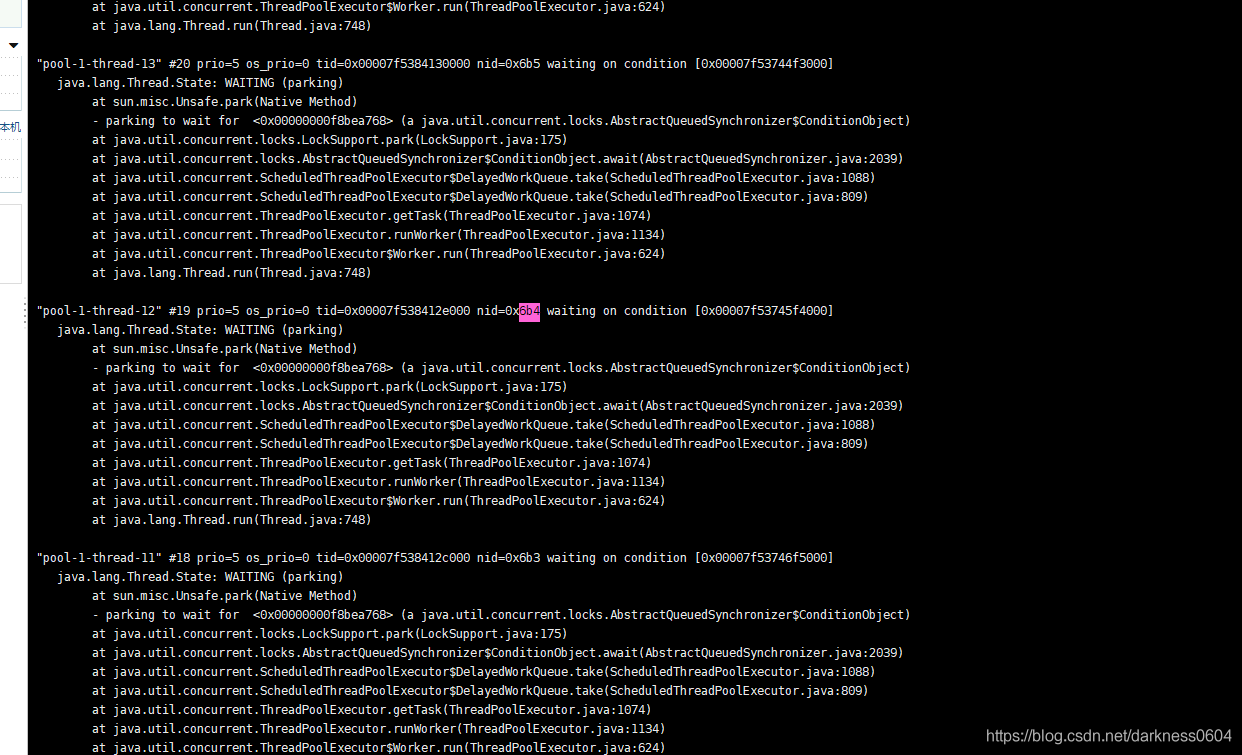
|  |
| --- |
| top -Hp 1695 |



找到CPU和内存占用居高不下的线程号，记住。

3、使用jstack 定位进程中的各线程的堆栈状况，找到对应的线程，需要注意的是，jstack线程的nid对应的是线程pid的16进制表现形式，这里拿1716举例，那么就应该找到nid=0x6b4的线程。

|  |
| --- |
| jstack 1695 |



可以看到这个线程现在处于WAITING的状态

并且可以看到在等待的锁对象是ConditionObject：

|  |
| --- |
| parking to wait for <0x00000000f8bea768> (a java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject) |

假如有一个进程中，很多线程都在waiting on ，一定要找到是哪个线程持有这把锁

怎么找？搜索jstack dump的信息，找 ，看哪个线程持有这把锁，并且状态是RUNNABLE的。

从jstack打印的线程名称，我们应该就意识到了为什么阿里规范里规定，线程的名称（尤其是线程池）都要写有意义的名称了，为了方便排查是哪个线程出现的问题。怎么样自定义线程池里的线程名称？（自定义ThreadFactory）。

通过上述操作，可以判断是否是由于死锁导致的CPU和内存飙高。

如果不是，可以再看占用较高的线程是否是一个垃圾回收线程，如果是的话，很有可能是系统频繁在发生GC

以上我们使用了top和jstack进行定位了哪个线程是可能存在问题的。

**定位疑似GC的问题**

**1、jinfo pid**

可以查看进程内的一些变量信息（这个和GC一般不太相关，这里只是提及一下）

**2、jstat -gc**

打印gc的一些情况，但格式不是很友好。

每隔500个毫秒打印GC的情况:

jstat -gc pid 500

**3、Jconsole远程连接**

比jstat友好一些，可以提供运行程序的各种信息（包括堆信息）的可视化界面。 基于JMX进行监控通信

1、程序启动加入参数：

|  |
| --- |
| java -Djava.rmi.server.hostname=192.168.17.11  -Dcom.sun.management.jmxremote  -Dcom.sun.management.jmxremote.port=11111  -Dcom.sun.management.jmxremote.authenticate=false  -Dcom.sun.management.jmxremote.ssl=false XXX |

2、关闭linux防火墙（实战中应该打开对应端口）

|  |
| --- |
| service iptables stop  chkconfig iptables off #永久关闭 |

3、windows上打开 jconsole远程连接 192.168.17.11:11111

**4、jvisualvm远程连接**

https://www.cnblogs.com/liugh/p/7620336.html （简单做法）

也是java提供的一种用于监控应用信息的可视化界面，和Jconsole差不太多，但比Jconsole要更好看一些，一般可视化界面可以用这个。

注意：上述的3,4的两种图像已经上线的系统一般不能直接开启JMX，然后使用远程监控的图形界面去定位OOM问题。

为什么呢？因为这意味着线上服务器的资源有部分是为attach监测数据做消耗的，而且attach的动作也比较影响服务器性能。

那不用图形界面可以用什么？得往命令行那方面想。。

拿图形界面到底能用在什么地方？测试！上线前测试的时候可以进行监控！（压测观察）

**5、jmap -histo**

一种命令行的监测命令。查看有哪些对象产生的比较多（可能会对线上运行系统性能造成影响，但可以接受）

|  |
| --- |
| jmap - histo 4655 | head -10 |

|  |
| --- |
| [root@dream01 jvm]# jmap -histo 1871 | head -10  num #instances #bytes class name  ----------------------------------------------  1: 946527 68149944 java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask  2: 946553 37862120 java.math.BigDecimal  3: 946527 30288864 T15\_FullGC\_Problem01$CardInfo  4: 946527 22716648 java.util.Date  5: 946527 22716648 java.util.concurrent.Executors$RunnableAdapter  6: 946527 15144432 T15\_FullGC\_Problem01$$Lambda$2/1044036744  7: 1 5391952 [Ljava.util.concurrent.RunnableScheduledFuture; |

**6、jmap -dump**

导出堆的整个转储文件

|  |
| --- |
| jmap -dump:format=b,file=xxx pid |

这个线上慎用，一般来说线上系统的话，内存都会特别大，jmap -dump导出期间会对进程产生很大影响，甚至卡顿（电商不适合）

到这里，可以发现，对于线上系统来说，远程监测可视化界面也不行，用jmap吧，一次性导出又会严重影响线上性能，那该咋办呢？

可以从以下方向去处理：

1、设定JVM参数HeapDumpOnOutOfMemoryError ，OOM的时候会自动产生堆转储文件

|  |
| --- |
| java -XX:+HeapDumpOnOutOfMemoryError com.peng.jvm.gc.T15\_FullGC\_Problem01 |

但其实也不是特别靠谱，因为内存因GC完全崩溃的时候，当你再去观察的时候，这个文件可能会非常大，难以读取。不过内存崩溃了，JVM进程也没终止，此时依旧可以谁用jmap dump去导出。

2、使用阿里的arthas进行在线定位

3、实际上生产环境基本都是多实例部署的，因此实际可以停掉某一台进行排查，而不影响整体对外服务。

**7、使用MAT / jhat /jvisualvm 进行dump文件分析**

一般来说，我们通过无论是通过jmap的dump命令，还是arthas的heap dump命令，都会导出一个.hprof的堆存储文件，接下来，我们就是需要对这份堆存储文件进行分析了。

这里举例通过jhat进行定位：

https://www.cnblogs.com/baihuitestsoftware/articles/6406271.html

一般需要通过-J-mx来指定最多可用内存是多少给它，如果不指定，那么堆文件多少，就会分配多大的内存给它。

|  |
| --- |
| jhat -J-mx512M ./20200323.hprof |

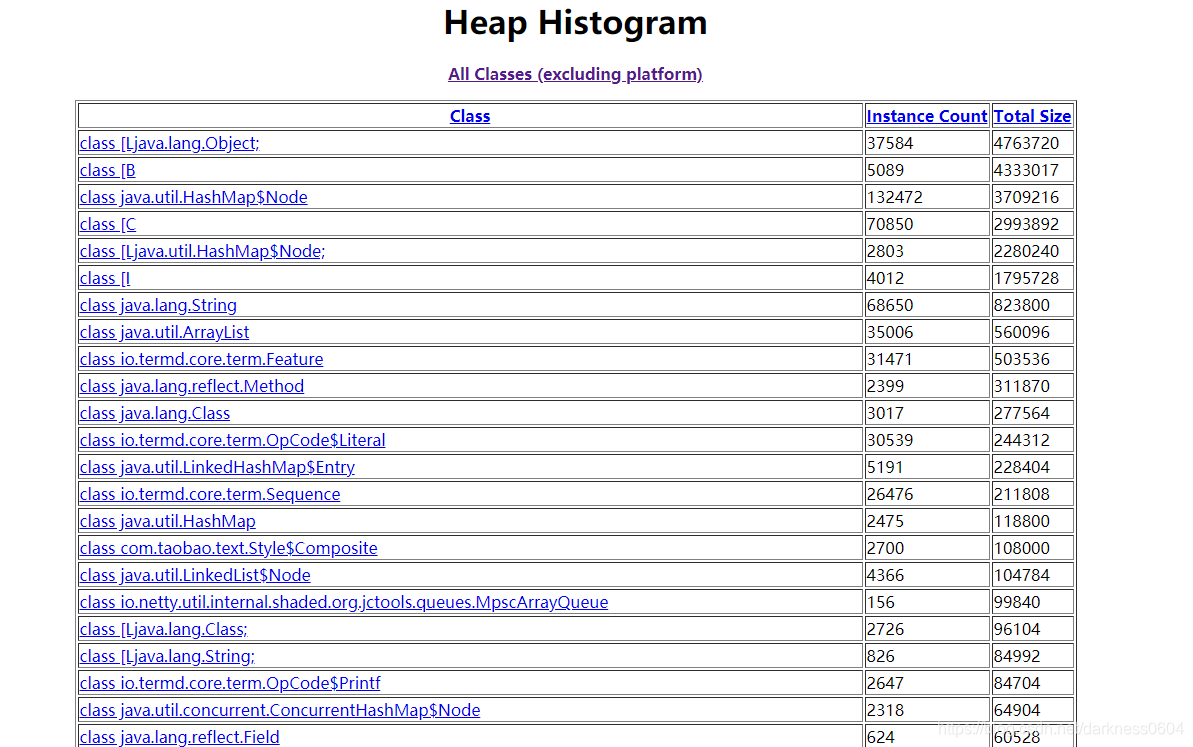
|  |
| --- |
| Chasing references, expect 104 dots........................................................................................................  Eliminating duplicate references........................................................................................................  Snapshot resolved.  Started HTTP server on port 7000  Server is ready. |

到此，已经开启了一个7000的HTTP端口。

访问这个端口：http://192.168.221.66:7000

拉到最后：找到对应链接。

在这里可以看到各种类的占用大小情况：



还可以使用OQL查找特定问题对象：

