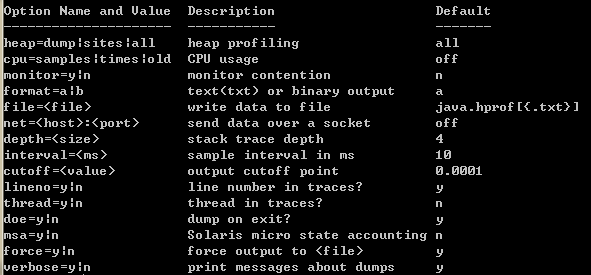
## HPROF（[A Heap/CPU Profiling Tool](http://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html)）

HPROF profiler 它可以统计CPU使用状况，堆内存分配情况，以及所有的锁和线程。HPROF在分析性能、锁、内存泄露等非常有用。

The Heap Profiler (HPROF) tool 是一个随JDK一同发布的简单剖析工具。

使用以下命令能得到提示。

java -agentlib:hprof=help



根据请求命令的类型， HPROF指示虚拟机发送给它相关的事件， 然后该工具处理这些事  
件，形成剖析信息。

hprof 命令格式为：java -agentlib:hprof[=options] ToBeProfiledClass

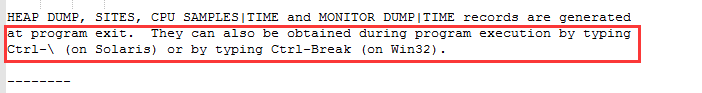
或javac -J-agentlib:hprof[=options] ToBeProfiledClass.java

说明：测试的时候用java老是失败，原因不详。用javac的可成功。

除了输入命令之外。还可以在启动java程序前加入参数，java程序运行完毕后会生成一个文件夹。如下图：



当添加这个vm参数后，在线上运行时，linux上通过kill -3 pid命令即可打印出消息。默认运行路径下的java.hprof.txt。

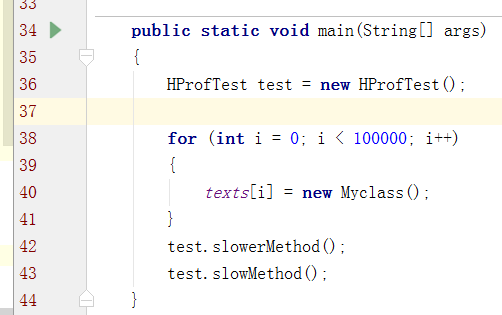
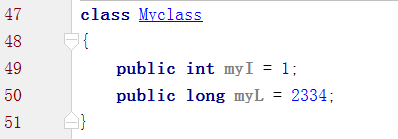


注意：javac -J-agentlib:hprof 生成的文件是由Java编译器(Javac)在编译一系列源代码文件时产生的信息。

直接在jvm配置参数运行的生成的信息更详细，以下例子用这种方式设置。

下面是一些使用的例子。

部分测试代码：

堆分配点剖析**(heap=sites)** 下面的输出是由Java编译器(Javac)在编译一系列源代码文件时  
产生的堆分配信息， 这里仅列出一部分：

-agentlib:hprof=heap=sites

SITES BEGIN (ordered by live bytes) Fri Apr 21 15:52:50 2017

percent live alloc'ed stack class

rank self accum bytes objs bytes objs trace name

1 74.46% 74.46% 2400000 100000 2400000 100000 300618 btrace.Myclass

2 12.43% 86.89% 400608 2 400608 2 300616 btrace.Myclass[]

3 0.19% 87.08% 6008 27 6008 27 300053 char[]

在堆分配剖析文件里，最关键的信息是程序每一部分分配的对象的数量。上面显示74.46%的SITES记录表示Myclass对象占用了总空间的74.46%。

关联源代码和分配点的最好方式是记录导致内存分配的线程堆栈。 下面的是剖析输出的  
另外一部分信息， 下面四个分配点说明了是由哪个调用堆栈产生的：

TRACE 300618:

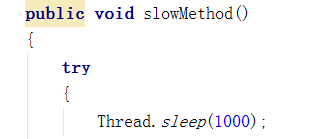
btrace.Myclass.<init>(HProfTest.java:47)

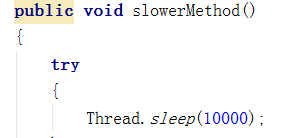
btrace.HProfTest.main(HProfTest.java:40)

**CPU**使用时间剖析**(cpu=times)** HPROF工具可以通过在每个方法进入和退出时注入代码来收集CPU的使用情况，因此可以知道每个方法的调用次数和运行的时间。这种方法叫做字节码注入(Byte Code Injection-BCI)，运行起来比cpu=samples要慢一些，下面是运行javac编译器获得的部分信息：

-agentlib:hprof=cpu=times

部分源码如下：





结果如下：

CPU TIME (ms) BEGIN (total = 11349) Fri Apr 21 16:16:53 2017

rank self accum count trace method

1 88.12% 88.12% 1 302122 btrace.HProfTest.slowerMethod

2 8.81% 96.93% 1 302123 btrace.HProfTest.slowMethod

3 1.07% 98.01% 1 302124 btrace.HProfTest.main

可看出88.12%的CPU时间在执行slowerMethod方法。

**HPROF Profiler** 正在运行的程序可以通过HPROF剖析器代理创建堆转储， 例如：  
$ java -agentlib:hprof=file=snapshot.hprof,format=b application

堆转储**(heap=dump)** 堆转储是通过heap=dump选项获得的，该输出文件可以是ASCII也可以是二进制，取决于format选项的设置。如果这些输出文件要被jhat工具分析，可以通过format=b指定输出格式为二进制。

当指定了二进制格式，转储文件包括原子类型字段和原子数组内容。

下面的转储片段是由javac编译器产生的：  
$ javac -J-agentlib:hprof=heap=dump Hello.java

-agentlib:hprof=heap=dump 转储的文件能够被堆分析工具使用。比如 ibm heapAnalyzer。

下面是一个例子：  
HEAP DUMP BEGIN (39793 objects, 2628264 bytes) Wed Oct 4 13:54:03 2006  
ROOT 50000114 (kind=<thread>, id=200002, trace=300000)  
ROOT 50000006 (kind=<JNI global ref>, id=8, trace=300000)  
ROOT 50008c6f (kind=<Java stack>, thread=200000, frame=5)  
:  
CLS 50000006 (name=java.lang.annotation.Annotation, trace=300000)  
loader 90000001  
OBJ 50000114 (sz=96, trace=300001, class=java.lang.Thread@50000106)  
name 50000116  
group 50008c6c  
contextClassLoader 50008c53  
inheritedAccessControlContext 50008c79  
blockerLock 50000115  
OBJ 50008c6c (sz=48, trace=300000, class=java.lang.ThreadGroup@50000068)  
name 50008c7d  
threads 50008c7c  
groups 50008c7b  
ARR 50008c6f (sz=16, trace=300000, nelems=1,  
elem type=java.lang.String[]@5000008e)  
[0] 500007a5  
CLS 5000008e (name=java.lang.String[], trace=300000)  
super 50000012  
loader 90000001  
:  
HEAP DUMP END  
每一个记录是一个根(Root). OBJ表示对象实例， CLS表示class，ARR表示数组。16进制数字是由HPROF分配的标识符，这些数字用来表示从一个对象到另一个对象的引用。例如，在上面的例子里java.lang.Thread实例50000114有一个到它的线程组(thread group:50008c6c)和另一个对象的引用

**CPU**使用率采样剖析**(cpu=samples)** HPROF工具可以通过对线程进行周期采样收集CPU的使用信息，下面是从运行javac编译器获得的采样信息的部分输出：

$ javac -J-agentlib:hprof=cpu=samples Hello.java

## jstack工具

主要用来打印进程堆栈。例子如下：

jstack 1636 > dump.tdump 打印pid为1636的进程堆栈到dump.tdump文件里

除了打印进程堆栈，还可以进程里具体线程堆栈。格式有下面2种。pid是进程id，tid是线程id

jstack pid |grep tid -A n

jstack pid grep -A n tid

具体例子如：jstack 3132 grep -A 10 7a91(线程16进制tid) 10 表示堆栈的深度。

## jmap工具

jmap（Memory Map for Java，内存映像工具），通过jmap工具同样可以获得堆转储，下面是一个例子：

$ jmap -dump:format=b,file=snapshot.jmap pid

option参数说明：

* -dump:[live,]format=b,file=<filename> 使用hprof二进制形式,输出jvm的heap内容到文件=<filename>，live子选项是可选的，假如指定live选项,那么只输出活的对象到文件.
* -finalizerinfo 打印正等候回收的对象的信息.
* -heap 打印heap的概要信息，GC使用的[算法](http://lib.csdn.net/base/datastructure)，heap的配置及wise heap的使用情况.
* -histo[:live] 打印每个class的实例数目,内存占用,类全名信息. VM的内部类名字开头会加上前缀”\*”. 如果live子参数加上后,只统计活的对象数量.
* -permstat 打印classload和jvm heap长久层的信息. 包含每个classloader的名字,活泼性,地址,父classloader和加载的class数量. 另外,内部String的数量和占用内存数也会打印出来.
* -F 强迫.在pid没有相应的时候使用-dump或者-histo参数. 在这个模式下,live子参数无效.
* -h | -help 打印辅助信息
* -J 传递参数给jmap启动的jvm.

64位机上使用需要使用如下方式：

jmap -J-d64 -heap pid

jmap输出的堆信息可用于对分析工具分析，jdk自带提供了*jhat* 工具，不过不方便，可以用其他分析工具，如ibm heapAnalyzer。

## jhat工具

用于查看jmap生成的文件。使用如下：

jhat -J-Xmx512m -port 9998 /tmp/dump.dat。

/tmp/dump.dat 为文件的路径。之后访问127.0.0.1:9998即可

## jps工具

jps（JVM Process Status Tool，虚拟机进程监控工具），这个命令可以列出正在运行的虚拟机进程，并显示虚拟机执行主类名称，以及这些进程的本地虚拟机唯一ID。

jps [options] [hostid]

options参数选项说明如下：

-q 不输出类名、Jar名和传入main方法的参数

-m 输出传入main方法的参数

-l 输出main类或Jar的全限名

-v 输出传入JVM的参数

## jinfo工具

jinfo （Configuration Info for Java，配置信息工具）这个命令可以实时地查看和调整虚拟机各项参数。

查看进程id为2788所有虚拟机的信息

jinfo 2788

或查看进程id为2788的MaxPerm大小可以用

jinfo -flag MaxPermSize 2788

jinfo -flag [+/-][flagName] [pid] #启用/禁止某个参数

jinfo -flag [flagName=value] [pid] #设置某个参数

对于上述的gc的情况，就可以使用以下命令打开heap dump并设置dump路径。

jinfo -flag +HeapDumpBeforeFullGC [pid]

jinfo -flag +HeapDumpAfterFullGC [pid]

jinfo -flag HeapDumpPath=/home/dump/dir [pid]

同样的也可以动态关闭。

jinfo -flag -HeapDumpBeforeFullGC [pid]

jinfo -flag -HeapDumpAfterFullGC [pid]

注意：动态修改了的值通过jinfo pid 打印是看不到的。还有不是什么参数都可以修改。

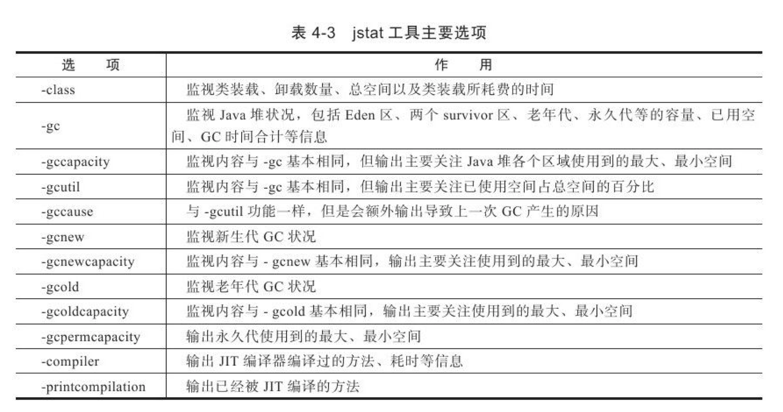
## jstat工具

jstat（JVM Statistics Monitoring Tool，虚拟机统计信息监视工具），这个命令用于监视虚拟机各种运行状态信息。它可以显示本地或者远程虚拟机进程中的类装载、内存、垃圾收集、JIT编译等运行数据，虽然没有GUI图形界面，只是提供了纯文本控制台环境的服务器上，但它是运行期间定位虚拟机性能问题的首选工具。主要是通过监控内存回收方面来定位问题

语法格式如下：

jstat [option vmid [interval [s | ms] [count ] ] ]

jstat的参数说明如下

• class - 打印class loader的统计或者状态信息。  
• compiler - 打印HotSpot compiler的统计信息.  
• gc - 打印堆内存回收的统计信息.  
• gccapacity - 打印代(generations)的容量统计信息等。  
• gccause - 打印垃圾回收的总结信息(同-gcutil), 已经最后和当前垃圾回收事件的原因(如果  
有的话) .  
• gcnew - 打印新生代的统计信息.  
• gcnewcapacity - 打印新生代大小， 空间等统计信息.  
• gcold - 打印老生代的统计信息.  
• gcoldcapacity - 打印老生代大小， 空间等统计信息.  
• gcpermcapacity - 打印持久代的大小统计信息.  
• gcutil - 打印垃圾回收的统计信息  
• printcompilation - 打印HotSpot compilation method 统计信息.

**Example of -gcutil Option** 下面是一个采用-gcutil 选项的例子，该工具绑定到进程id为21942的进程上，3000毫秒采样9次。

$ jstat -gcutil 21942 3000 9

S0 S1 E O P YGC YGCT FGC FGCT GCT

62.05 0.00 77.98 85.24 49.91 1852 15.179 1 0.234 15.413

62.05 0.00 79.52 85.24 49.91 1852 15.179 1 0.234 15.413

62.05 0.00 81.77 85.24 49.91 1852 15.179 1 0.234 15.413

62.05 0.00 84.08 85.24 49.91 1852 15.179 1 0.234 15.413

62.05 0.00 85.61 85.24 49.91 1852 15.179 1 0.234 15.413

62.05 0.00 96.37 85.24 49.91 1852 15.179 1 0.234 15.413

62.05 0.00 97.99 85.24 49.91 1852 15.179 1 0.234 15.413

0.00 27.90 1.56 85.27 49.91 1853 15.189 1 0.234 15.424

0.00 27.90 3.10 85.27 49.91 1853 15.189 1 0.234 15.424

这个输出显示新生代的回收发生在第七第八次采用之间， 回收用了0.010秒，对象搬移的顺序为E >S0>S1>O.

  现在来解释各列含义：

S0、S1：Survivor 0/1和使用量

E：Eden使用量

O：年老代使用量

P：永久代使用量

YGC、YGCT：年轻代GC次数和GC耗时

FGC、FGCT：Full GC次数和Full GC耗时

GCT：GC总耗时

## pidstat工具

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

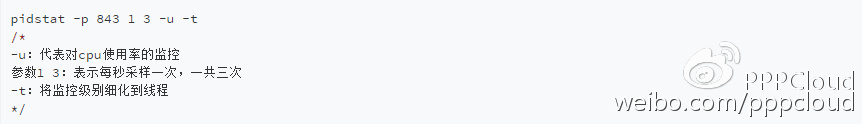
<http://sebastien.godard.pagesperso-orange.fr/download.html> 进行下载

利用pidstat找出cpu消耗异常。

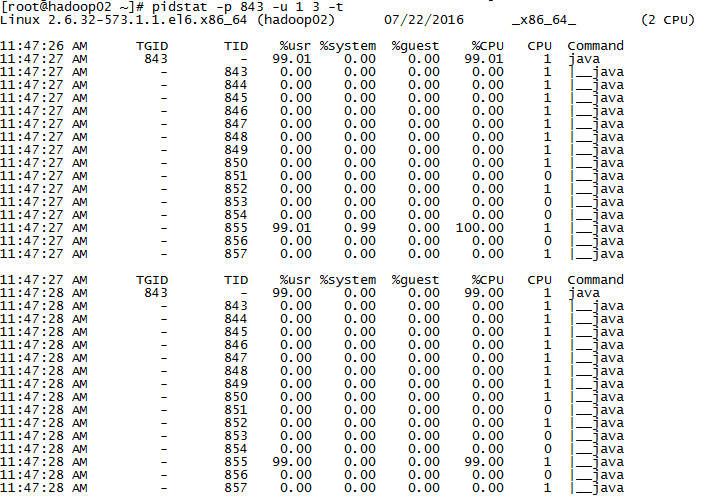
这是一段测试用的java程序，将其运行起来。



在命令行输入：



运行命令显示如下图所示：

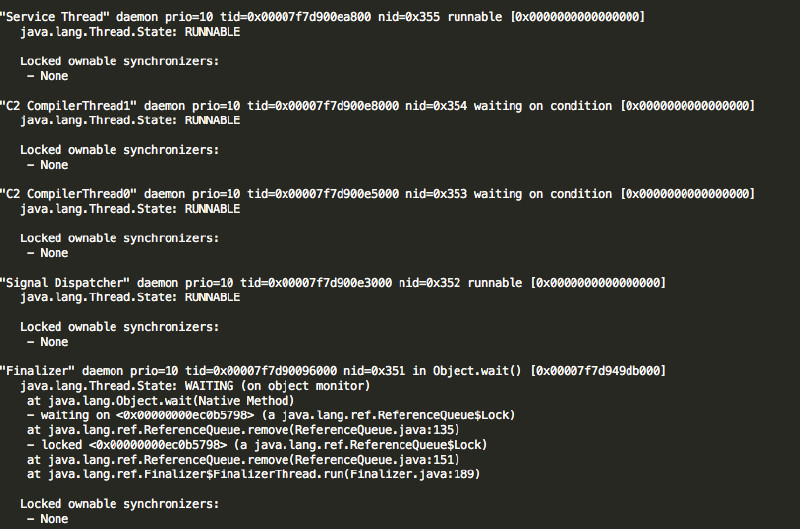
图片描述

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。

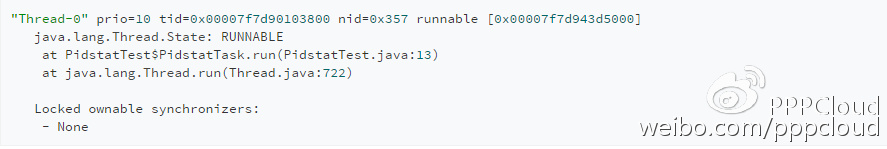
再输入如下命令：

http://ww1.sinaimg.cn/large/0068cbuBgw1f660n8unepj308s00swee.jpg

查看testlog.txt显示如下部分内容：

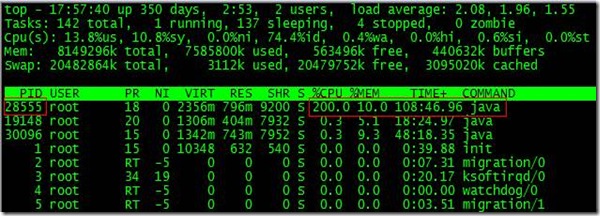
图片描述

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：



一个应用占用CPU很高，除了确实是计算密集型应用之外，通常原因都是出现了死循环。

另一个找出cpu高消耗的方法。（可参考：[**http://www.blogjava.net/hankchen**](http://www.blogjava.net/hankchen)**）**

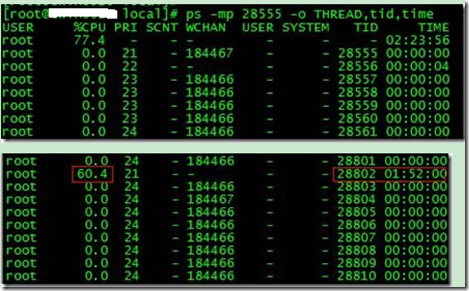
[](http://www.blogjava.net/images/blogjava_net/hankchen/WindowsLiveWriter/CPU_121DA/clip_image002_2.jpg)

根据top命令，发现PID为28555的Java进程占用CPU高达200%，出现故障。

通过ps aux | grep PID命令，可以进一步确定是tomcat进程出现了问题。但是，怎么定位到具体线程或者代码呢？

**首先显示线程列表:**

ps -mp pid -o THREAD,tid,time

[](http://www.blogjava.net/images/blogjava_net/hankchen/WindowsLiveWriter/CPU_121DA/1.png)

找到了耗时最高的线程28802，占用CPU时间快两个小时了！

**其次将需要的线程ID转换为16进制格式：**

printf "%x\n" tid

[2](http://www.blogjava.net/images/blogjava_net/hankchen/WindowsLiveWriter/CPU_121DA/2.png)

**最后打印线程的堆栈信息**

jstack pid |grep tid -A 30

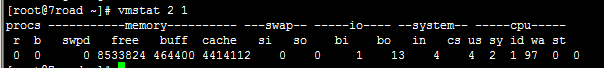
## vmstat工具

此工具是查看liunx状态的。

一般vmstat工具的使用是通过两个数字参数来完成的，第一个参数是采样的时间间隔数，单位是秒，第二个参数是采样的次数，如:

vmstat 2 1

2表示每个两秒采集一次服务器状态，1表示只采集一次。



现在开始讲解每个参数的意思。

**r** 表示运行队列(就是说多少个进程真的分配到CPU)，我测试的服务器目前CPU比较空闲，没什么程序在跑，当这个值超过了CPU数目，就会出现CPU瓶颈了。这个也和top的负载有关系，一般负载超过了3就比较高，超过了5就高，超过了10就不正常了，服务器的状态很危险。top的负载类似每秒的运行队列。如果运行队列过大，表示你的CPU很繁忙，一般会造成CPU使用率很高。

**b** 表示阻塞的进程,这个不多说，进程阻塞，大家懂的。

**swpd** 虚拟内存已使用的大小，如果大于0，表示你的机器物理内存不足了，如果不是程序内存泄露的原因，那么你该升级内存了或者把耗内存的任务迁移到其他机器。

**free**   空闲的物理内存的大小，我的机器内存总共8G，剩余3415M。

**buff**   Linux/Unix系统是用来存储，目录里面有什么内容，权限等的缓存，我本机大概占用300多M

**cache** cache直接用来记忆我们打开的文件,给文件做缓冲，我本机大概占用300多M(这里是Linux/Unix的聪明之处，把空闲的物理内存的一部分拿来做文件和目录的缓存，是为了提高 程序执行的性能，当程序使用内存时，buffer/cached会很快地被使用。)

**si**  每秒从磁盘读入虚拟内存的大小，如果这个值大于0，表示物理内存不够用或者内存泄露了，要查找耗内存进程解决掉。我的机器内存充裕，一切正常。

**so**  每秒虚拟内存写入磁盘的大小，如果这个值大于0，同上。

**bi**  块设备每秒接收的块数量，这里的块设备是指系统上所有的磁盘和其他块设备，默认块大小是1024byte，我本机上没什么IO操作，所以一直是0，但是我曾在处理拷贝大量数据(2-3T)的机器上看过可以达到140000/s，磁盘写入速度差不多140M每秒

**bo** 块设备每秒发送的块数量，例如我们读取文件，bo就要大于0。bi和bo一般都要接近0，不然就是IO过于频繁，需要调整。

**in** 每秒CPU的中断次数，包括时间中断

**cs** 每秒上下文切换次数，例如我们调用系统函数，就要进行上下文切换，线程的切换，也要进程上下文切换，这个值要越小越好，太大了，要考虑调低线程或者进程的数目,例如在apache和nginx这种web服务器中，我们一般做性能测试时会进行几千并发甚至几万并发的测试，选择web服务器的进程可以由进程或者线程的峰值一直下调，压测，直到cs到一个比较小的值，这个进程和线程数就是比较合适的值了。系统调用也是，每次调用系统函数，我们的代码就会进入内核空间，导致上下文切换，这个是很耗资源，也要尽量避免频繁调用系统函数。上下文切换次数过多表示你的CPU大部分浪费在上下文切换，导致CPU干正经事的时间少了，CPU没有充分利用，是不可取的。

**us** 用户CPU时间，我曾经在一个做加密解密很频繁的服务器上，可以看到us接近100,r运行队列达到80(机器在做压力测试，性能表现不佳)。

**sy** 系统CPU时间，如果太高，表示系统调用时间长，例如是IO操作频繁。

**id**  空闲 CPU时间，一般来说，id + us + sy = 100,一般我认为id是空闲CPU使用率，us是用户CPU使用率，sy是系统CPU使用率。

**wt** 等待IO CPU时间。