**生产问题OOM**

## OutOfMemoryError，内存不足

**1 java.lang.OutOfMemoryError**

这个错误我相信大部分开发人员都有遇到过，产生该错误的原因大都出于以下原因：JVM内存过小、程序不严密，产生了过多的垃圾。

**2 导致OutOfMemoryError异常的常见原因有以下几种：**

（1）内存中加载的数据量过于庞大，如一次从数据库取出过多数据；

（2）集合类中有对对象的引用，使用完后未清空，使得JVM不能回收；

（3）代码中存在死循环或循环产生过多重复的对象实体；

（4）使用的第三方软件中的BUG；

（5）启动参数内存值设定的过小；

（6）检查代码中是否有死循环或递归调用。

（7）检查是否有大循环重复产生新对象实体。

（8）检查对数据库查询中，是否有一次获得全部数据的查询。一般来说，如果一次取十万条记录到内存，就可能引起内存溢出。

（9）这个问题比较隐蔽，在上线前，数据库中数据较少，不容易出问题，上线后，数据库中数据多了，一次查询就有可能引起内存溢出。

因此对于数据库查询尽量采用分页的方式查询。

（10）检查List、MAP等集合对象是否有使用完后，未清除的问题。List、MAP等集合对象会始终存有对对象的引用，使得这些对象不能被GC回收。

**3 此错误常见的错误提示：**

tomcat:java.lang.OutOfMemoryError: PermGen space

tomcat:java.lang.OutOfMemoryError: Java heap space

weblogic:Root cause of ServletException java.lang.OutOfMemoryError

resin:java.lang.OutOfMemoryError

java:java.lang.OutOfMemoryError

**4 解决java.lang.OutOfMemoryError的方法有如下几种**

**（1）增加jvm的内存大小。**

1）在执行某个class文件时候，可以使用java -Xmx256M aa.class来设置运行aa.class时jvm所允许占用的最大内存为256M。 2）对tomcat容器，可以在启动时对jvm设置内存限度。对tomcat，可以在catalina.bat中添加：

set CATALINA\_OPTS=-Xms128M -Xmx256M

set JAVA\_OPTS=-Xms128M -Xmx256M

或者把%CATALINA\_OPTS%和%JAVA\_OPTS%代替为-Xms128M -Xmx256M

**（2）优化程序，释放垃圾。**

主要包括避免死循环，应该及时释放种资源：内存, 数据库的各种连接，防止一次载入太多的数据。导致java.lang.OutOfMemoryError的根本原因是程序不健壮。因此，从根本上解决Java内存溢出的唯一方法就是修改程序，及时地释放没用的对象，释放内存空间。 遇到该错误的时候要仔细检查程序，嘿嘿，遇多一次这种问题之后，以后写程序就会小心多了。

**5 tomcat中java.lang.OutOfMemoryError: PermGen space异常处理**

PermGen space的全称是Permanent Generation space,是指内存的永久保存区域,这块内存主要是被JVM存放Class和Meta信息的,Class在被Loader时就会被放到PermGen space中, 它和存放类实例(Instance)的Heap区域不同,GC(Garbage Collection)不会在主程序运行期对PermGen space进行清理，所以如果你的应用中有很多CLASS的话,就很可能出现PermGen space错误, 这种错误常见在web服务器对JSP进行pre compile的时候。如果你的WEB APP下都用了大量的第三方jar, 其大小超过了jvm默认的大小(4M)那么就会产生此错误信息了。

解决方法： 手动设置MaxPermSize大小修改TOMCAT\_HOME/bin/catalina.sh在

echo "Using CATALINA\_BASE: $CATALINA\_BASE"

上面加入以下行：

JAVA\_OPTS="-server -XX:PermSize=64M -XX:MaxPermSize=128m

建议：将相同的第三方jar文件移置到tomcat/shared/lib目录下，这样可以达到减少jar 文档重复占用内存的目的。

## 内存泄露

内存泄漏是指无用对象（不再使用的对象）**持续占有内存或无用对象的内存得不到及时释放，从而造成内存空间的浪费称为内存泄漏。**内存泄露有时不严重且不易察觉，这样开发者就不知道存在内存泄露，但有时也会很严重，会提示你Out of memory。Java内存泄漏的根本原因是什么呢？

**长生命周期的对象持有短生命周期对象的引用就很可能发生内存泄漏，尽管短生命周期对象已经不再需要，但是因为长生命周期持有它的引用而导致不能被回收**，这就是Java中内存泄漏的发生场景。

**1、静态集合类引起内存泄漏：**

像HashMap、Vector等的使用最容易出现内存泄露，这些静态变量的生命周期和应用程序一致，他们所引用的所有的对象Object也不能被释放，因为他们也将一直被Vector等引用着。

例如

Static Vector v = new Vector(10);

for (int i = 1; i<100; i++)

{

Object o = new Object();

v.add(o);

o = null;

}

在这个例子中，循环申请Object 对象，并将所申请的对象放入一个Vector 中，如果仅仅释放引用本身（o=null），那么Vector 仍然引用该对象，所以这个对象对GC 来说是不可回收的。因此，如果对象加入到Vector 后，还必须从Vector 中删除，最简单的方法就是将Vector对象设置为null。

**2、当集合里面的对象属性被修改后，再调用remove()方法时不起作用。**

例如：

public static void main(String[] args)

{

Set<Person> set = new HashSet<Person>();

Person p1 = new Person("唐僧","pwd1",25);

Person p2 = new Person("孙悟空","pwd2",26);

Person p3 = new Person("猪八戒","pwd3",27);

set.add(p1);

set.add(p2);

set.add(p3);

System.out.println("总共有:"+set.size()+" 个元素!"); //结果：总共有:3 个元素!

p3.setAge(2); //修改p3的年龄,此时p3元素对应的hashcode值发生改变

set.remove(p3); //此时remove不掉，造成内存泄漏

set.add(p3); //重新添加，居然添加成功

System.out.println("总共有:"+set.size()+" 个元素!"); //结果：总共有:4 个元素!

for (Person person : set)

{

System.out.println(person);

}

}

**3、监听器**

在java 编程中，我们都需要和监听器打交道，通常一个应用当中会用到很多监听器，我们会调用一个控件的诸如addXXXListener()等方法来增加监听器，但往往在释放对象的时候却没有记住去删除这些监听器，从而增加了内存泄漏的机会。

**4、各种连接**

比如数据库连接（dataSourse.getConnection()），网络连接(socket)和io连接，除非其显式的调用了其close（）方法将其连接关闭，否则是不会自动被GC 回收的。对于Resultset 和Statement 对象可以不进行显式回收，但Connection 一定要显式回收，因为Connection 在任何时候都无法自动回收，而Connection一旦回收，Resultset 和Statement 对象就会立即为NULL。但是如果使用连接池，情况就不一样了，除了要显式地关闭连接，还必须显式地关闭Resultset Statement 对象（关闭其中一个，另外一个也会关闭），否则就会造成大量的Statement 对象无法释放，从而引起内存泄漏。**这种情况下一般都会在try里面去的连接，在finally里面释放连接。**

**5、内部类和外部模块的引用**

内部类的引用是比较容易遗忘的一种，而且一旦没释放可能导致一系列的后继类对象没有释放。此外程序员还要小心外部模块不经意的引用，例如程序员A 负责A 模块，调用了B 模块的一个方法如：

public void registerMsg(Object b);

这种调用就要非常小心了，传入了一个对象，很可能模块B就保持了对该对象的引用，这时候就需要注意模块B 是否提供相应的操作去除引用。

**6、单例模式**

不正确使用单例模式是引起内存泄漏的一个常见问题，**单例对象在初始化后将在JVM的整个生命周期中存在（以静态变量的方式），如果单例对象持有外部的引用，那么这个对象将不能被JVM正常回收**，导致内存泄漏，考虑下面的例子：

class A{

public A(){

B.getInstance().setA(this);

}

....

}

//B类采用单例模式

class B{

private A a;

private static B instance=new B();

public B(){}

public static B getInstance(){

return instance;

}

public void setA(A a){

this.a=a;

}

//getter...

}

## 线程死锁

**1 实例分析**

public class DeadThread implements Runnable {

public String username;

public Object lock1 = new Object();

public Object lock2 = new Object();

@Override

public void run() {

if (username.equals("a")) {

synchronized (lock1) {

try {

System.out.println("username = " + username);

System.out.println(Thread.currentThread().getName());

Thread.sleep(3000);

} catch (Exception e) {

// TODO: handle exception

e.printStackTrace();

}

synchronized (lock2) {

System.out.println("按lock1->lock2的顺序执行代码");

}

}

}

if (username.equals("b")) {

synchronized (lock2) {

try {

System.out.println("username = " + username);

System.out.println(Thread.currentThread().getName());

Thread.sleep(3000);

} catch (Exception e) {

e.printStackTrace();

}

synchronized (lock1) {

System.out.println("按lock2->lock1顺序执行代码");

}

}

}

}

public void setFlag(String username) {

this.username = username;

}

public static void main(String[] args) {

DeadThread dt1 = new DeadThread();

dt1.setFlag("a");

Thread t1 = new Thread(dt1);

t1.start();

try {

Thread.sleep(2000);

} catch (InterruptedException e) {

e.printStackTrace();

}

dt1.setFlag("b");

Thread t2 = new Thread(dt1);

t2.start();

}

}

**2 分析和解决**

使用jstack

## 锁争用（Lock Contention）

区分竞争锁和非竞争锁对性能的影响非常重要。如果一个锁自始至终只被一个线程使用，那么 JVM 有能力优化它带来的绝大部分损耗。如果一个锁被多个线程使用过，但是在任意时刻，都只有一个线程尝试获取锁，那么它的开销要大一些。我们将以上两种锁称为非竞争锁。而对性能影响最严重的情况出现在多个线程同时尝试获取锁时。这种情况是 JVM 无法优化的，而且通常会发生从用户态到内核态的切换。现代 JVM 已对非竞争锁做了很多优化，使它几乎不会对性能造成影响。常见的优化有以下几种。

如果一个锁对象只能由当前线程访问，那么其他线程无法获得该锁并发生同步 , 因此 JVM 可以去除对这个锁的请求。

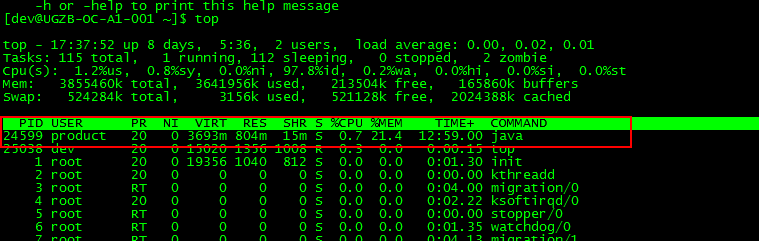
逸出分析 (escape analysis) 可以识别本地对象的引用是否在堆中被暴露。如果没有，就可以将本地对象的引用变为线程本地的 (thread local) 。

编译器还可以进行锁的粗化 (lock coarsening) 。把邻近的 synchronized 块用相同的锁合并起来，以减少不必要的锁的获取和释放。

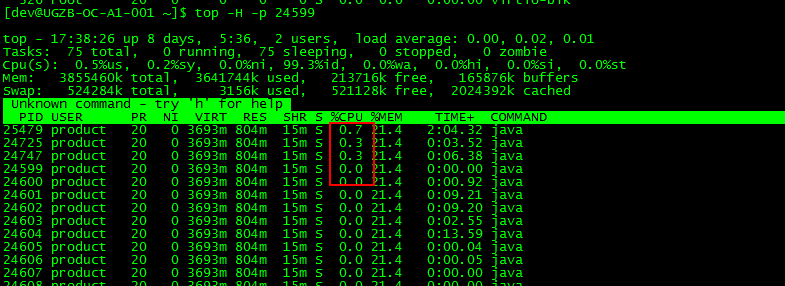
因此，不要过分担心非竞争锁带来的开销，要关注那些真正发生了锁竞争的临界区中性能的优化。

## Java进程消耗CPU过高

1 直接使用**top**命令，找到占用CPU、内存最高(较高)的java进程。



2 使用**top -H -p 24599**，找到该进程占用cpu/内存的线程；



**3 printf %x 15664线程id转换成十六进制**

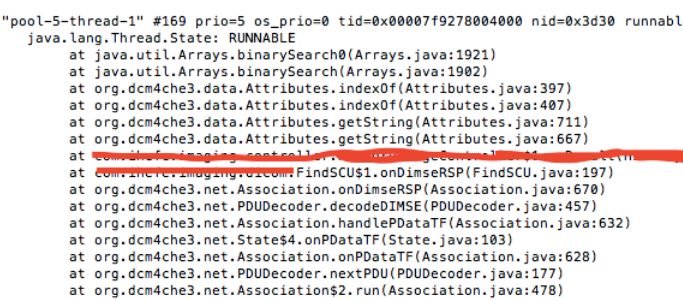


**4 使用jstack查询线程的堆栈信息**

jstack <pid> | grep -a 线程id（十六进制）

**jstack 24599|grep -a 6387**



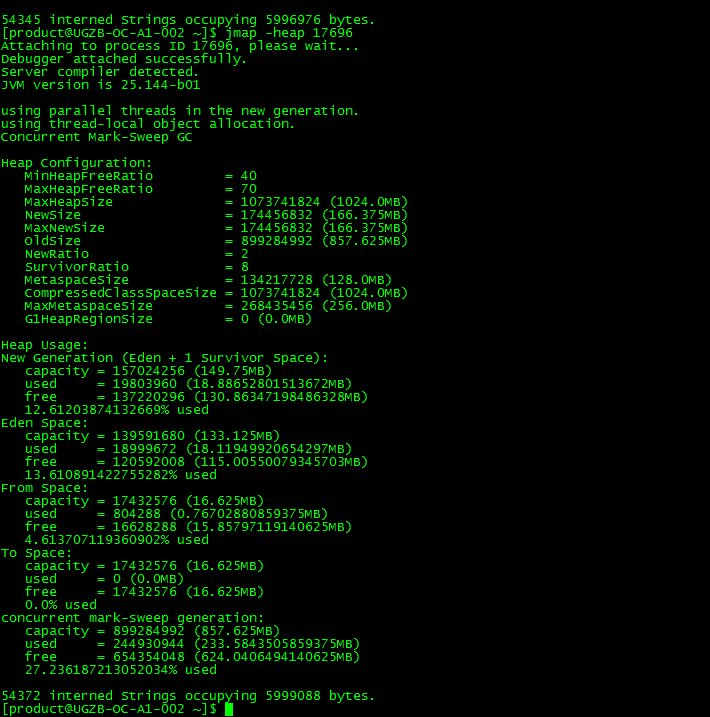


## OutOfMemoryError：Java heap space

**1 用jmap查看进程堆信息**

**ps -ef|grep java**

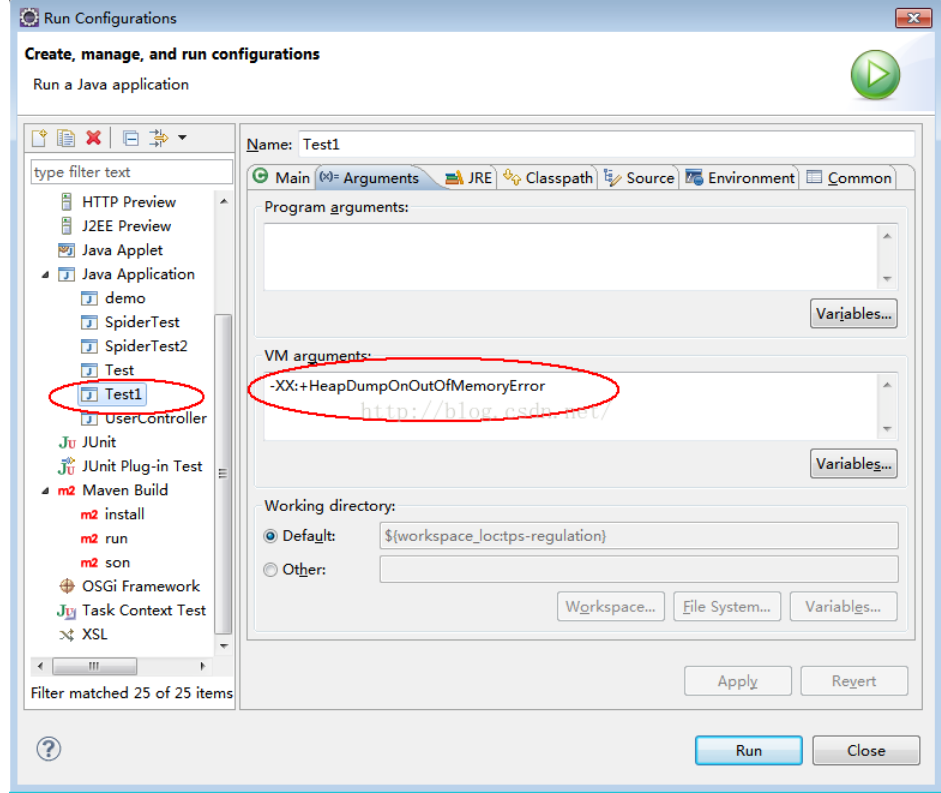
**jmap -heap 17696**

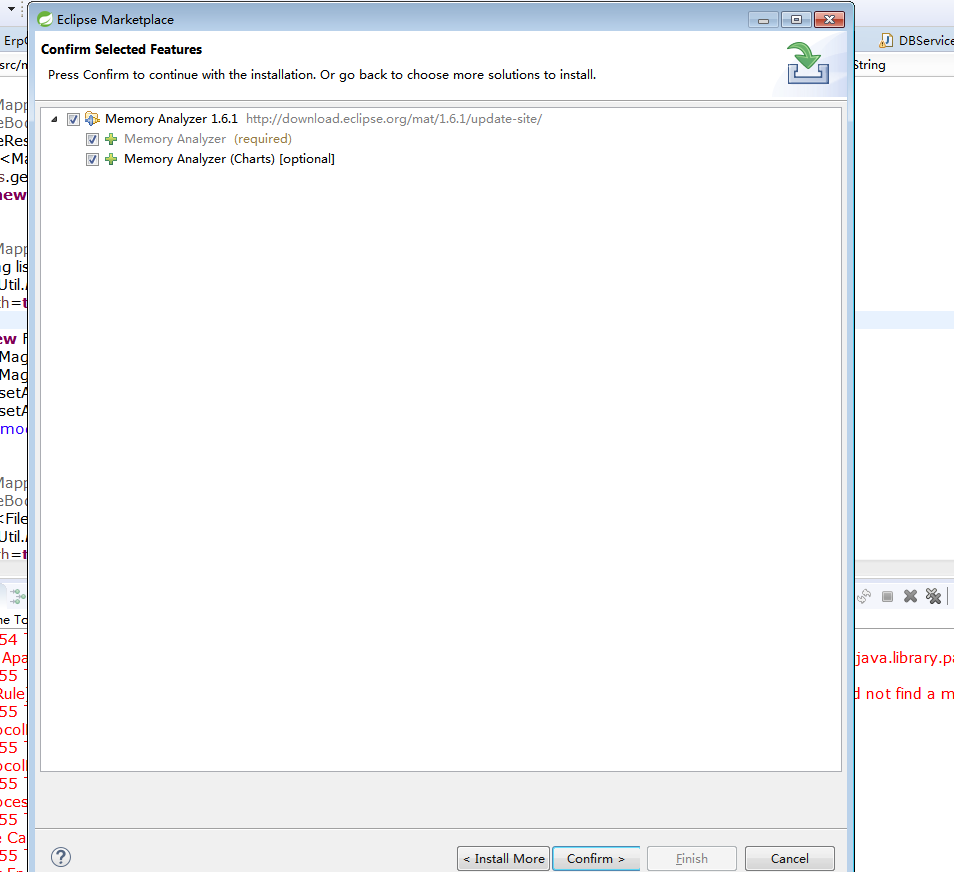


**2 找出第一次出现Caused by: java.lang.OutOfMemoryError: Java heap space的地方**，分析前面的操作，极有可能造成oom.

**3 使用dump分析工具**

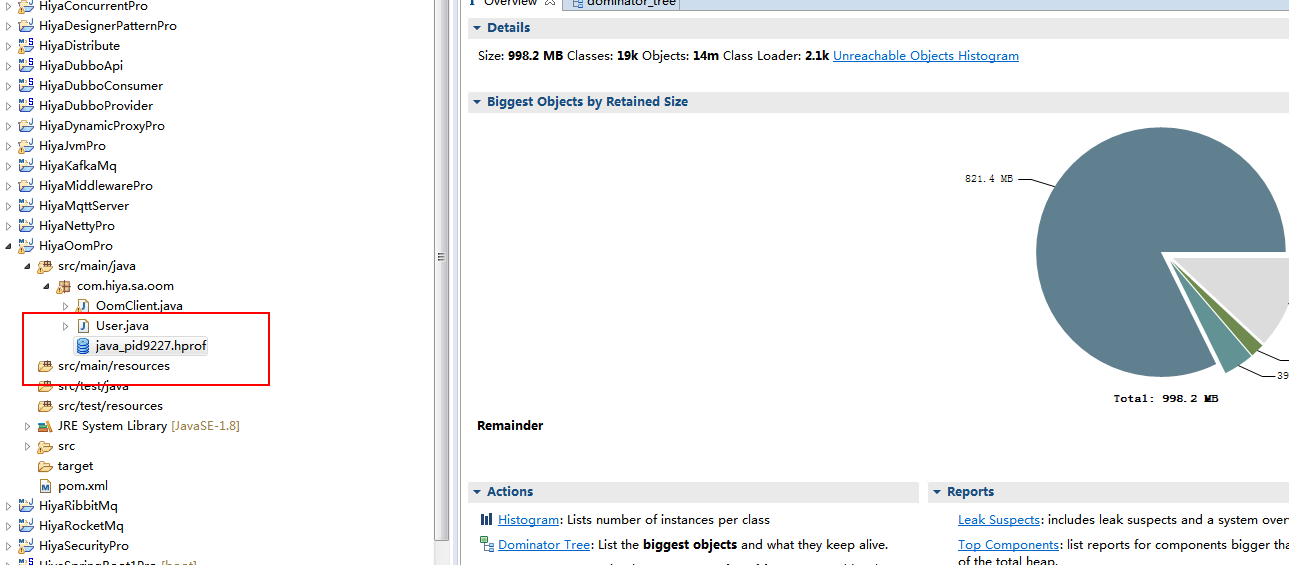
一个heap dump就是一个程序heap的快照，可以获知程序的哪些部分正在使用大部分的内存。它保存为一种叫做HPROF的二进制格式。

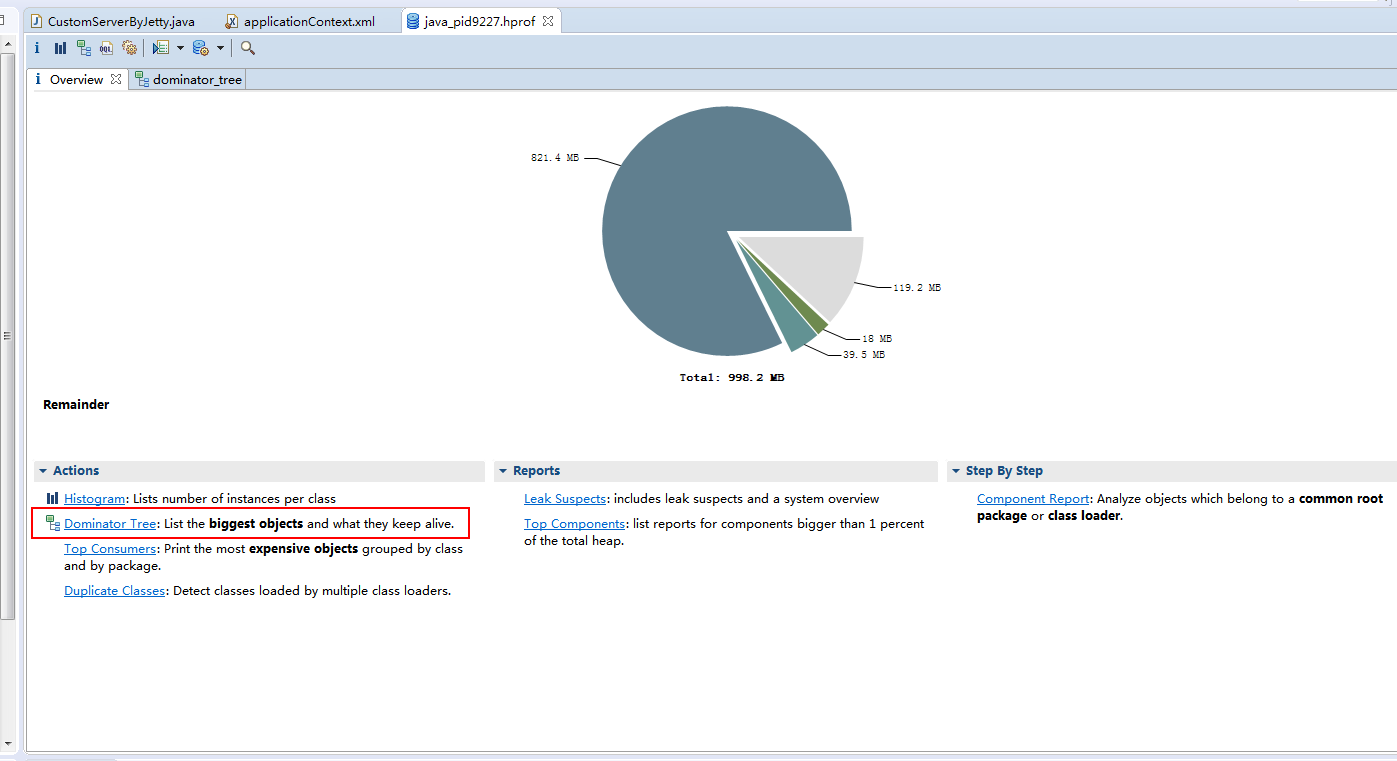


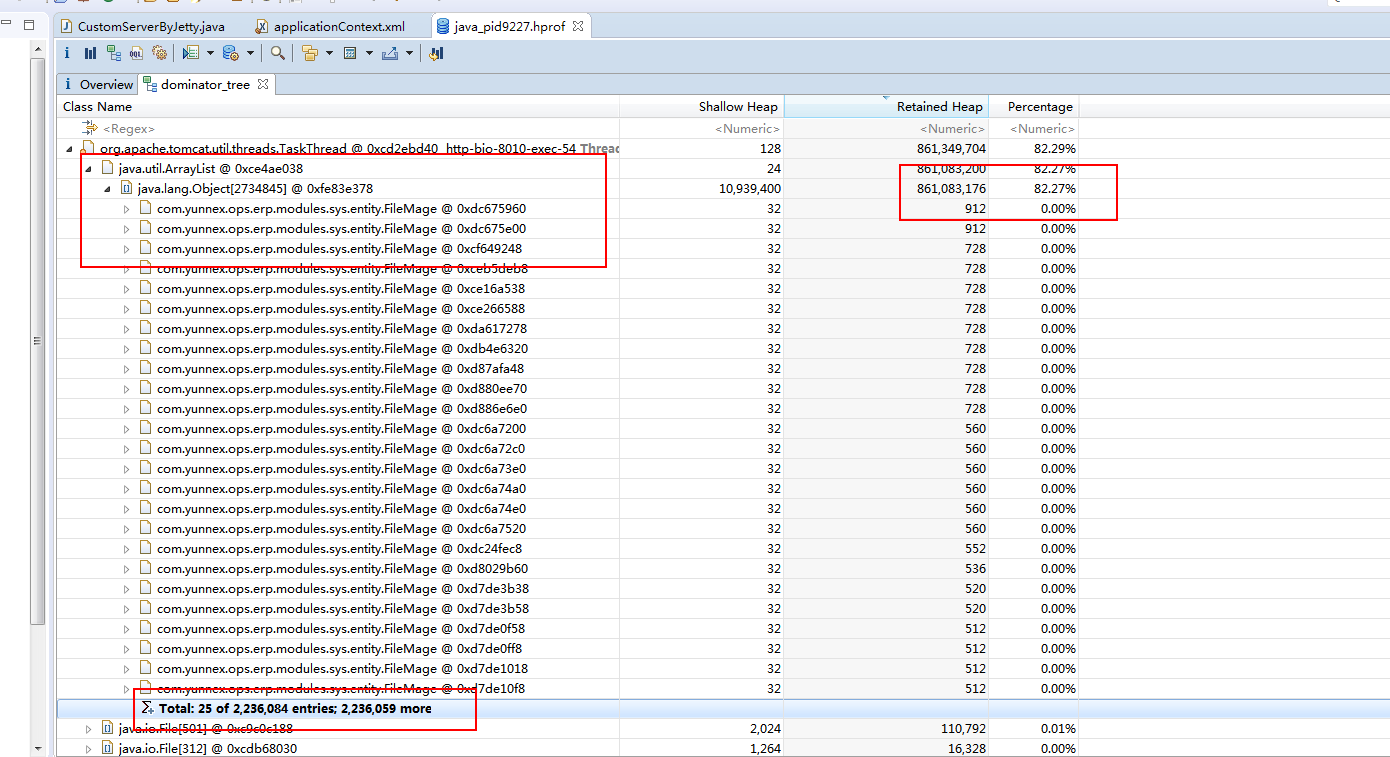


用 MAT(Memory Analyze Tool)工具--插件下载、安装及打开hprof文件见以下链接， 安装好了重启eclipse.

分析这个hprof文件（java\_pid39340.hprof），我们发现了，它的确探测出了memory leak问题，如下：







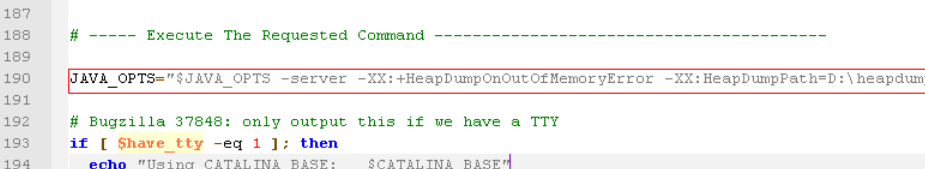
Shallow Size是对象本身占据的内存的大小，不包含其引用的对象。对于常规对象（非数组）的Shallow Size由其成员变量的数量和类型来定，而数组的ShallowSize由数组类型和数组长度来决定，它为数组元素大小的总和。

Retained Size=当前对象大小+当前对象可直接或间接引用到的对象的大小总和。(间接引用的含义：A->B->C,C就是间接引用) ，并且排除被GC Roots直接或者间接引用的对象

# **tomcat中设置jvm dump参数**

1.打开/tomcat\_home/bin/catalina.sh文件

2.加上：JAVA\_OPTS="$JAVA\_OPTS -server -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=D:\heapdump"



注：其中不设-XX:HeapDumpPath时，dump出的文件在/tomcat\_home/bin目录下

## 七、OutOfMemoryError：GC overhead limit exceeded

**1 Java运行时环境内置了垃圾收集(GC) 模块.**

上一代的很多编程语言中并没有自动内存回收机制, 需要程序员手工编写代码来进行内存分配和释放, 以重复利用堆内存。在Java程序中, 只需要关心内存分配就行。如果某块内存不再使用, 垃圾收集(Garbage Collection) 模块会自动执行清理。

**java.lang.OutOfMemoryError: GC overhead limit exceeded 这种情况发生的原因是, 程序基本上耗尽了所有的可用内存, GC也清理不了。**

**2 原因分析**

JVM抛出 java.lang.OutOfMemoryError: GC overhead limit exceeded 错误就是发出了这样的信号: 执行垃圾收集的时间比例太大, 有效的运算量太小. 默认情况下, 如果GC花费的时间超过 98%, 并且GC回收的内存少于 2%, JVM就会抛出这个错误。注意, java.lang.OutOfMemoryError: GC overhead limit exceeded 错误只在连续多次 GC 都只回收了不到2%的极端情况下才会抛出。

假如不抛出 GC overhead limit 错误会发生什么情况呢? 那就是GC清理的这么点内存很快会再次填满, 迫使GC再次执行. 这样就形成恶性循环, CPU使用率一直是100%, 而GC却没有任何成果. 系统用户就会看到系统卡死 - 以前只需要几毫秒的操作, 现在需要好几分钟才能完成。

**3 以下代码在无限循环中往 Map 里添加数据。**

这会导致 “GC overhead limit exceeded” 错误:

package com.cncounter.rtime;

import java.util.Map;

import java.util.Random;

public class TestWrapper {

public static void main(String args[]) throws Exception {

Map map = System.getProperties();

Random r = new Random();

while (true) {

map.put(r.nextInt(), "value");

}

}

}

**4 配置JVM参数: -Xmx12m。执行时产生的错误信息如下所示:**

Exception in thread "main" java.lang.OutOfMemoryError: GC overhead limit exceeded

at java.util.Hashtable.addEntry(Hashtable.java:435)

at java.util.Hashtable.put(Hashtable.java:476)

at com.cncounter.rtime.TestWrapper.main(TestWrapper.java:11)

这里在 Map 进行 rehash 时抛出了 java.lang.OutOfMemoryError: Java heap space 错误消息. 如果使用其他 垃圾收集算法, 比如 -XX:+UseConcMarkSweepGC, 或者 -XX:+UseG1GC, 错误将被默认的 exception handler 所捕获, 但是没有 stacktrace 信息, 因为在创建 Exception 时 没办法填充stacktrace信息。

例如配置:

-Xmx12m -XX:+UseG1GC

在Win7x64, Java8环境运行, 产生的错误信息为:

Exception: java.lang.OutOfMemoryError thrown from the UncaughtExceptionHandler in thread "main"

建议读者修改内存配置, 以及垃圾收集算法进行测试。

这些真实的案例表明, 在资源受限的情况下, 无法准确预测程序会死于哪种具体的原因。所以在这类错误面前, 不能绑死某种特定的错误处理顺序。

**5 解决方案**

**（1）有一种应付了事的解决方案,** 就是不想抛出 “java.lang.OutOfMemoryError: GC overhead limit exceeded” 错误信息, 则添加下面启动参数:

// 不推荐

-XX:-UseGCOverheadLimit

我们强烈建议不要指定该选项: 因为这不能真正地解决问题，只能推迟一点 out of memory 错误发生的时间，到最后还得进行其他处理。指定这个选项, 会将原来的 java.lang.OutOfMemoryError: GC overhead limit exceeded 错误掩盖，变成更常见的 java.lang.OutOfMemoryError: Java heap space 错误消息。

**（2）有时候触发 GC overhead limit 错误的原因**, 是因为分配给JVM的堆内存不足。这种情况下只需要增加堆内存大小即可。

**（3）在大多数情况下, 增加堆内存并不能解决问题。**例如程序中存在内存泄漏, 增加堆内存只能推迟产生 java.lang.OutOfMemoryError: Java heap space 错误的时间。

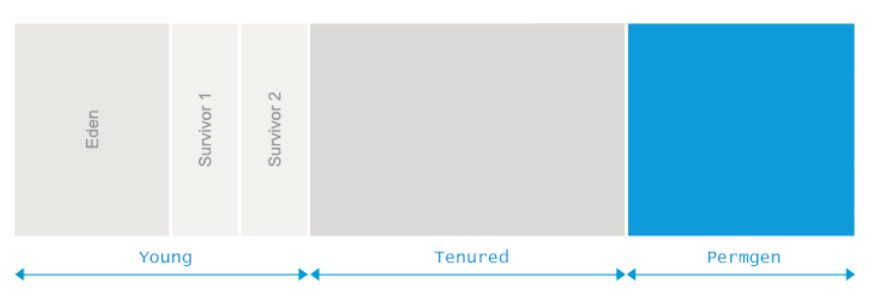
当然, 增大堆内存, 还有可能会增加 GC pauses 的时间, 从而影响程序的 吞吐量或延迟。

如果想从根本上解决问题, 则需要排查内存分配相关的代码. 简单来说, 需要回答以下问题:哪类对象占用了最多内存？这些对象是在哪部分代码中分配的。

**使用和 Java heap space一样的方法定位问题**

## OutOfMemoryError：Permgen space

**1 Java的堆内存被划分为多个区域**



**2 java.lang.outofmemoryerror: Permgen space**

这些区域的最大值, 由JVM启动参数 -Xmx 和 -XX:MaxPermSize 指定. 如果没有明确指定, 则根据操作系统平台和物理内存的大小来确定。**PermGen space 错误信息所表达的意思是: 永久代(Permanent Generation) 内存区域已满。**

**3 原因分析**

我们先看看 PermGen 是用来干什么的。

在JDK1.7及之前的版本, 永久代(permanent generation) 主要用于存储加载/缓存到内存中的 class 定义, 包括 class 的 名称(name), 字段(fields), 方法(methods)和字节码(method bytecode); 以及常量池(constant pool information); 对象数组(object arrays)/类型数组(type arrays)所关联的 class, 还有 JIT 编译器优化后的class信息等。很容易看出, PermGen 的使用量和JVM加载到内存中的 class 数量/大小有关。**可以说 java.lang.OutOfMemoryError: PermGen space 的主要原因, 是加载到内存中的 class 数量太多或体积太大。**

**4 代码演示**

import javassist.ClassPool;

public class MicroGenerator {

public static void main(String[] args) throws Exception {

for (int i = 0; i < 100\_000\_000; i++) {

generate("eu.plumbr.demo.Generated" + i);

}

}

public static Class generate(String name) throws Exception {

ClassPool pool = ClassPool.getDefault();

return pool.makeClass(name).toClass();

}

}

这段代码在 for 循环中, 动态生成了很多class。可以看到, 使用 javassist 工具类生成 class 是非常简单的。执行这段代码, 会生成很多新的 class 并将其加载到内存中, 随着生成的class越来越多,将会占满Permgen空间, 然后抛出 java.lang.OutOfMemoryError: Permgen space 错误, 当然, 也有可能会抛出其他类型的 OutOfMemoryError。要快速看到效果, 可以加上适当的JVM启动参数, 如: -Xmx200M -XX:MaxPermSize=16M 等等。

**4 解决方案**

**（1）解决程序启动时产生的 OutOfMemoryError**

在程序启动时, 如果 PermGen 耗尽而产生 OutOfMemoryError 错误, 那很容易解决. 增加 PermGen 的大小, 让程序拥有更多的内存来加载 class 即可. 修改 -XX:MaxPermSize 启动参数, 类似下面这样:

java -XX:MaxPermSize=512m

以上配置允许JVM使用的最大 PermGen 空间为 512MB, 如果还不够, 就会抛出 OutOfMemoryError。

**（2）解决 redeploy 时产生的 OutOfMemoryError**

我们可以进行堆转储分析(heap dump analysis) —— 在 redeploy 之后, 执行堆转储, 类似下面这样:

jmap -dump:format=b,file=dump.hprof <process-id>

然后通过堆转储分析器(如强悍的 Eclipse MAT)加载 dump 得到的文件。

找出重复的类, 特别是类加载器(classloader)对应的 class. 你可能需要比对所有的 classloader, 来找出当前正在使用的那个。

对于不使用的类加载器(inactive classloader), 需要先确定最短路径的 GC root , 看看是哪一个阻止其被 垃圾收集器 所回收. 这样才能找到问题的根源. 如果是第三方库的原因, 那么可以搜索 Google/StackOverflow 来查找解决方案. 如果是自己的代码问题, 则需要在恰当的时机来解除相关引用。

**（3）解决运行时产生的 OutOfMemoryError**

如果在运行的过程中发生 OutOfMemoryError, 首先需要确认 GC是否能从PermGen中卸载class。 官方的JVM在这方面是相当的保守(在加载class之后,就一直让其驻留在内存中,即使这个类不再被使用). 但是, 现代的应用程序在运行过程中, 会动态创建大量的class, 而这些class的生命周期基本上都很短暂, 旧版本的JVM 不能很好地处理这些问题。那么我们就需要允许JVM卸载class。使用下面的启动参数:

-XX:+CMSClassUnloadingEnabled

默认情况下 CMSClassUnloadingEnabled 的值为false, 所以需要明确指定。

启用以后, GC 将会清理 PermGen, 卸载无用的 class. 当然, 这个选项只有在设置 UseConcMarkSweepGC 时生效。 如果使用了 ParallelGC, 或者 Serial GC 时, 那么需要切换为CMS:

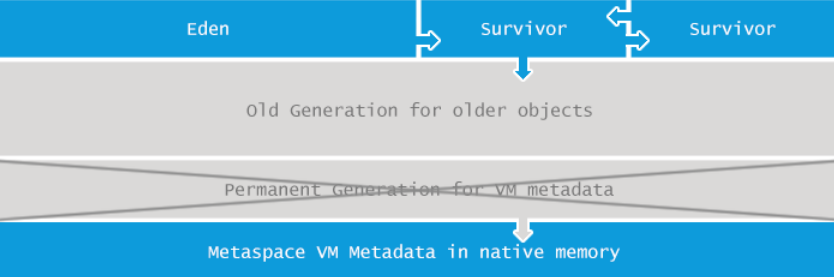
-XX:+UseConcMarkSweepGC

如果确定 class 可以被卸载, 假若还存在 OutOfMemoryError, 那就需要进行堆转储分析了, 类似下面这种命令:

jmap -dump:file=dump.hprof,format=b <process-id>

## OutOfMemoryError：Metaspace

**1 Java将堆内存划分为多个部分**



**2 java.lang.OutOfMemoryError:** **Metaspace 错误所表达的信息是: 元数据区(Metaspace) 已被用满**

**3 原因分析**

如果你是Java老司机, 应该对 PermGen 比较熟悉. 但从Java 8开始,内存结构发生重大改变, 不再使用Permgen, 而是引入一个新的空间: Metaspace. 这种改变基于多方面的考虑, 部分原因列举如下:

@Permgen空间的具体多大很难预测。指定小了会造成 java.lang.OutOfMemoryError: Permgen size 错误, 设置多了又造成浪费。

@为了 GC 性能 的提升, 使得垃圾收集过程中的并发阶段不再 停顿, 另外对 metadata 进行特定的遍历(specific iterators)。

@对 G1垃圾收集器 的并发 class unloading 进行深度优化。

在Java8中,将之前 PermGen 中的所有内容, 都移到了 Metaspace 空间。例如: class 名称, 字段, 方法, 字节码, 常量池, JIT优化代码, 等等。Metaspace 的使用量与JVM加载到内存中的 class 数量/大小有关**。可以说, java.lang.OutOfMemoryError: Metaspace 错误的主要原因, 是加载到内存中的 class 数量太多或者体积太大。**

**4 示例**

和 上一章的PermGen 类似, Metaspace 空间的使用量, 与JVM加载的 class 数量有很大关系。下面是一个简单的示例:

public class Metaspace {

static javassist.ClassPool cp = javassist.ClassPool.getDefault();

public static void main(String[] args) throws Exception{

for (int i = 0; ; i++) {

Class c = cp.makeClass("eu.plumbr.demo.Generated" + i).toClass();

}

}

}

可以看到, 使用 javassist 工具库生成 class 那是非常简单。在 for 循环中, 动态生成很多class, 最终将这些class加载到 Metaspace 中。

执行这段代码, 随着生成的class越来越多, 最后将会占满 Metaspace 空间, 抛出 java.lang.OutOfMemoryError: Metaspace. 在Mac OS X上, Java 1.8.0\_05 环境下, 如果设置了启动参数 -XX:MaxMetaspaceSize=64m, 大约加载 70000 个class后JVM就会挂掉。

**5 解决方案**

如果抛出与 Metaspace 有关的 OutOfMemoryError , 第一解决方案是增加 Metaspace 的大小. 使用下面这样的启动参数

-XX:MaxMetaspaceSize=512m

这里将 Metaspace 的最大值设置为 512MB, 如果没有用完, 就不会抛出 OutOfMemoryError。

## 十、常用诊断工具

**1 jmap查看内存使用情况与生成heapdump**

**（1）查看内存使用**

**ps -ef|grep java**

**jmap -heap 1234**

[-heap：打印heap空间的概要检验heap空间的使用情况。]

Attaching to process ID 2657, please wait...

Debugger attached successfully.

Client compiler detected.

JVM version is 1.5.0\_16-b02

using thread-local object allocation.

Mark Sweep Compact GC

Heap Configuration:

MinHeapFreeRatio = 40

// jvm heap 在使用率小于 n 的情况下 ,heap 进行收缩

MaxHeapFreeRatio = 70

// jvm heap 在使用率大于 n 的情况下 ,heap 进行扩张

MaxHeapSize = 67108864 (64.0MB)

//对应jvm启动参数-XX:MaxHeapSize=设置JVM堆的最大大小

NewSize = 655360 (0.625MB)

//对应jvm启动参数-XX:NewSize=设置JVM堆的‘新生代’的默认大小

MaxNewSize = 4294901760 (4095.9375MB)

//对应jvm启动参数-XX:MaxNewSize=设置JVM堆的‘新生代’的最大大小

OldSize = 1441792 (1.375MB)

//对应jvm启动参数-XX:OldSize=<value>:设置JVM堆的‘老生代’的大小

NewRatio = 12

//jvm参数-XX:NewRatio=, 新生代（Eden + 2\*S）与老年代（不包括永久区）的比值,4表示新生代老年代=14,意思是老年代占4/5

SurvivorRatio = 8

//2个Survivor区和Eden区的比值,8表示两个SurvivorEden=28,每个Survivor占 1/10

PermSize = 8388608 (8.0MB)

//对应jvm启动参数-XX:PermSize=<value>:设置JVM堆的‘永生代’的初始大小

MaxPermSize = 67108864 (64.0MB)

//对应jvm启动参数-XX:MaxPermSize= :设置JVM堆的‘永生代’的最大大小

Heap Usage:

New Generation (Eden + 1 Survivor Space):------------------------新生代区

capacity = 4521984 (4.3125MB) //容量

used = 1510200 (1.4402389526367188MB)

free = 3011784 (2.8722610473632812MB)

33.39684527853261% used //使用量

Eden Space:-------------------------------------伊甸园区

capacity = 4063232 (3.875MB)

used = 1495992 (1.4266891479492188MB)

free = 2567240 (2.4483108520507812MB)

36.81778446320565% used //使用量

From Space:-----------------------------------------年轻代（幸存者乐园1）

capacity = 458752 (0.4375MB)

used = 14208 (0.0135498046875MB)

free = 444544 (0.4239501953125MB)

3.0970982142857144% used //使用量

To Space:--------------------------------------------------年轻代（幸存者乐园2）

capacity = 458752 (0.4375MB)

used = 0 (0.0MB)

free = 458752 (0.4375MB)

0.0% used //使用量

concurrent mark-sweep generation:------------------------------老年代

capacity = 8589934592 (8192.0MB)

used = 0 (0.0MB)

free = 8589934592 (8192.0MB)

0.0% used

Perm Generation:--------------------------------------------------永久代

capacity = 11796480 (11.25MB)

used = 11712040 (11.169471740722656MB)

free = 84440 (0.08052825927734375MB)

99.28419325086806% used

以上的输出很简单，第四行起开始输出此进程我们的JAVA使用的环境。

Heap Configuration，指在我们启动时设置的一些JVM参数。像最大使用内存大小，年老代，年青代，持久代大小等。有这个可以很简单的查看本进程的内存使用情况。也许进程占用的总内存比较多，但我们在这里可以看到真正用到的并没有多少，很多都是"Free"。内存使用的堆积大多在老年代，内存池露始于此，所以要格外关心“tenured generation”。

**（2）产生一个HeapDump文件**

JVM Memory Map命令用于生成heap dump文件，如果不使用这个命令，还可以使用**-XX:+HeapDumpOnOutOfMemoryError**参数来让虚拟机出现OOM的时候自动生成dump文件。

dump堆到文件,format指定输出格式，live指明是活着的对象,file指定文件名

[root@localhost jdk1.7.0\_79]# **jmap -dump:live,format=b,file=dump.hprof 24971**

Dumping heap to /usr/local/java/jdk1.7.0\_79/dump.hprof ...

Heap dump file created

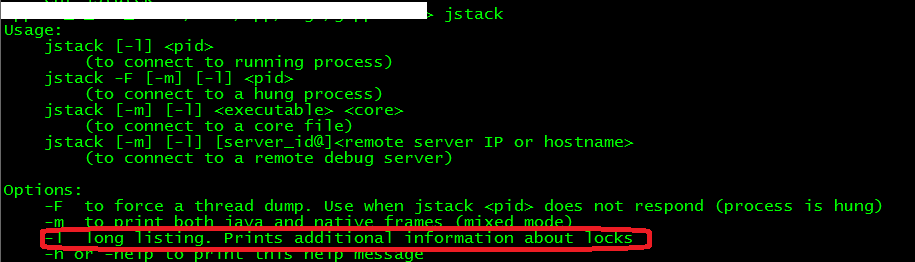
**2 Jstack是JDK自带的命令行工具，主要用于线程Dump分析。**

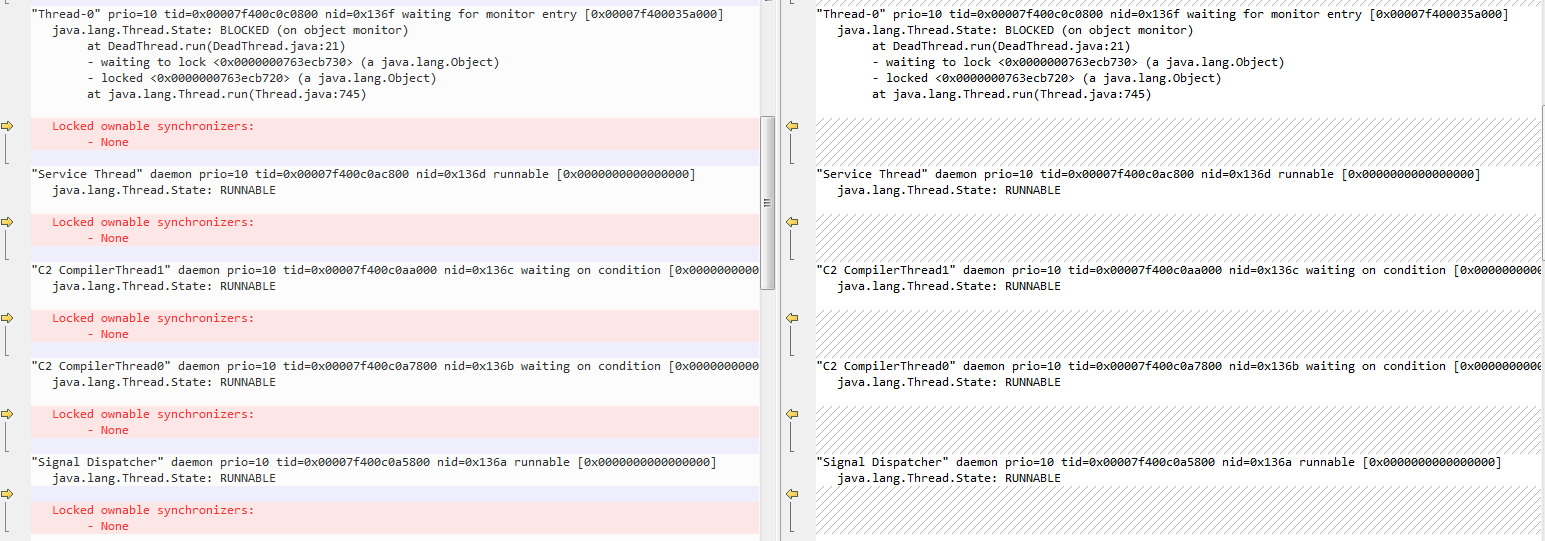
**（1）找出java进程号**

ps -ef|grep java

**（2）jstack输出线程dump信息到文件**

jstack -l 4955 > jstack.log





**3 jstat 查看classloader，compiler，gc相关信息，实时监控资源和性能 。**

jstat工具特别强大，可以用来监视VM内存内的各种堆和非堆的大小及其内存使用量。

语法结构：jstat -<option> [-t] [-h<lines>] <pid> [<interval> [<count>]]

Options — 通常使用 -gcutil 查看gc情况

interval – 间隔时间，单位为秒或者毫秒

count — 打印次数，如果缺省则打印无数次

结果说明：

S0 — Heap上的 Survivor space 0 区已使用空间的百分比

S1 — Heap上的 Survivor space 1 区已使用空间的百分比

E — Heap上的 Eden space 区已使用空间的百分比

O — Heap上的 Old space 区已使用空间的百分比

P — Perm space 区已使用空间的百分比

YGC — 从应用程序启动到采样时发生 Young GC 的次数

YGCT— 从应用程序启动到采样时 Young GC 所用的时间(单位秒)

FGC — 从应用程序启动到采样时发生 Full GC 的次数

FGCT— 从应用程序启动到采样时 Full GC 所用的时间(单位秒)

GCT — 从应用程序启动到采样时用于垃圾回收的总时间(单位秒)

jstat –class <pid> : 显示加载class的数量，及所占空间等信息。

jstat -compiler <pid>: 显示VM实时编译的数量等信息。

jstat -gc <pid>: 显示gc的信息，查看gc的次数，及时间。

jstat -gccapacity <pid>: 显示VM内存中三代（young,old,perm）对象的使用和占用大小

jstat -gcutil <pid>: 统计gc信息

jstat -gcnew / gcnewcapacity<pid>: 年轻代对象的信息(及其占用量)。

jstat -gcold / gcoldcapacity <pid> ：old代对象的信息(及其占用量)。

jstat -gcpermcapacity <pid>: perm对象的信息及其占用量。

jstat -printcompilation <pid>: 当前VM执行的信息。

