**项目上遇到服务慢、卡住等情况的解决方案**

**企业研发三部**

目录

[一、首先：需要项目上确认的内容 3](#_Toc10018468)

[1、确认部署方式，是集群（几个节点）还是单节点？ 3](#_Toc10018469)

[2、是并发慢，还是单用户就慢？ 3](#_Toc10018470)

[3、 确定服务卡的场景或功能点，个别还是所有？ 3](#_Toc10018471)

[4、确认程序版本 3](#_Toc10018472)

[二、 其次：需现场提供的分析文件及方法 4](#_Toc10018473)

[1、服务器资源占用情况 4](#_Toc10018474)

[2、 制作线程dump 5](#_Toc10018475)

[4、 飞行记录 6](#_Toc10018476)

[4、制作堆dump 7](#_Toc10018477)

[5、数据库awr文件（oracle） 7](#_Toc10018478)

[6、网络情况 11](#_Toc10018479)

[7、数据库相关配置及查询 11](#_Toc10018480)

[三、 单用户性能问题定位分析（yourkit） 11](#_Toc10018481)

[适用场景 11](#_Toc10018482)

[步骤一： 12](#_Toc10018483)

[步骤二： 13](#_Toc10018484)

[步骤三： 13](#_Toc10018485)

[步骤四： 14](#_Toc10018486)

[步骤五： 14](#_Toc10018487)

[步骤六： 15](#_Toc10018488)

[四、最后：常见研发问题分析 16](#_Toc10018489)

[五、其他：启动参数问题 17](#_Toc10018490)

[附录：性能优化案例 19](#_Toc10018491)

[中国建筑预算管理系统上线前压力测试 19](#_Toc10018492)

[粮食局项目优化记录 24](#_Toc10018493)

[国电投内存异常分析 28](#_Toc10018494)

[附录：AWR日志分析 30](#_Toc10018495)

[附录：ORACLE常用查询 32](#_Toc10018496)

[查询当前数据库所有的连接 32](#_Toc10018497)

[查询当前数据库锁的对象 32](#_Toc10018498)

[查询数据库执行的慢的语句 32](#_Toc10018499)

[根据sql\_id查询执行计划 33](#_Toc10018500)

[获取一个SQL绑定的变量 33](#_Toc10018501)

[通过snap\_id查看系统时间 34](#_Toc10018502)

更新说明

|  |  |  |  |
| --- | --- | --- | --- |
| **日期** | **版本** | **更新说明** | **作者** |
| 2018-12-24 | 20181224 | 初稿 | 龙双凤 |
| 2019-01-02 | 20190102 | 增加Linux下记录飞行记录的详细方式，以及如何选择配置文件 | 王星 |
| 2019-01-15 | 20190115 | 增加单用户性能问题分析工具yourkit的入门使用方法说明 | 龙双凤 |
| 2019-01-17 | 20190117 | 1、在yourKit步骤五中增加了Start CPU Profiling的三个模式的区别和说明。  2、增加了附录：性能优化按钮，同时增加了中建前段时间压测的分析情况，后续会追加粮食局最近的分析案例。  3、增加了附录：AWR日志分析（未完待续...） | 王星 |
| 2019-01-17 | 20190117 | 增加第5小节、数据库awr文件（oracle）补充了awr的制作过程 | 张凯 |
| 2019-02-13 | 20190213 | 增加了元旦时粮食局突发性能问题的分析案例 | 王星 |
| 2019-02-14 | 20190214 | 增加了国电投内存异常分析 | 王星 |
| 2019-05-23 | 20190523 | 新增了一种在Windows下简易制作线程dump的语句 | 王星 |
| 2019-05-29 | 20190529 | 新增了一个附录“ORACLE常用查询” | 王星 |
| 2019-07-17 | 20190717 | 新增Linux下ps和jstat联合排查CPU问题 | 王星 |
| 2020-05-15 | 20200515 | 修改堆dump制作命令错误问题 | 杨自明 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

当项目上反馈存在性能问题，操作卡等请出现，相关人员需要知道如何进行分析，需要哪些工具及日志来分析，什么样的情况采用什么样的方法，采集哪些分析日志，最后总结常见性能问题的场景、分析方法及优化策略。

# 一、首先：需要项目上确认的内容

## 1、确认部署方式，是集群（几个节点）还是单节点？

如果是集群或分布式部署，首先需确认是否是负载出问题：

（1）从负载均衡（nginx、f5等）后的统一访问入口访问一下；

（2）单独直接访问每个地址（如果外网不可访问，则内外或者登录到服务器访问）；

## 2、是并发慢，还是单用户就慢？

（1）单用户慢：是否测试环境或者生产环境用户量极少时候就慢；

1. 并发慢：是否是用户使用高峰期慢，但是用户降下来后就不慢，如果是则提供大致的高峰期在线用户数；

## 确定服务卡的场景或功能点，个别还是所有？

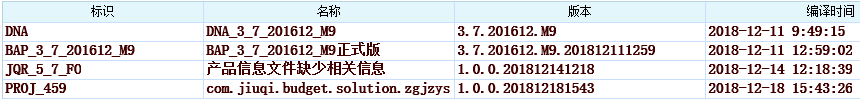
（1）经常出现的是登录或打开登录界面卡，一定要分清楚，是打不开登录界面，还是输入用户密码后点登录按钮卡或者打开首页卡；

（2）是某个功能慢（卡）还是所有功能（可以以一些非业务功能做验证，如基础数据、用户、登录等），还是只是打开某个表、运算等，以此初步定位问题；

（3）如果是集群部署，一般需区分是缓存资源类操作的卡还是非缓存类操作卡，比如基础数据、表样设计、公式管理、用户权限等均为缓存类，编制录入、运算、单据录入等为非缓存类操作；

## 4、确认程序版本

可访问服务：[http://ip:port/app/about 查看版本](http://ip:port/app/about查看版本)



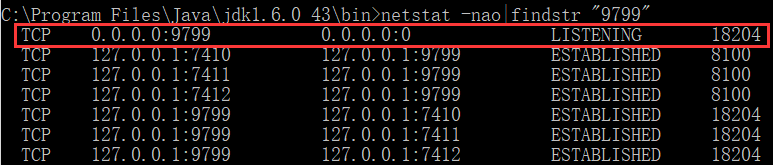
# 其次：需现场提供的分析文件及方法

## 1、服务器资源占用情况

首先查看应用服务器、数据库服务器的资源占用情况（内存、包括新生代，cpu、磁盘等，windows和linux方法不同）；

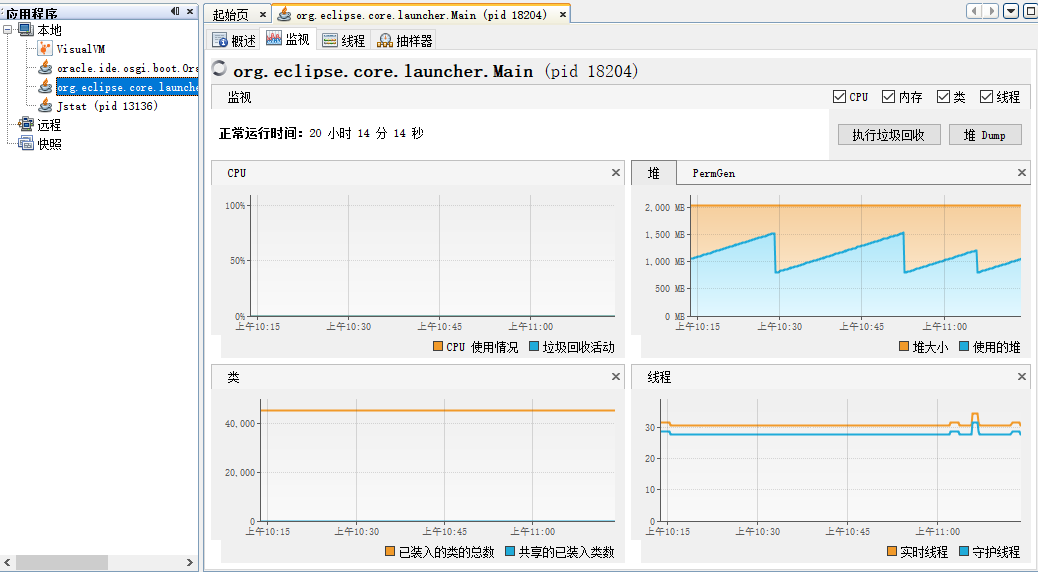
这里只介绍windows环境下的方法：

通过cmd命令：netstat -nao|findstr “服务端口号”，找到其中LISTENING状态的对应pid，如下图中的红框中的18204即为应用服务的端口号。



**方法1：**从任务管理器可以查看其内存、及整体CPU占用情况。

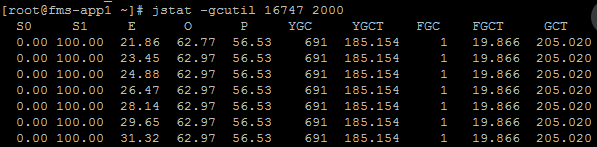
**方法2：**双击左侧虚拟机进程，右侧打开如下图中资源监控界面（监视页签），分4个区域，一般重点关注右上角的内存区域。



**方法3：**通过jdk自带的一些命令，如常用的查看内存占用情况并截图给研发分析：

命令：jstat -gcutil pid 输出间隔时间 示例：jstat -gcutil 16747 2000

即动态监控java进程的内存占用情况，pid是16747的进程，每2s输出一次；



|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 字段 | S0 | S1 | E | O | P | YGC | YGCT | FGC | FGCT | GCT |
| 说明 | 新生代耗用比例 | 新生代耗用比例 | 新生代耗用比例 | 老年代耗用比例 | 持久代耗用比例 | Minor垃圾回收次数 | Minor垃圾回总耗时 | Full垃圾回收次数 | Full垃圾回收总耗时 | 垃圾回收总耗时 |
| 对应启动参数 | -Xmn | -Xmn | -Xmn | -Xmx | PermSize |  |  |  |  |  |
| 说明 | 一般为xmx的1/3到1/2 | | |  | 一般512M够用 |  | 秒 |  | 秒 | 秒 |

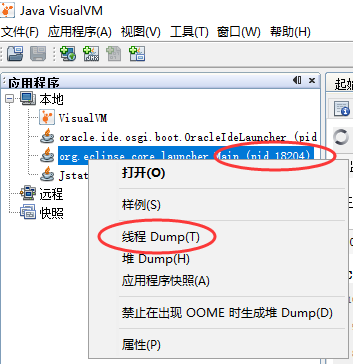
一般full垃圾回收过于频繁，一般需要加大堆内存大小或优化程序。

## 制作线程dump

线程dump是分析所有服务慢问题的首要方法，一般遇到服务卡都可先做线程dump再考虑重启恢复服务。如果直接就重启服务不留下分析日志，研发将无法提供任何优化协助。

**Windows系统：**

到jdk安装目录下，找到/bin/jvisualvm.exe文件，通过上述方法找到的pid找到对应的虚拟机进程，右键菜单即可制作线程dump或堆dump，线程dump用的比较多，做线程dump时间隔3-5s做一个，每次出问题做5个左右，将制作的线程dump发给研发分析。



或者进入jdk/bin目录下执行命令：jstack pid >d:/1.txt（每次执行需要换名字）

或者进入jdk/bin目录下执行命令：jstack pid >>d:/1.txt（每次执行不需要换名字）

**Linux系统：**

第一步：先找jdk安装路径：命令：echo $JAVA\_HOME

第二步：进入jdk安装路径 cd “路径”

第三步：找bin文件夹下的jstack文件   ls

第四步：查找进程号 ps -ef|grep java

第五步：运行此命令./jstack “进程号” >/home/appuser/`date "+%H:%M:%S:%N"`.txt

`date "+%H:%M:%S:%N"`.txt为以当前时间作为文件名。

## PS命令

使用下面的命令可以查看当前系统中所有线程的所有进程的CPU使用情况

|  |
| --- |
| ps H -eo user,pid,ppid,tid,time,%cpu,cmd --sort=%cpu |

使用下面的命令可以查看当前系统中PID为“48662”的所有进程的CPU使用情况，并输出到文件1.txt中

|  |
| --- |
| ps H -eo user,pid,ppid,tid,time,%cpu,cmd --sort=%cpu | grep "48662" >>1.txt |

## 飞行记录

应用服务器所使用jdk必须是1.7.4以上。飞行记录能够记录一段时间系统运行的主要日志，包括线程争用、内存耗用、垃圾回收、内存分配等情况。

**Windows系统：**

可以使用java/bin下的jmc.exe工具生成

**Linux系统：**

使用jcmd命令可以生成，如下：

./jcmd [pid] JFR.start delay=1s duration=60s name=serverRecording filename=recording\_`date +%Y%m%d%H%M%S`.jfr settings=profile

delay代表延迟时间；

duration代表记录时长；

settings代表配置文件，配置文件默认有两个，一个是profile.jfc，一个是default.jfc，如果不加settings，默认使用default.jfc，default.jfc中记录事件较少，profile.jfc中记录事件较多，一般使用profile.jfc就可以满足一般场景了，这两个jfc文件默认在jre\lib\jfr下，可针对个性化场景增加或删减事件；

## 制作堆dump

堆dump文件会比较大，一般约等于当时堆内存占用大小，一般都是几十个G。堆dump中记录了当前虚拟机中内存情况，可用于分析内存泄露等情况。

**Windows系统：**

可以使用java/bin下的jvisualvm.exe工具生成

**Linux系统：**

使用jmap命令生成，如下：

./jmap -dump:live,format=b,file=dumpfile.hprof [pid]

## 数据库awr文件（oracle）

如果从开发分析线程dump反馈效率问题都在数据库端，则可从现场提供awr分析。一般用于分析数据库一段时间内的运行情况，慢的sql语句等。

awr文件制作步骤：

一、以[**Oracle**](http://lib.csdn.net/base/oracle)用户登录到**[数据库](http://lib.csdn.net/base/mysql)**服务器

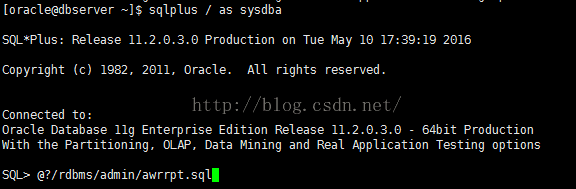
IMG_256

二、进入SQLPLUS

IMG_256

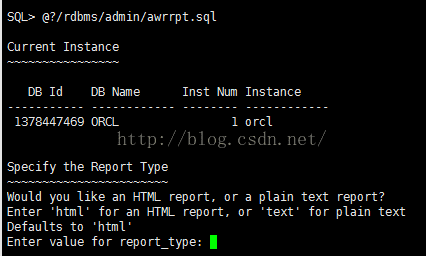
三、执行脚本

@?/rdbms/admin/awrrpt.sql



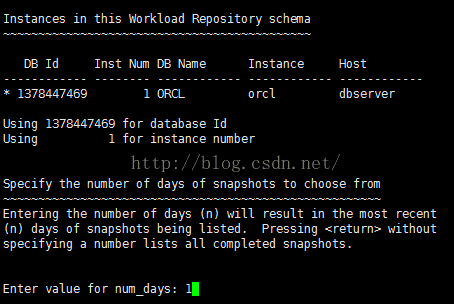
四、设定生成报告的格式

说明：不用输入，默认即可，直接按Enter



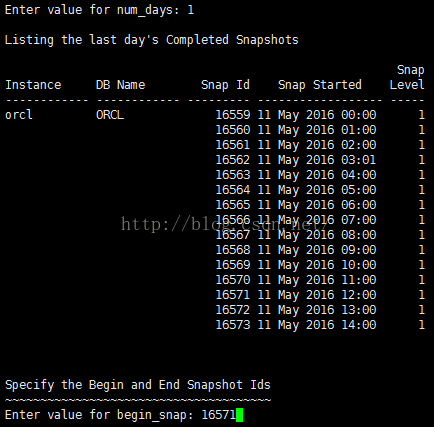
五、设定报告时间段的天数

说明：只导当天的时间点则输入1；导昨天的时间点则输入2；前天则3，默认最大7 天。



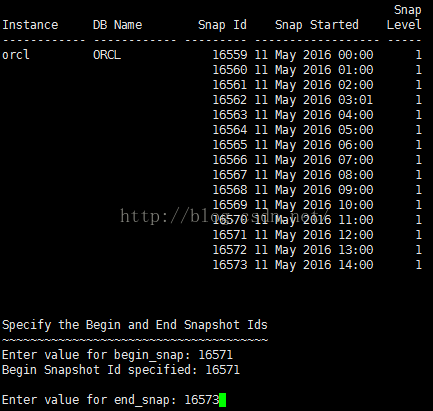
六、设定报告开始时间点

说明：例如下图所示16571代表的是2016年5月11日12时。



七、设定报告结束时间点

说明：例如下图所示16573代表的是2016年5月11日14时。

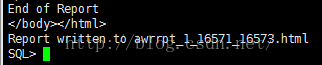


八、设定报告名称

说明：不用输入，默认即可，直接按Enter

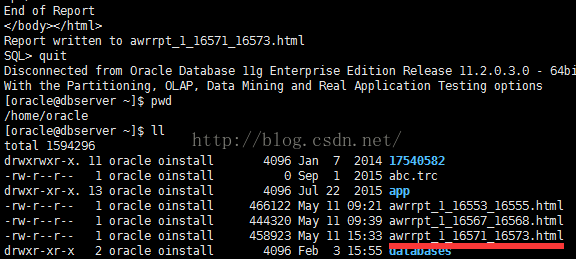


1. AWR报告生成正常结束



十、获取报告

说明：通过FTP软件，从当前路径中获取AWR报告。



## 网络情况

包括客户端到应用的网络，应用服务器到数据库服务器的网络（ping等方式验证）。网络问题涉及三段：每段都需要进行确认，ping默认为32字节的数据包，实际我们的页面大小可能达到100kB级别，因此在ping地址时可增加数据包大小的参数，如：ping 10.1.21.144 -l 20000。查看丢包情况及速度，另外可以换其他网络ping一下baidu等地址做对比。



## 数据库相关配置及查询

包括死锁、会话个数、允许连接个数。

# 单用户性能问题定位分析（yourkit）

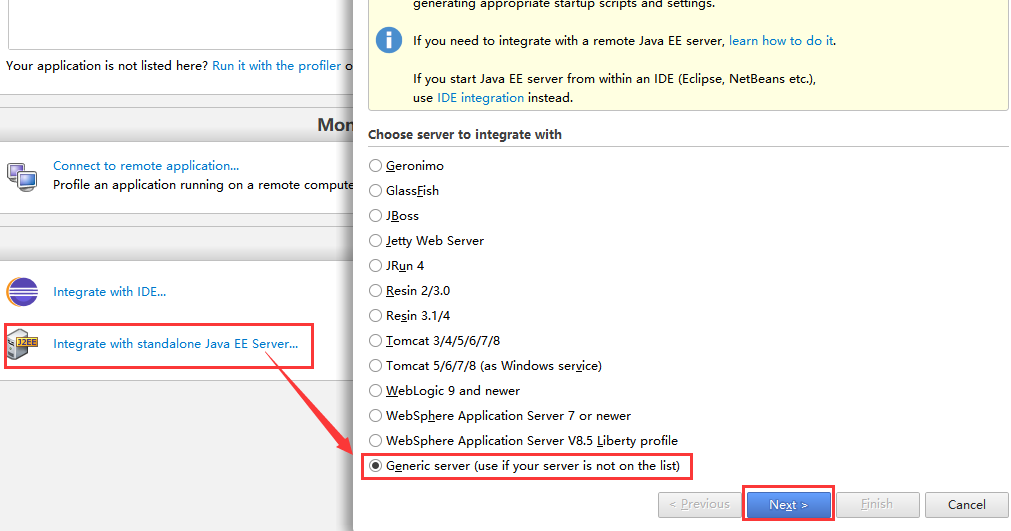
## 适用场景

如果单用户访问情况下效率低下，则最佳方式就是使用类似yourkit的工具进行分析，单用户慢的情况做线程dump分析并非第一选择。

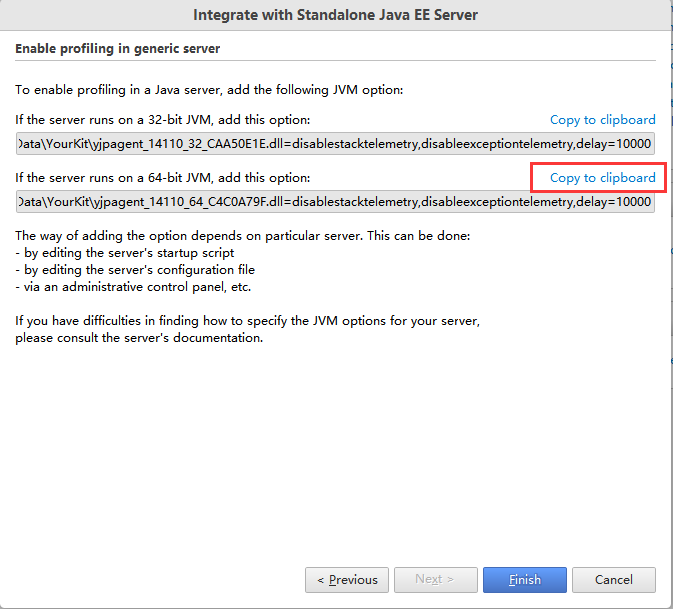
只有先完成了单用户下的性能优化，并发测试才有意义。

打开yourkit软件，按如下步骤选择，软件下载地址：\\file01\软件资源\非公共资源\开发辅助\YourKit

## 步骤一：

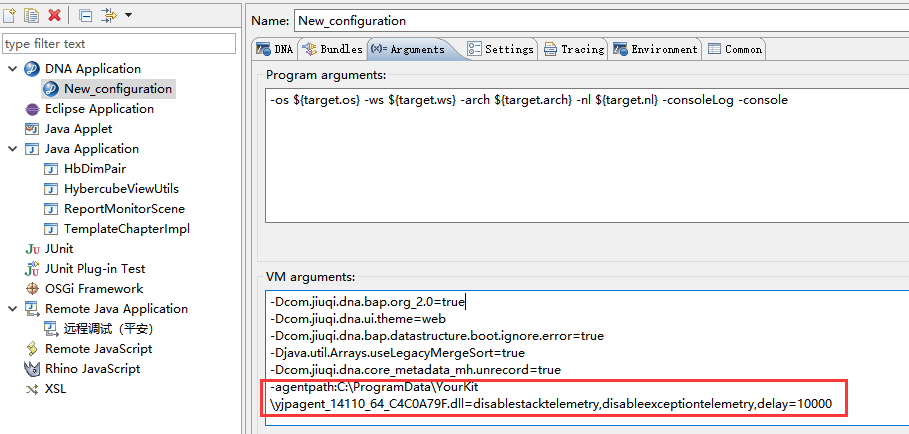


## 步骤二：



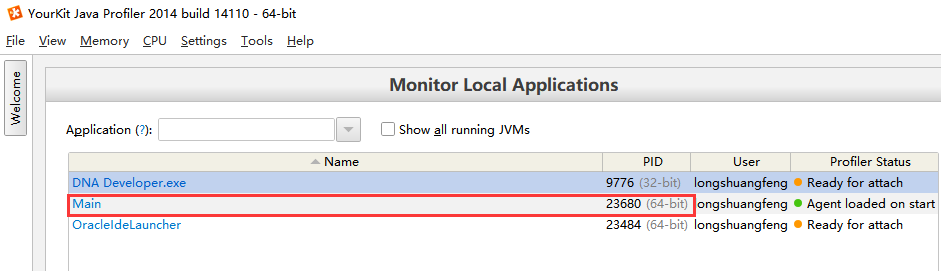
## 步骤三：

将步骤二拷贝的参数放到启动参数中，如下图：



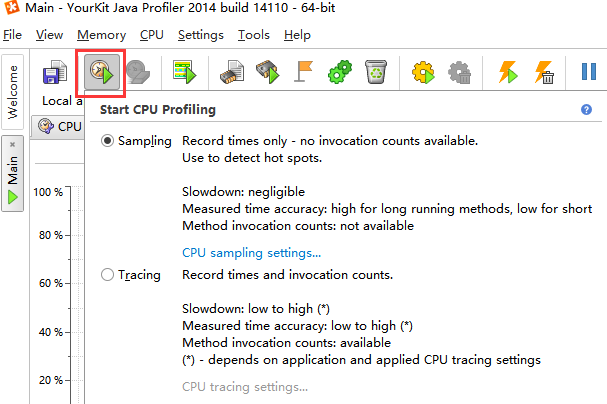
## 步骤四：

启动dna服务，并在yourkit主界面，根据pid，一般名称为Main的进程，点击打开：



## 步骤五：

浏览器上进入到待点击的功能界面，点击如下图按钮，然后点击浏览器上要跟踪的功能点。



Start CPU Profiling中的类型说明：

1. Sampling

该模式会定期的访问栈，评估出运行较慢的代码。该模式没有方法的调用次数记录，仅有CPU耗时。该模式是在进行提供瓶颈分析中最好的模式，对服务器额外开销最小。

1. Tracin

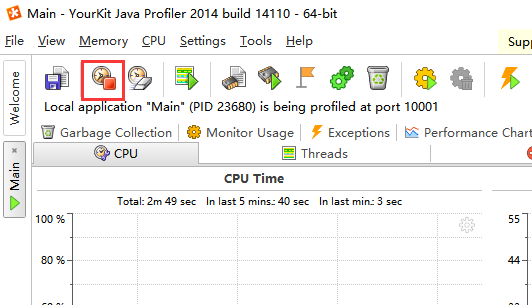
被监控应用的每一个方法都会被转化成字节码记录其消耗CPU的情况。方法调用次数和CPU耗时都被记录下来。但是Tracing模式对服务器影响较大。被检测服务的方法越多，对服务器性能影响越大。

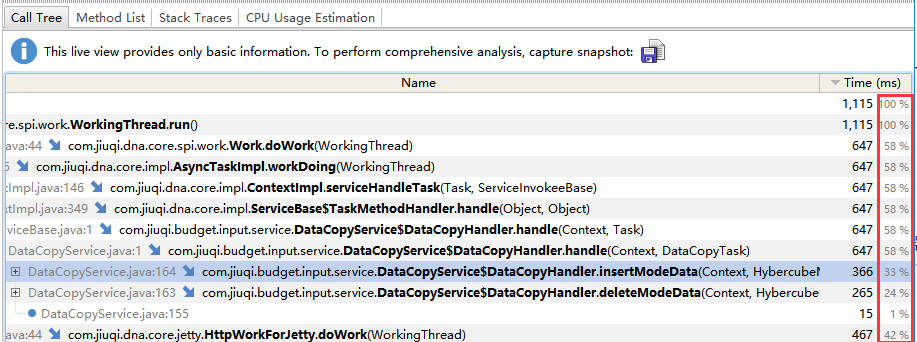
1. Call counting

最轻量级的CPU性能分析模式，对服务器的性能影响几乎为零。该模式结果不一定完全精确。为了能够保证对服务器几乎为零的性能影响，该模式没有应用任何locks和test-and-set 的atomic操作。因此如果同一个方法是多线程的，同时在不同CPU的不同核上调用的时候，就出发生计数错误。该模式不支持调用计数的auto activity mode，不收集hight-level-statistics（公司提供的2014版本上并没有这个模式）。

## 步骤六：

待浏览器中的功能操作完成后，在点击yourkit中的停止按钮，接下来就可以进行效率分析定位，如下图，重点关注各个方法的百分比，百分比越大说明其在整个操作过程中的占用时间越长，优化的重点方向。





# 四、最后：常见研发问题分析

性能问题分析日志文件一般主要分析线程dump和飞行记录，从而定位到是软件、网络、数据库等问题，定位或怀疑到问题场景后再进一步分析优化代码、定位网络、awr等。

1、由研发排查现场dump，大量runnable状态的带jdbc的类调用且多个线程中长期存在，则可能存在数据库死锁或者代码中逻辑有问题，存在死锁或者并发等待严重，导致事物不提交，可由现场查看数据库死锁、负载情况；

2、大量runnable状态的jdbc状态的线程，也可能导致数据库连接池满了，新线程等待allocate数据库连接池导致操作卡死；

3、oracle是行锁机制，同时只能由一个连接更新，更新多条数据的线程在更新完所有数据后才会提交数据库事务，其他线程才能锁定对应数据行进行更新操作，因此多个线程更新同一批数据时必须保证更新顺序一致，否则容易死锁。如线程1循环更新一个表中的多条数据，顺序为1,2,3,4,5，线程2也循环更新同一个表中的多条数据，顺序为5,4,3,2,1，则很容易出现死锁；

4、某些操作如字符串的contains、split等等，如果量太大则耗用cpu较多，导致争用严重，拖慢效率，可通过飞行记录分析查看争用线程、调用次数等分析；如非必须使用正则表达式对字符串进行处理，则可以用BudgetUtils中提供的字符串处理方法。

5、内存泄露，如某个大对象创建后被顶层对象引用，如会话、框架界面等，即当前请求结束后并不会销毁的界面或对象，则在垃圾回收时系统并不会释放这些对象内存，导致内存占满后服务卡死，可以通过-XX:HeapDumpOnOutOfMemoryError参数让JVM在内存溢出时创建堆dump文件，默认生成路径为satartup文件所属目录下，后缀为.hprof；

6、对于需要连接数据库的操作，如果循环较多，需要多次连接数据库、关闭数据库连接等操作，数据库连接和关闭是比较慢的操作，可尝试对数据库进行批量操作，然后在jvm内存中进行后续处理，如tfv\_batch\_zb；

7、如果单条sql查询的数据量太大（如千万以上），可能会占用大量jvm内存，导致应用服务内存占满或者触发大量的垃圾回收，暂停请求线程；

8、try catch，或者大量的打印日志操作，可能影响整体性能；

9、sql语句参数不要硬编码，要采用变量绑定的方式，硬编码方式不但效率低下，还会被安全扫描认定为存在sql注入的风险；

|  |
| --- |
| 硬编码：”select \* from tb where id=’+id+”’”  变量绑定：”select \* from tb where id=?” |

10、如果定位到是sql语句慢，除了业务上做优化外，sql本身优化可从加索引、修改表连接方式、分表、sql写法等方式；

11、由研发排查现场dump，大量in object wait状态的含有业务逻辑的线程存在，则可能存在着严重的GC问题；

12、由研发排查现场dump，大量waiting for monitor状态的含有业务逻辑的线程存在，则可能存在严重的资源争夺问题；

13、由研发排查飞行记录，发现JVM存在非常频繁的GC操作，且GC后内存回收效果十分明显，则需先确认TLAB中内存的分配情况，可能存在细小对象创建的过于频繁的问题，其次可以适当调整JVM参数，适当减少GC次数，需注意的是，年轻代及总内存并不是越大越好，内存越大，GC时间越长。

14、由研发排查飞行记录，发现JVM存在非常频繁的GC操作，且GC后内存并无明显变化，则需现场提供堆dump进行下一步分析，可能存在内存泄露，也可能是超大对象占满了内存；

15、集群情况下，更新缓存资源数据卡住时，如果线程dump中问题线程的栈中存在cluster，则说明大概率是集群缓存出现了问题；

16、存在一种情况，登录界面打不开（空白），通过netstat查看连接数异常高，是否存在负载均衡设备或软件在请求或者其他异常请求导致网络链接数占满；

17、一些重量级的计划任务（如批量复制数据、批量推送、接收数据、批量运算等），要配置好执行时机，不要过于频繁、选择合适的时机执行；

18、如编制表样打开较慢（有自动运算公式），先停用公式，如果速度快了，证明是自动运算公式影响，逐步启用部分公式定位是哪个公式效率低下，从而优化公式写法或优化公式执行逻辑等；

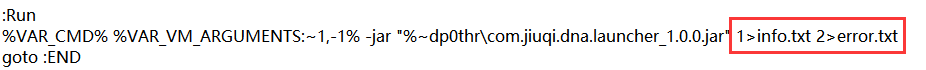
# 五、其他：启动参数问题

1、垃圾回收一般可配置使用G1回收机制，参数为：-XX:+UserG1Gc；

2、-XX:MaxPermSize为持久代最大内存，用于存储类、方法、静态内容等，和用户使用量关系不大（一般和da包中的模块个数成正比），一般情况512M足够，极端情况到1G足够；

3、参数xmn一般配置为xmx的1/3到1/2，预算类项目推荐1/2；

4、启动过程闪退：启动服务卡住或者启动窗口一闪或启动到半途自动关闭，则可将启动日志输出到文件中（增加如下红框中的启动参数），查看文件日志定位问题。



可能的问题有：

（1）jdk版本问题，更换jdk版本；

（2）启动过程中dna服务丢字段问题，增加启动参数忽略；-Dcom.jiuqi.dna.bap.datastructure.boot.ignore.error=true

（3）部分情况也遇到过jdk有bug的情况，升级jdk版本，如升级到1.7\_80版本；

5、集群服务第二个之后的节点启动过程卡住：集群部署，需按顺序启动，启动过程中需屏蔽掉所有界面访问入口（有配置可控制），包括外部webservice调用情况；

6、Minor垃圾回收单次耗时一般是毫秒级别，Full垃圾回收单次耗时一般可能几秒或者更长，且Full垃圾回收时会暂停所有用户线程，即所有界面操作卡住，可通过飞行记录或者jstat命令查看；

**附：预算项目效率问题场景及常见优化方案**

最常见的效率问题场景：

1. 编制表打开（不含自动运算）；
2. 运算操作：包括打开表时自动运算、全部运算等；
3. 汇总操作；

**优化方案（大方向）：**

1、**预解析：**通配公式预解析，减少实时解析时间；

2、**表分区：**数据库按年做分区，提升最常见的非跨年操作效率；

3、**批量处理：**预算编制表展示取数、运算、审核取数、更新操作批量执行，减少数据库交互次数；

4、**缓存化：**常用元数据、主数据等缓存化，提升查询效率；

5、**微服务：**对部分批量场景采用分布式、微服务等方案并行处理；

6、**避高峰：**对于实时性要求不是特别高的场景采用定时任务方式在低峰期执行；

# 附录：性能优化案例

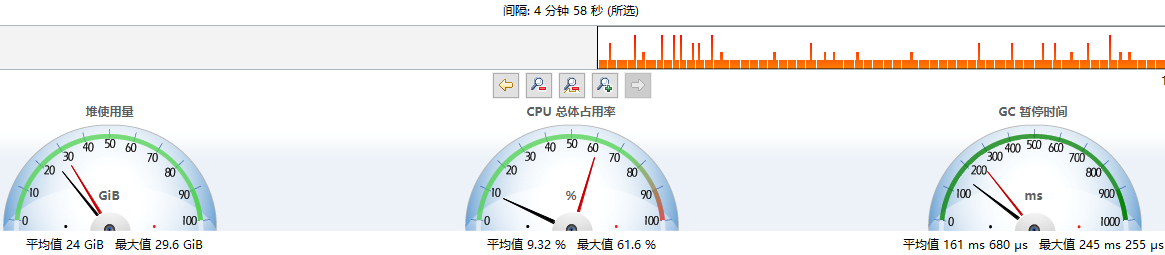
## 中国建筑预算管理系统上线前压力测试

应用服务是分布式环境，一个参数端，二十个应用端。用户统一访问Nginx，通过Nginx做转发，从而登陆到各个应用端节点。

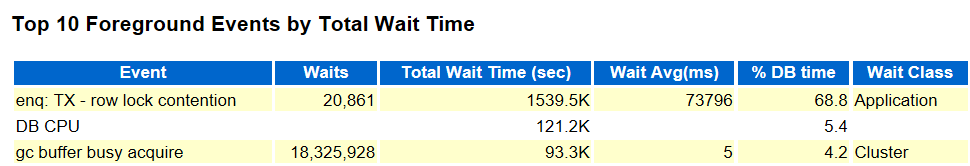
第一次2000人压力测试结果如下：



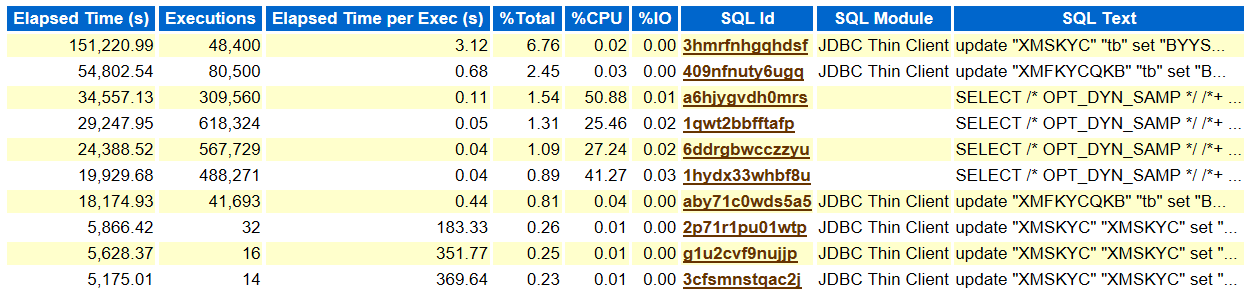
从结果来看，十分不理想，表打开、切换、保存、运算等操作都需要200秒以上。接下来开始分析问题，首先，先看下飞行记录，概览如下：



从中可以看出两点，1应用服务器CPU使用率很低，平均9.32%，最大值61.6%；2内存GC无明显异常。那么说明性能瓶颈可能不在应用服务上。这时候，我们需要看下数据库的AWR日志（仅Oracle数据库有，其他数据库需要数据库层面提供分析文件）。从AWR日志中发现了大量的数据库行锁的等待，如下：



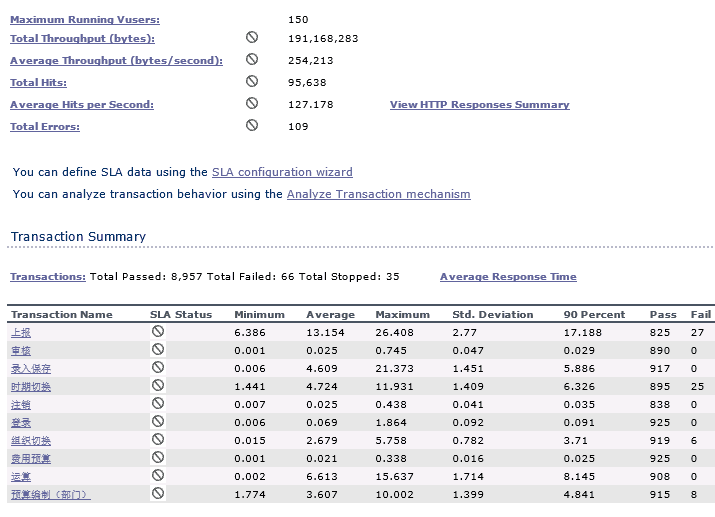
个别update语句甚至要几百秒才能执行完一次



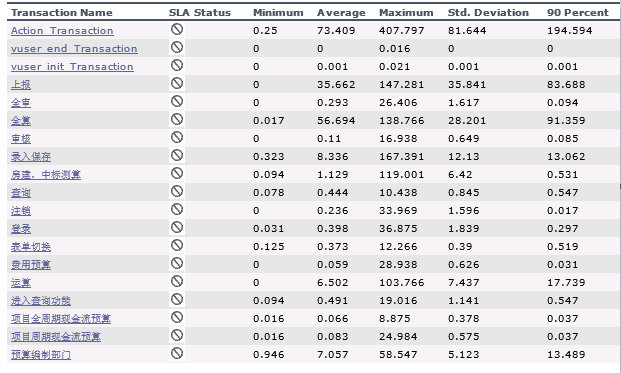
造成这种行锁的原因可能有两个：

1. 内存事物不提交，导致数据库事物无法提交，最终导致行锁的等待事件。
2. 数据库层面资源争夺严重，大量内存事物需要更新同一条数据库记录。

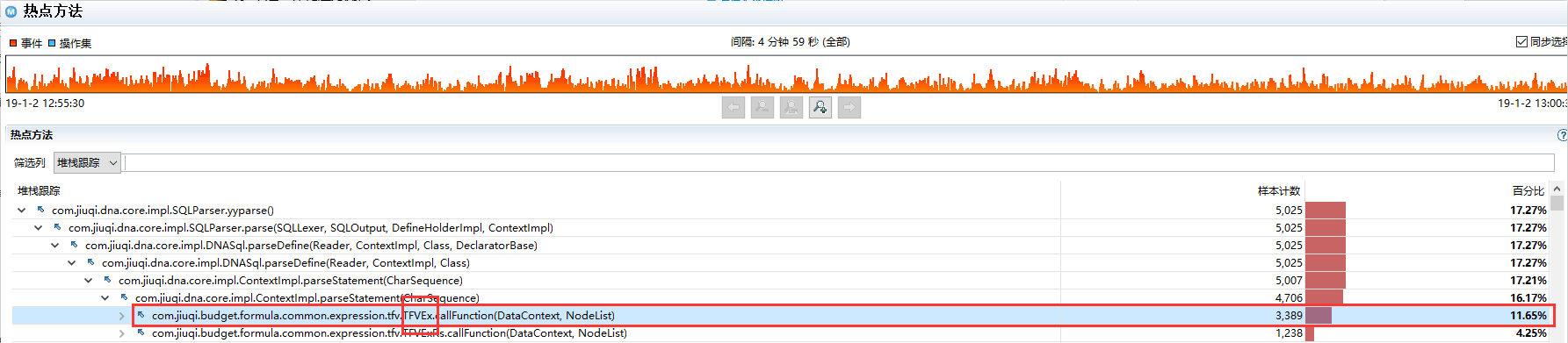
简单来说，要么是应用层面跑的慢，最终影响了数据库层面。要么是业务层面定制的用户场景不合理，导致应用中多个线程同时更新数据库的一条记录。为了定位问题，我们决定先压测一个节点，看看应用层面有没有问题，单节点150压力测试结果如下：



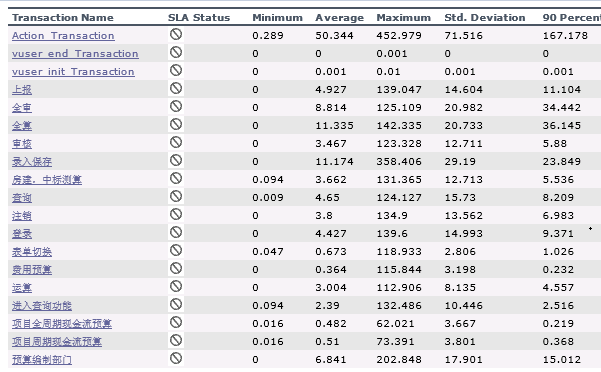
从结果上看远远比2000人20个节点时好，说明程序本身没有太大问题，后来又做了一次2000人的压力测试，结果依旧是行锁严重，系统响应慢。所以，实施方面重新定制了压测的用户，然后又跑了一次2000人的压力测试，结果如下：



在换了新的用户之后系统明显变快了，再次说明系统本身没有太大的性能瓶颈，之前2000人压力测试结果不理想应该是准备的用户不合理导致的。但是从这次的结果上依旧存在某些操作耗时较长。分析飞行记录后发现在热点方法中TFVEx公式占了很大的比重，如下：



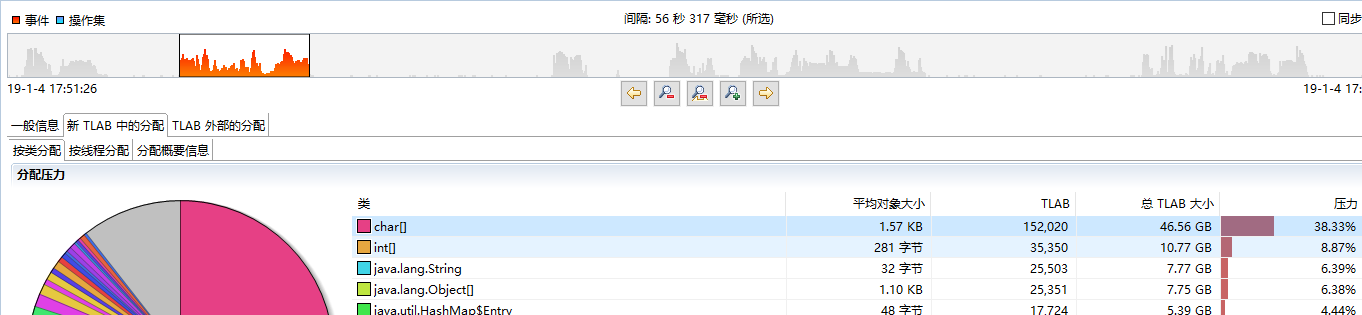
在和实施了解过过业务场景后，我们发现，目前系统中存在大量的TFVEx运算、审核公式，所以在全部运算和上报时系统响应速度较慢。针对性的分析过系统中的TFVEx之后，我们指导实施将TFVEx尽可能改成TFVExBatch\_ZB或者单元格表达式公式。公式调整后，性能组又做了一次压力测试，结果已经非常好了，所以我们又做了一次4000人的压力测试，在4000人的压力下，系统又开始出现了卡顿的情况，如下：



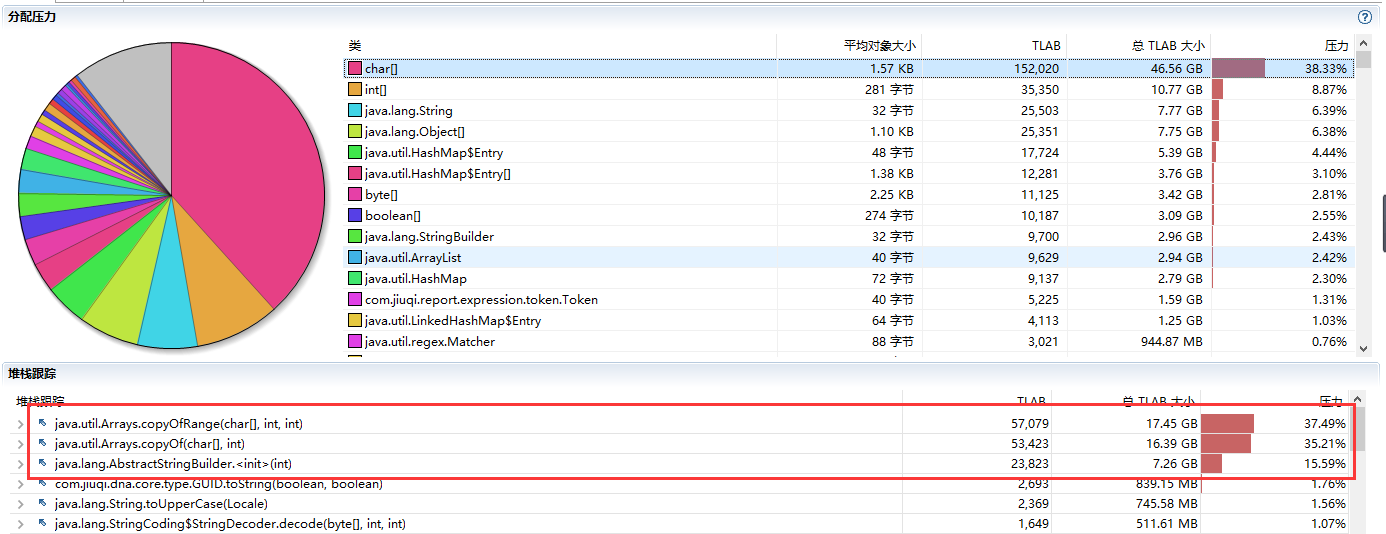
分析飞行记录后发现GC耗用了比较多的时间：



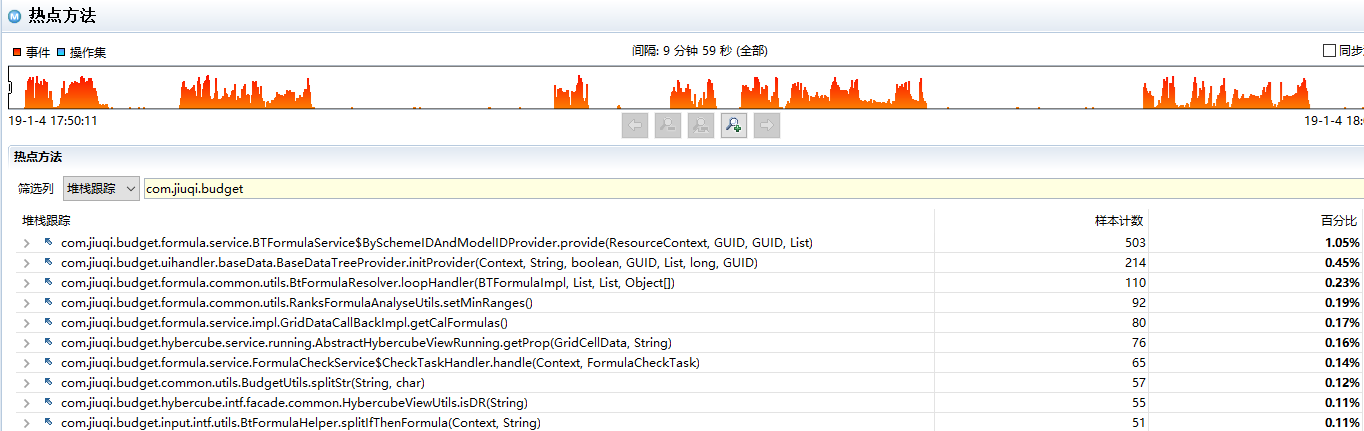
GC耗时，一般来说可能是短时间内产生了大量的零散的小对象，这些小对象一般会被创建到TLAB上，截取了一段50多秒的记录看下：



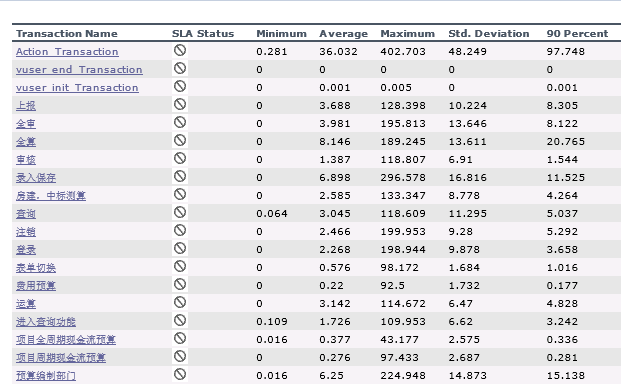
占用比例最大的就是char[]了这个大多是String对象new的太多了，接着从跟踪堆栈中看下是否存在预算相关的代码，然后针对性的优化就可以了：



同时，在热点方法中搜索预算相关的代码并进行分析：



最终进过一轮优化后，4000人压力测试的结果如下：



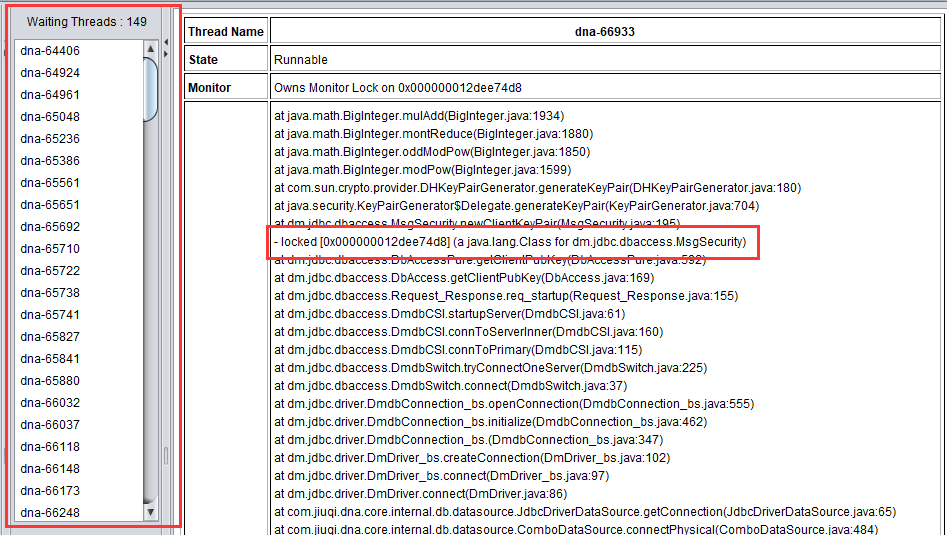
这个结果已经让我们觉得比较满意了，本次优化的代码已经提交到[*BUDGET-7190*](http://10.2.12.123:8080/browse/BUDGET-7190)。

## 粮食局项目优化记录

应用服务器有三台，一共是5个节点的DNA集群。数据库用的是DM7，主备集群，备机仅提供数据备份功能。

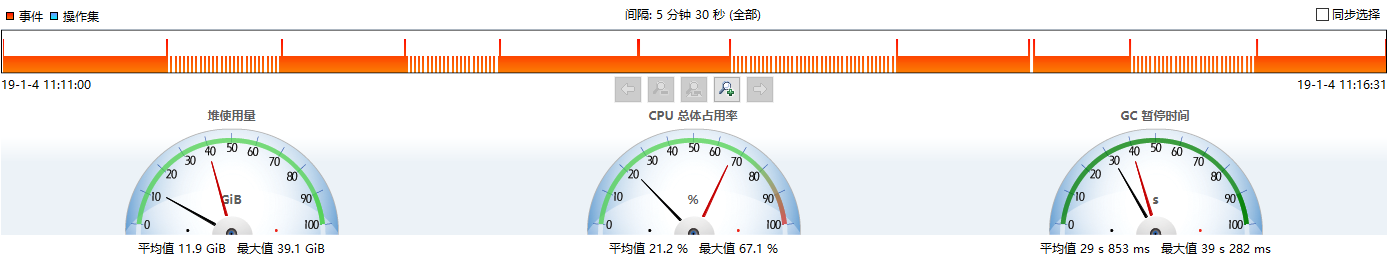
问题描述，从2018年12月30日起，系统开始卡顿，一直到元旦节后依然无法解决。这个时候正赶上年底开始年报填报以及12月月报的填报，系统压力比较大，用户点击月报功能点平均要2分钟才能进入功能点，随便点一点就非常卡。

系统在发生这个问题之前并没有换过什么增量，最近一次的增量也是将近一个月以前换上的。因为是月报或年报功能点打开慢，所以针对具体的功能点的打开，我们做了线程dump，部分截图如下：



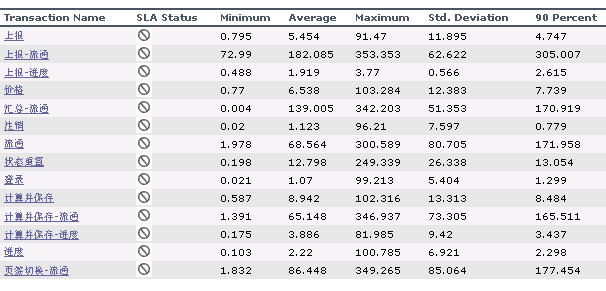
从图中发现，在DM JDBC的驱动中有一个全局的独占锁，每次连接数据库时都要先等待这个独占锁，请求到独占锁的线程才能进行数据库连接。针对这个问题我们联系了达梦的运维同学，然后达梦给出了一个更新版的JDBC驱动，打算第二天再观察一下。

第二天，情况依旧没有好转……继续看线程dump……从现场反馈回的线程dump中并没有看到某一个具体的慢的操作，JDBC的问题也已经得到了解决。同时我们发现系统属于整体卡顿并不是某个功能点的卡顿，所以我们做了飞行记录，如下：



从图中可以看出，堆的使用量的平均值并不高，CPU占用率也不高，但是GC却很高，平均29秒，最大值39秒。现在只要降低GC时间，减少GC次数应该就可以解决问题了。由于我们近期没有上过什么增量，所以想先从JVM参数着手进行调优。

先看下后台模拟单节点200人绝对并发的压力测试截图：



基本是惨不忍睹了。JVM参数方面我们咨询了刘艳辉，他了我们一些建议：G1相比CMS在预测和压缩方面有优势，G1目标是替代CMS，但直到java9出来才默认使用G1，我们还在用7，说不好会有什么bug，所以在不修改程序代码的情况下可以G1和CMS都尝试一下。所以有了如下两套启动参数：

CMS回收方式：

|  |
| --- |
| -XX:+UnlockCommercialFeatures -XX:+FlightRecorder -XX:+HeapDumpOnOutOfMemoryError  -XX:MaxNewSize=20480M -XX:MaxPermSize=1024M -XX:PermSize=1024M -Xmn37000M  -Xms50000M -Xmx50000M -XX:+PrintGCDetails -XX:+PrintCommandLineFlags  -Duser.timezone=GMT+08 -XX:+DisableExplicitGC -Djava.util.Arrays.useLegacyMergeSort=true  -XX:SurvivorRatio=6 -XX:-UseAdaptiveSizePolicy -XX:+UseConcMarkSweepGC  -XX:+UseCMSCompactAtFullCollection -XX:CMSFullGCsBeforeCompaction=5  -XX:CMSInitiatingOccupancyFraction=40 -XX:+UseCMSInitiatingOccupancyOnly  -XX:+CMSScavengeBeforeRemark |

重点参数说明：

-XX:+UseCMSCompactAtFullCollection使用并发收集器时,开启对年老代的压缩

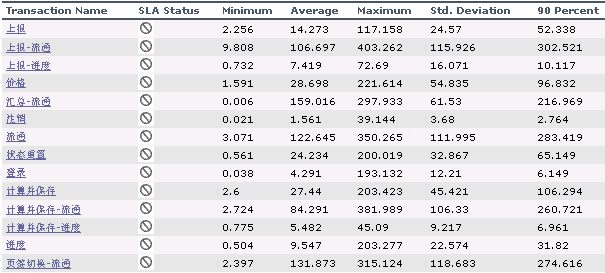
-XX:CMSFullGCsBeforeCompaction=5 这里设置多少次Full GC后,对年老代进行压缩

-XX:CMSInitiatingOccupancyFraction=40 是指设定CMS在对内存占用率达到40%的时候开始GC(因为CMS会有浮动垃圾,所以一般都较早启动GC)

-XX:+UseCMSInitiatingOccupancyOnly 只是用设定的回收阈值(上面指定的70%),如果不指定,JVM仅在第一次使用设定值,后续则自动调整

-XX:+CMSScavengeBeforeRemark GC前启动一次ygc，目的在于减少old gen对ygc gen的引用，降低remark时的开销,一般CMS的GC耗时 80%都在remark阶段

效果如下：



G1回收方式：

|  |
| --- |
| -XX:+UnlockCommercialFeatures -XX:+FlightRecorder  -Dcom.jiuqi.dna.bap.datastructure.boot.ignore.error=true -Dfile.encoding=GBK  -Dosgi.console.blockOnReady=true -XX:+HeapDumpOnOutOfMemoryError -XX:+UseG1GC  -XX:MaxNewSize=20480M -XX:MaxPermSize=1024M -XX:PermSize=1024M -Xms50000M  -Xmx50000M -XX:MaxGCPauseMillis=5000 -XX:G1HeapRegionSize=32M -XX:+PrintGCDetails  -XX:+PrintCommandLineFlags -Duser.timezone=GMT+08 -XX:+DisableExplicitGC  -Djava.util.Arrays.useLegacyMergeSort=true -XX:InitiatingHeapOccupancyPercent=30  -XX:ParallelGCThreads=16 -XX:ConcGCThreads=4 |

重点参数说明：

-XX:MaxGCPauseMillis=5000设置G1收集过程目标时间，默认值200ms，不是硬性条件。在设置了这个参数后，最好不要再配置年轻代，否则会覆盖。

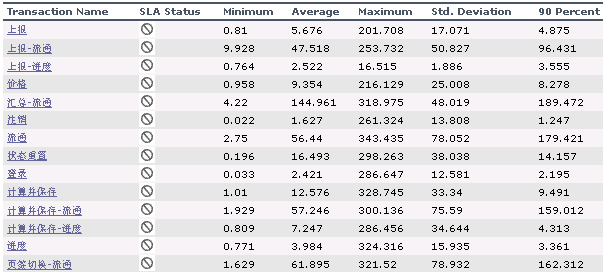
-XX:G1HeapRegionSize=32M 一个Region的大小，取值范围从1M到32M，且是2的指数

-XX:InitiatingHeapOccupancyPercent=30 启动并发GC时的堆内存占用百分比。G1用它来触发并发GC周期,基于整个堆的使用率,而不只是某一代内存的使用比例。值为 0 则表示“一直执行GC循环)'. 默认值为 45 (例如, 全部的 45% 或者使用了45%)

-XX:ParallelGCThreads=16 设置垃圾收集器并行阶段的线程数量为16（这个看具体的服务器的逻辑CPU个数，粮食局的服务器的逻辑CPU是40个，一般配置为一半左右）

-XX:ConcGCThreads=4 设置并发垃圾收集器使用的线程数量为4

效果如下：



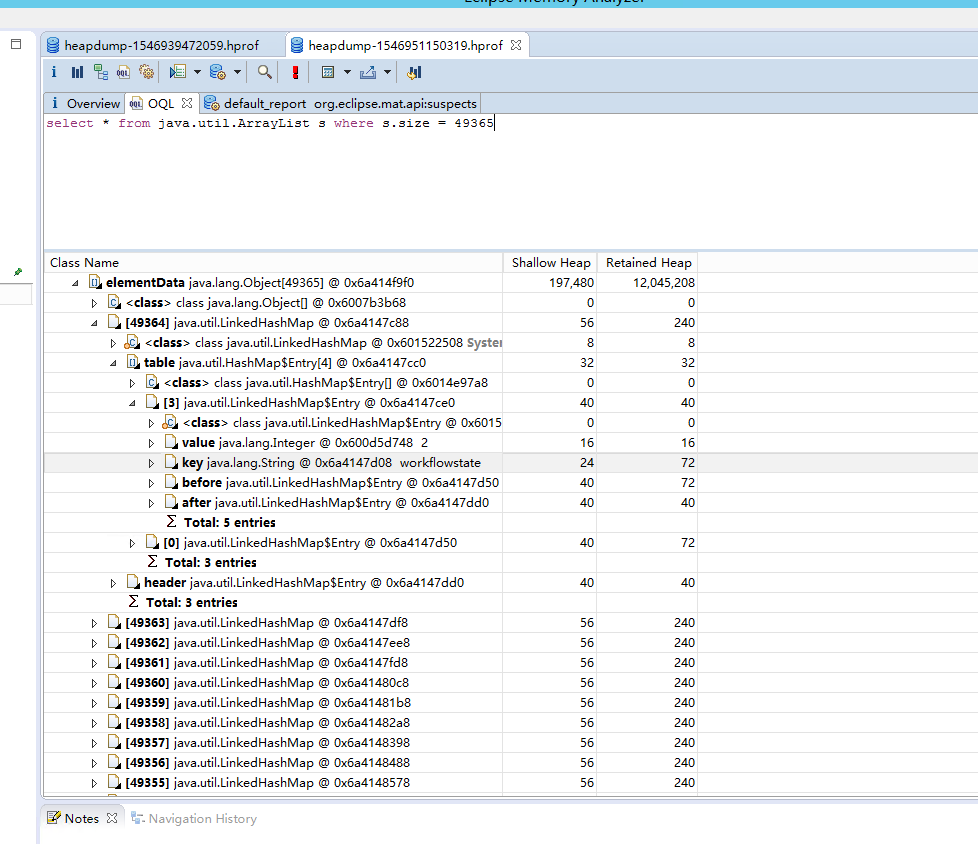
可以说仅仅调整了JVM参数就能使系统的响应时间有本质上的提升，这个简直amazing！但是这个结果依旧不能满足用户需求。同时我们发现服务刚启动完毕时，在没有人访问的情况下，月报功能点打开也要10秒左右。说明程序本身还有很大的优化空间。分析线程dump后发现，打开月报时，最耗时的是主单位树的着色的查询，SQL语句如下：

|  |
| --- |
| DEFINE QUERY "commonQuery"()  BEGIN  SELECT  "T"."orgid" AS "orgid",  "T"."workflowstate" AS "workflowstate"  FROM  "BT\_DATASTATE\_LOG" AS "T"  WHERE  "T"."linkid" = 'NONE' AND "T"."modeid" = '4DAD9E96A00000013D146A729767BA37' AND "T"."periodnum" = 12 AND "T"."year" = 2018 AND "T"."period" = 'Y' AND "T"."serialno" = '0000'  END |

这个语句在我们后台的服务器上执行并获取全部，大约耗时0.3秒，但是同样的查询，在数据库服务器上执行，查询并获取全部，居然要4秒。查询结果约5万条记录，后台数据库比正式环境少1千多条记录，说明正式环境数据库服务器的磁盘读取性能很差。所以这个启动完打开也不快的问题只能在正式环境上重现，后台测试环境从来没出现过。

但是这个SQL也是有问题的，这个SQL查询的数据量太大了，这个语句按照上报对象、时期、linkid查询了所有机构的状态，正好粮食局的组织机构数据量又多。如果正式环境想提升效率，那么必须优化这段逻辑，按需查询。

同时，通过堆dump，我们发现一个长度为49365的ArrayList<Map<String,Object>>居然有20M，（前面的SQL在后台数据库查询后总数据量为49365）如下图：



一个长度为16万ArrayList<FbaseDataObjct>才1M不到，可见Map对内存的浪费…….

由此，我们猜测了一下，每次打开月报功能点都需要查询状态表，状态表的本来就查询非常慢，在业务高峰时期，可能会更慢，同时，这个操作必须得申请到20M的内存来存储这5万的记录，在高并发下，内存不足，通过不断的FGC来压榨内存，最后体现上就是啥啥都卡。

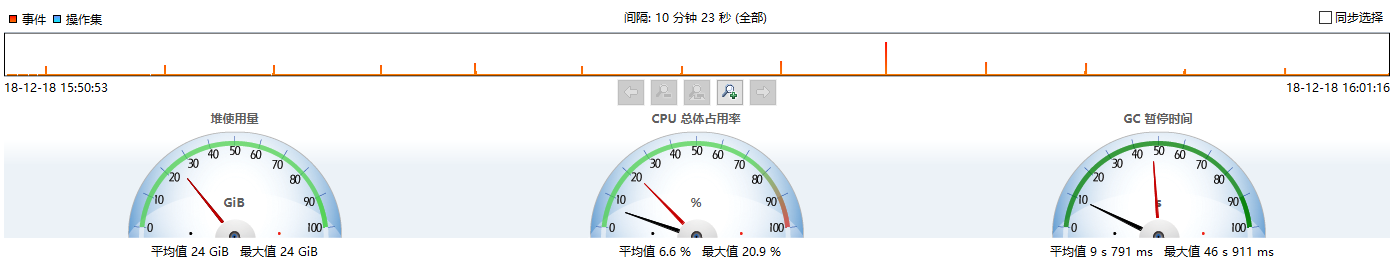
在对着色进行了优化之后，系统有了非常明显的性能提升，业务高峰期顺利度过。

粮食局这次的问题爆发的十分凶猛，而且后台的压力测试并没有帮助我们100%重现问题，压力测试虽然也卡，但是根本原因不同，所以有些时候不能完全死盯着后台的压力测试环境。这次性能问题有两个关键点，一个是通过JVM参数调优，直接大幅提升了系统的响应时间，一个是线程dump+堆dump定位了正式环境单用户慢的问题。当然这次的优化还伴随着一些其他的优化，这些代码都提交到[*BUDGET-7198*](http://10.2.12.123:8080/browse/BUDGET-7198)、*[BUDGET-7262](http://10.2.12.123:8080/browse/BUDGET-7262)*了。

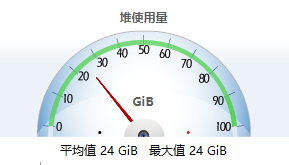
## 国电投内存异常分析

国电投项目，Nginx集群，两台DNA服务。系统登录非常卡，从浏览器输入服务器地址，点击回车后，需要等待2分钟左右才能出来登录页面，再过2分钟才能进入系统，没有用debug模式，非业务高峰期，用户在线人数很少，当前主要业务就是上传一些附件到系统中。服务重启后，会好很多，但是过半小时左右，又开始卡顿。

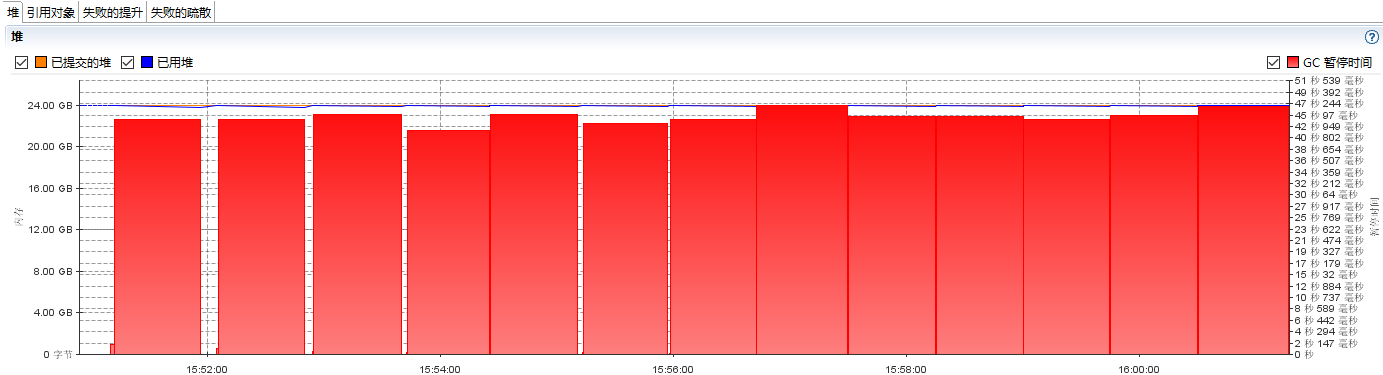
由于是系统整体卡顿，所以直接跳过了线程dump，从飞行记录开始看，如下图：



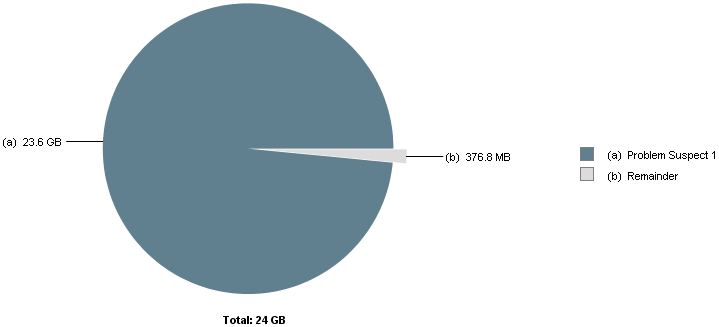
首先，GC问题很明显，平均9秒，最大46秒。这个和现在系统的现象对上号了，说明系统卡顿的主要原因是GC暂停。然后，还有一个很罕见的问题“堆使用量”，平均值24G，最大值24G，如下：



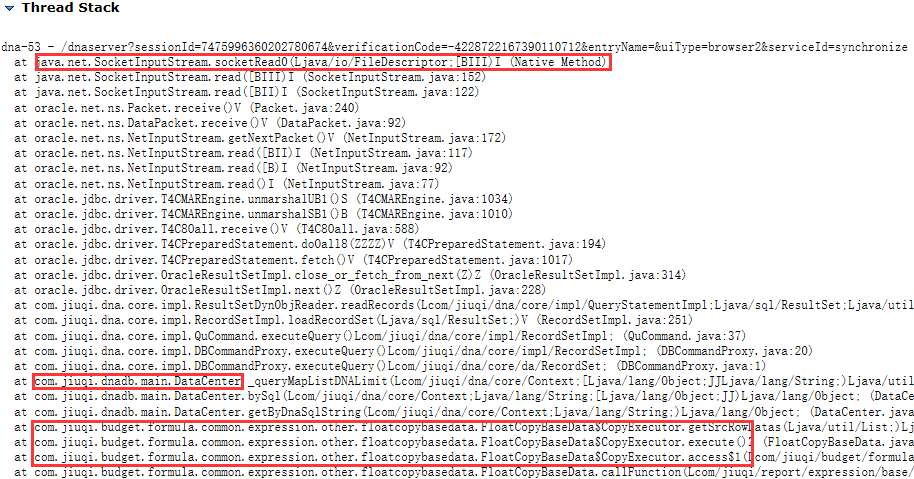
这说明，频繁的GC几乎没有任何效果。飞行记录中切到“内存”→“垃圾回收”如下：



从图中可以看出不论如何GC，堆的大小几乎没有任何变化。同时，每次重启都能好一会儿，过会儿有卡顿，这说明很有可能出现了内存泄露。这时候需要借助堆dump来观察下内存情况了。使用met加载堆dump文件后，从OverView中可以看出来在24G的堆中，有一个对象占用了23.6G，如下图：



定位一下这个对象的线程信息，如下图：



从图中可以看出这个是一个running的数据库读的操作，由FloatCopyBaseData发起，通过DataCenter读取数据库中的数据库，而且居然已经读了23.6G的对象……万幸的是，并没有发现内存泄露，仅仅是一个超大内存对象而已。

通过和现场的实施同胞沟通，我们定位到了这个FloatCopyBaseData公式，发现里面的SQL写的有问题，不断的full join导致结果集无穷无尽（至少我在sql developer中没执行完）。修改SQL语句后，这个问题没有再次出现，系统稳定了下来。

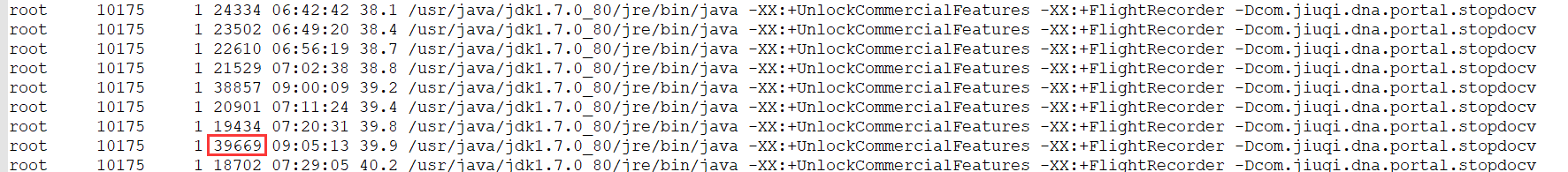
国电投的这个问题是一个典型的内存问题，和代码没有关系。这种完全需要堆dump才能定位问题的情况比较少见，所以和大家一起分享一下。堆dump文件及报告位置*[详见](file:///\\\\wangxing\\分析文件\\国电投\\20181218系统无法登陆)*。

# Linux下线程CPU使用分析

在程序运行过程中，经常出现服务器CPU占用过高的问题。出现这种问题大多都是录飞行记录才能排查，但是飞行记录作为商业特性可能将来会被替代掉，所以这里介绍一种不使用飞行记录也能精准定位CPU消耗线程的方式。

首先，需要用PS命令（详见[PS命令](#_PS命令)）将当前进程的线程的CPU使用情况输出出来，同时使用JSTAT命令制作线程dump。

在PS命令中，找到CPU占用较高的线程的TID。需要注意的是，这里的TID是十进制的，但是thread dump中的nid是16进制的，需要转换。如下“39669”转换成16进制之后是“0x9af5”，在thread dump中查找“0x9af5”，就可以看到对应的线程的dump了。

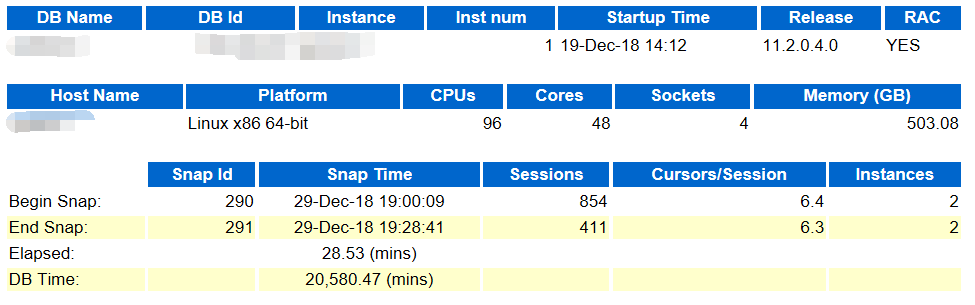




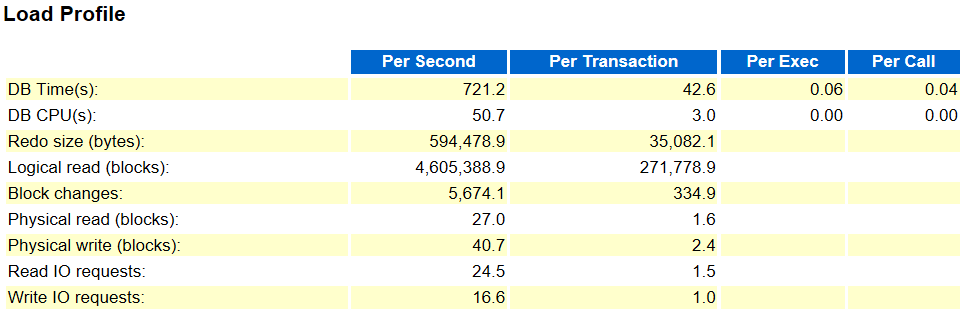
# 附录：AWR日志分析

在Oracle数据库学习和使用中，遇到性能问题，首要的步骤就是导出AWR分析报告，AWR是Oracle的一个脚本工具，通过周期性快照记录下当时的所有运行数据，数据库管理员可以导出其中一部分数据进行分析，从而找出来哪些脚本导致了目前的数据性能问题。

首先数据库概要信息及AWR日志时间区间如下：



从概要信息中看出来，在28.53分钟里，DB Time居然有20580.47分钟，数据库服务器有96个逻辑核，20580.47/96≈214分钟。远大于28.53，说明数据库CPU处于繁忙状态。公式： DB Time=DB Wait Time（用户会话的非空闲等待）+DB CPU Time（前台session调用）

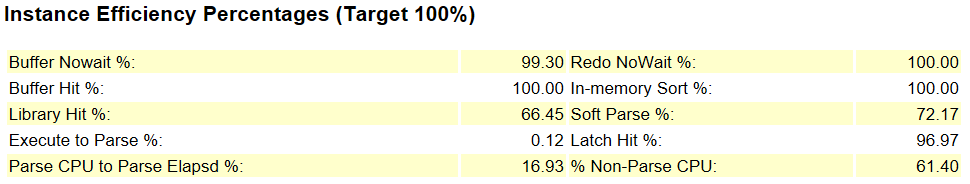


其中：

DB Time（Per Second）=DB Time/Elapsed=721.2

DB CPU（Per Second）=DB CPU/Elapsed=50.7

50.7/721.2\*100%≈7%，说明在DB Time中只有约7%为DB CPU Time，其余为非空闲等待时间，非空闲等待时间占用了大量的资源



1. Buffer Nowait %：表示会话向Database Buffer Catch【数据高速缓冲区】申请1个缓存时不等待的比例
2. Buffer Hit %：表示数据高速缓冲区的命中率，也叫Catch Hit Ratio。该指标应按照实际业务系统类型来分析，如OLAP系统，该值可能在20%就算是合理了，而对于OLTP来说，理想值应该在90%以上。当然，并非该值达到100%就没问题了，系统中可能依然难以避免物理读等待。
3. Library Hit %：Library Catch Hit Ration【库高速缓冲区命中率】，表示向共享池的Library Catch中申请1个Library Catch Object对象时，其已经在Library Catch中存在的比例。该指标的一个理想值应该达到95%以上。
4. Execute to Parse %：表示执行解析比例（SQL语句解析后被重复执行的命中率），目标是希望一次解析多次执行。
5. Parse CPU to Parse Elapsd %：表示解析消耗的CPU时间与解析消耗的总时间的比值，目标是100%。我们希望在解析的过程中时间都消耗在CPU上，而不希望在解析的过程中出现其他等待事件从而拉长解析消耗的总时间。如果该指标偏低，说明在解析的过程中，除了消耗CPU资源以外，还有其他等待事件，如等待共享池对象、[闩锁](https://blog.csdn.net/zengmuansha/article/details/6757812)。
6. Redo NoWait %：表示写Redo Entry时不等待的比例，redo相关的资源如redo space request争用可能造成redo时的等待
7. In-memory Sort %：表示在内存中排序的比例。
8. Soft Parse %：表示软解析比例，太低则需要调整应用使用绑定变量。[软解析与硬解析详见](https://www.cnblogs.com/discuss/articles/1866946.html)。
9. Latch Hit %：表示以willing-to-wait方式去获取内存闩锁的命中率指标，通常这个指标要求至少在99%一上，否则很有可能意味着大量的闩锁，影响性能。
10. %Non-Parse CPU：表示除解析之外的CPU使用率。

# 附录：ORACLE常用查询

## 查询当前数据库所有的连接

|  |
| --- |
| select \* from gv$session; |

## 查询当前数据库锁的对象

|  |
| --- |
| SELECT l.session\_id sid,  s.serial#,  l.locked\_mode 锁模式,  l.oracle\_username 登录用户,  l.os\_user\_name 登录机器用户名,  s.machine 机器名,  s.terminal 终端用户名,  o.object\_name 被锁对象名,  s.logon\_time 登录数据库时间  FROM gv$locked\_object l, all\_objects o, gv$session s  WHERE l.object\_id = o.object\_id AND l.session\_id = s.sid ; |

## 查询数据库执行的慢的语句

|  |
| --- |
| SELECT S.SQL\_TEXT,  S.SQL\_FULLTEXT,  S.SQL\_ID,  ROUND(ELAPSED\_TIME / 1000000 / (CASE  WHEN (EXECUTIONS = 0 OR NVL(EXECUTIONS, 1 ) = 1) THEN  1  ELSE  EXECUTIONS  END),  2) "执行时间'S'",  S.EXECUTIONS "执行次数",  S.OPTIMIZER\_COST "COST",  S.SORTS,  S.MODULE, --连接模式（JDBC THIN CLIENT：程序）  -- S.LOCKED\_TOTAL,  S.PHYSICAL\_READ\_BYTES "物理读",  -- S.PHYSICAL\_READ\_REQUESTS "物理读请求",  S.PHYSICAL\_WRITE\_REQUESTS "物理写",  -- S.PHYSICAL\_WRITE\_BYTES "物理写请求",  S.ROWS\_PROCESSED "返回行数",  S.DISK\_READS "磁盘读",  S.DIRECT\_WRITES "直接路径写",  S.PARSING\_SCHEMA\_NAME,  S.LAST\_ACTIVE\_TIME  FROM GV$SQLAREA S  WHERE ROUND(ELAPSED\_TIME / 1000000 / (CASE  WHEN (EXECUTIONS = 0 OR NVL(EXECUTIONS, 1 ) = 1) THEN  1  ELSE  EXECUTIONS  END),  2) > 5 --100 0000微秒=1S  AND S.PARSING\_SCHEMA\_NAME = USER  AND TO\_CHAR(S.LAST\_LOAD\_TIME, 'YYYY-MM-DD') =  TO\_CHAR( SYSDATE, 'YYYY-MM-DD' )  AND S.COMMAND\_TYPE IN (2 , 3, 5, 6 , 189)  ORDER BY "执行时间'S'" DESC; |

## 根据sql\_id查询执行计划

|  |
| --- |
| select \* from table(dbms\_xplan.display\_awr('sql\_id'));  或者  select \* from table(dbms\_xplan.display\_cursor('sql\_id',null,'ADVANCED ALLSTATS LAST PEEKED\_BINDS')); |

## 获取一个SQL绑定的变量

|  |
| --- |
| SELECT t.VALUE\_STRING, t.VALUE\_ANYDATA FROM v$sql\_bind\_capture t WHERE sql\_id = 'sql\_id';  SELECT snap\_id, NAME, position, value\_string, last\_captured, WAS\_CAPTURED FROM dba\_hist\_sqlbind WHERE sql\_id = ' sql\_id ';  SELECT  dbms\_sqltune.extract\_bind(bind\_data, 1).value\_string AS a,  dbms\_sqltune.extract\_bind(bind\_data, 2).value\_string AS b  FROM wrh$\_sqlstat  WHERE sql\_id = 'sql\_id'; |

不一定好使…

## 通过snap\_id查看系统时间

|  |
| --- |
| select \* from DBA\_HIST\_SNAPSHOT ; |

注意：v$开头的视图仅可以用来查询当前连接的数据库节点的信息，如果需要查看集群全局信息，需要查询gv$开头的视图。