最近在工作中经常和性能压测工作打交道，积累了一些性能分析经验，我觉得这些经验对每一个开发者都有帮助的，能开发出性能高的代码也是我们的最终目标。

由易到难，我们逐步介绍不同命令的用法和好处，这些命令是如何帮助我们开发人员进行性能分析的。

一、开发者的自测利器-Hprof命令

**1. 示例演示**

例子程序：

/\*\*

\* PROJECT\_NAME: test

\* DATE: 16/7/22

\* CREATE BY: chao.cheng

\*\*/

public class HProfTest {

public void slowMethod() {

try {

Thread.sleep(1000);

} catch (Exception e) {

e.printStackTrace();

}

}

public void slowerMethod() {

try {

Thread.sleep(10000);

} catch (Exception e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

HProfTest test = new HProfTest();

test.slowerMethod();

test.slowMethod();

}

}

**注**：这是一段测试代码通过sleep方法进行延时，在程序运行过程中很慢，我想知道到底是哪段程序影响的整体性能呢？

我在这个java程序中，加了如下运行参数：

-agentlib:hprof=cpu=times,interval=10

/\*

times：java函数的执行时间

hprof=cpu是针对cpu统计时间

interval=10 采样10次

\*/

再次运行这段程序显示如下图：

图片描述

这时候还发现在工程目录里面，多了一个文本文件java.hprof.txt，如下图所示：

图片描述

内容如下：

CPU TIME (ms) BEGIN (total = 11542) Fri Jul 22 11:00:34 2016

rank self accum count trace method

1 86.65% 86.65% 1 303422 com.test.HProfTest.slowerMethod

2 8.66% 95.31% 1 303423 com.test.HProfTest.slowMethod

3 0.25% 95.56% 36 300745 java.util.zip.ZipFile.<init>

4 0.20% 95.76% 36 300434 java.lang.String.equals

5 0.13% 95.89% 14 301138 java.net.URLStreamHandler.parseURL

6 0.11% 96.01% 6 301339 java.net.URLClassLoader$1.run

7 0.10% 96.10% 14 301124 java.lang.String.<init>

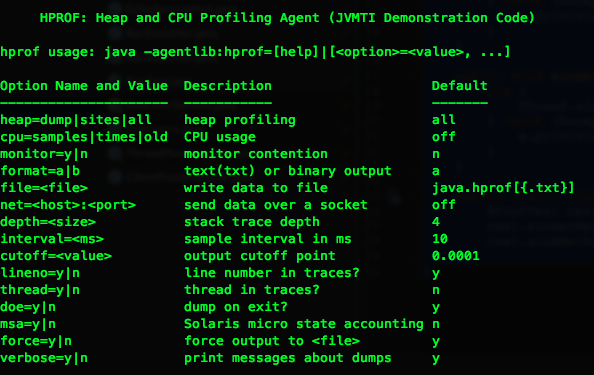
8 0.09% 96.19% 3407 300355 java.lang.String.charAt

9 0.08% 96.27% 36 300443 java.io.UnixFileSystem.normalize

**注**：通过上面内容可以看到，哪个类的方法执行时间长，耗费了CPU时间，一目了然，方便我们快速定位问题。

**2. 命令的具体讲解**

hprof不是独立的监控工具，它只是一个java agent工具，它可以用在监控Java应用程序在运行时的CPU信息和堆内容，使用java -agentlib:hprof=help命令可以查看hprof的使用文档。



通过上图可以看到这个工具非常强大，可以统计的东西很多，上面的例子统计的是CPU时间，同样我们还可以统计内存占用的dump信息。   
如：-agentlib:hprof=heap,format=b,file=/test.hprof

这个hprof小工具，非常方便我们在用JUnit自测代码的时候结合使用，既可以解决业务上的BUG，又能够在一定程序上解决可发现的性能问题，非常实用。

二、性能排查工具-pidstat

**1. 示例演示**

例子程序：

/\*\*

\* PROJECT\_NAME: test

\* DATE: 16/7/22

\* CREATE BY: chao.cheng

\*\*/

public class PidstatTest {

public static class PidstatTask implements Runnable {

public void run() {

while(true) {

double value = Math.random() \* Math.random();

}

}

}

public static class LazyTask implements Runnable {

public void run() {

try {

while (true) {

Thread.sleep(1000);

}

} catch (Exception e) {

e.printStackTrace();

}

}

}

public static void main(String[] args) {

new Thread(new PidstatTask()).start();

new Thread(new LazyTask()).start();

new Thread(new LazyTask()).start();

}

}

**注**：这是一段测试用的java程序，将其运行起来。

在命令行输入：

pidstat -p 843 1 3 -u -t

/\*

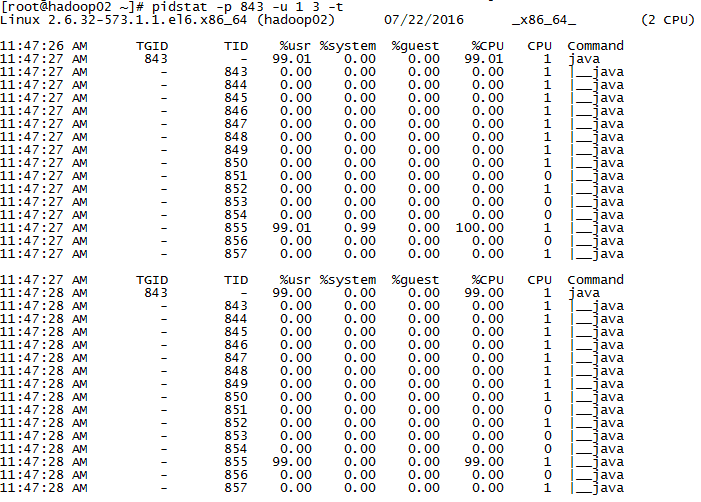
-u：代表对cpu使用率的监控

参数1 3：表示每秒采样一次，一共三次

-t：将监控级别细化到线程

\*/

运行命令显示如下图所示：

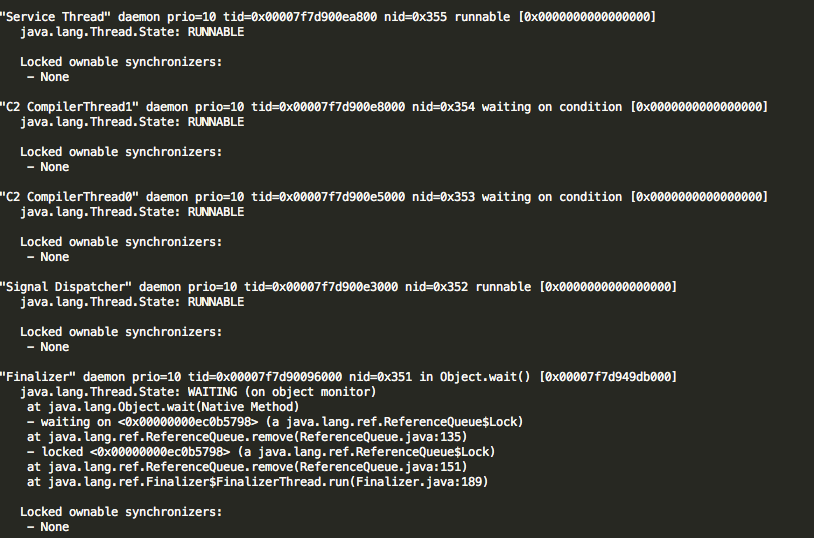


**注**：其实中TID就是线程ID，%usr表示用户线程使用率，从图中可以看到855这个线程CPU占用非常高。

再输入如下命令：

jstack -l 843 > /tmp/testlog.txt

查看testlog.txt显示如下部分内容：



**注**：我们关注的是日志文件的NID字段，它对应的就是我们上面说的TID，NID是TID的16进制表示，将上面的十进制855转换成十六进制为357，在日志中进行搜索看到如下内容：

"Thread-0" prio=10 tid=0x00007f7d90103800 nid=0x357 runnable [0x00007f7d943d5000]

java.lang.Thread.State: RUNNABLE

at PidstatTest$PidstatTask.run(PidstatTest.java:13)

at java.lang.Thread.run(Thread.java:722)

Locked ownable synchronizers:

- None

以此可以推断出有性能瓶颈的程序点。

**2. pidstat具体命令详解**

pidstat是一个功能非常强大的性能监测工具，他是Sysstat的组件之一，可以从<http://sebastien.godard.pagesperso-orange.fr/download.html> 进行下载，下载后可以通过./configure等命令进行安装，这个命令的强大之处在于不仅可以监控进程的性能情况，也可以监控线程的性能情况。

pidstat监控CPU常用显示字段内容如下：

1、PID - 被监控的任务的进程号

2、%usr - 当在用户层执行（应用程序）时这个任务的cpu使用率，和 nice 优先级无关。注意这个字段计算的cpu时间不包括在虚拟处理器中花去的时间。

3、%system - 这个任务在系统层使用时的cpu使用率。

4、%guest - 任务花费在虚拟机上的cpu使用率（运行在虚拟处理器）。

5、%CPU - 任务总的cpu使用率。在SMP环境（多处理器）中，如果在命令行中输入-I参数的话，cpu使用率会除以你的cpu数量。

6、CPU - 正在运行这个任务的处理器编号。

7、Command - 这个任务的命令名称。

pidstat监控IO常用的字段显示内容如下：

1、kB\_rd/s - 任务从硬盘上的读取速度（kb）

2、kB\_wr/s - 任务向硬盘中的写入速度（kb）

3、kB\_ccwr/s - 任务写入磁盘被取消的速率（kb）

三、一个内存溢出案例分析

**1. 内存溢出现象**

系统共有8台服务器，每次随机只有一台服务器报java.lang.OutOfMemoryError: GC overhead limit exceeded错误，然后接着就报内存溢出错误java.lang.OutOfMemoryError: Java heap space。

**2. 理论支撑**

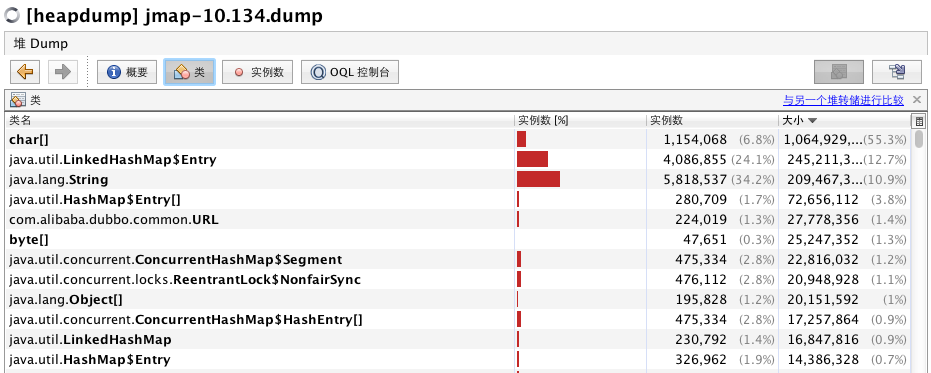
我们先解释一下什么是GC overhead limit exceeded错误。

GC overhead limt exceed检查是Hotspot VM 1.6定义的一个策略，通过统计GC时间来预测是否要OOM了，提前抛出异常，防止OOM发生。Sun 官方对此的定义是：“并行/并发回收器在GC回收时间过长时会抛出OutOfMemroyError。过长的定义是，超过98%的时间用来做GC并且回收了不到2%的堆内存。用来避免内存过小造成应用不能正常工作。

可以看到当堆中的对象无法被收回的时候，就提前遇警报出这样的错误，此时内存并没有溢出，这个特性在JDK中是默认添加的。

**3. DUMP文件分析**

将dump文件导入VisualVM工具中，如下图所示：



通过上图可以看出类结构图中，最占用内存的是char[]，LinkedHashMap和String三项。但是这三项的实例数并没有占满，看样子不会内存溢出，怎么才能具体分析呢？原因就在于GC overhead limt exceed，这个错并不会在内存真正溢出才会报，所以通过dump文件，我们只能自己去判断分析，哪些项有可能会造成溢出，我们进入char[]项具体来看，会发现里面有很多hessian的url字符被缓存，通过排除程序可以看到由于底层中间件程序为了提高“性能”，将每次调用的url都缓存起来，不用每次都生成，但没有相应缓存释放操作，于是造成了大量字符对象长期持有从而报错，在此就不截图来具体看代码，涉及一些公司信息。

**4. 问题解决方案**

* 可以添加JVM的启动参数来去掉提前报警限制：-XX:-UseGCOverheadLimit，于其让应用每次都提前报警，还不如让暴风雨来的更猛些，直接内存溢出，因为服务器是集群，其中一台挂掉不会影响线上正常交易，同时也方便我们通过日志来排错。
* 通过排查程序，检查系统是否有使用大内存的代码不释放或死循环。