

**2019年春季学期  
计算机学院《软件构造》课程**

**Lab 5实验报告**

|  |  |
| --- | --- |
| 姓名 | 张景润 |
| 学号 | 1172510217 |
| 班号 | 1703002 |
| 电子邮件 | [2584363094@qq.com](mailto:2584363094@qq.com) |
| 手机号码 | 18530272728 |

**目录**

[1 实验目标概述 1](#_Toc10322344)

[2 实验环境配置 1](#_Toc10322345)

[3 实验过程 4](#_Toc10322346)

[3.1 Static Program Analysis 4](#_Toc10322347)

[3.1.1 人工代码走查（walkthrough） 4](#_Toc10322348)

[3.1.1.1 代码缩进问题（2998处） 6](#_Toc10322349)

[3.1.1.2 本行字符数X个：一行字符数超过100个 6](#_Toc10322350)

[3.1.1.3 javadoc第一句缺少一个结束日期 7](#_Toc10322351)

[3.1.1.4 Javadoc缺少 7](#_Toc10322352)

[3.1.1.5 名称X必须匹配表达式‘X’ 7](#_Toc10322353)

[3.1.1.6 导入语句X字典顺序错误 8](#_Toc10322354)

[3.1.1.7 其他一些错误 9](#_Toc10322355)

[3.1.1.8 最终结果（解决所有规范问题） 10](#_Toc10322356)

[3.1.2 使用CheckStyle和SpotBugs进行静态代码分析 10](#_Toc10322357)

[3.2 Java I/O Optimization 11](#_Toc10322358)

[3.2.1 新功能-持久化储存系统信息 11](#_Toc10322359)

[3.2.2 多种I/O实现方式 14](#_Toc10322360)

[3.2.2.1 实现三个I/O方式+具体实现方法 14](#_Toc10322361)

[3.2.2.2 strategy设计模式实现策略之间的切换 16](#_Toc10322362)

[3.2.3 多种I/O实现方式的效率对比分析 16](#_Toc10322363)

[3.2.3.1 收集I/O时间的方式 16](#_Toc10322364)

[3.2.3.2 表格方式对比性能 17](#_Toc10322365)

[3.2.3.3 图形方式对比性能 17](#_Toc10322366)

[3.3 Java Memory Management and Garbage Collection (GC) 18](#_Toc10322367)

[3.3.1 使用-verbose:gc参数 18](#_Toc10322368)

[3.3.2 用jstat命令行工具的-gc和-gcutil参数 20](#_Toc10322369)

[3.3.3 使用jmap -heap命令行工具 20](#_Toc10322370)

[3.3.4 使用jmap -histo命令行工具 21](#_Toc10322371)

[3.3.5 使用jmap -clstats命令行工具 22](#_Toc10322372)

[3.3.6 使用jmap -permstat命令行工具 22](#_Toc10322373)

[3.3.7 使用JMC/JFR、jconsole或VisualVM工具（我选择VisualVM工具） 23](#_Toc10322374)

[3.3.8 分析垃圾回收过程 24](#_Toc10322375)

[3.3.9 配置JVM参数并发现优化的参数配置 25](#_Toc10322376)

[3.4 Dynamic Program Profiling 25](#_Toc10322377)

[3.4.1 使用VisualVM进行CPU Profiling 26](#_Toc10322378)

[3.4.2 使用VisualVM进行Memory profiling 29](#_Toc10322379)

[3.5 Memory Dump Analysis and Performance Optimization 30](#_Toc10322380)

[3.5.1 内存导出 30](#_Toc10322381)

[3.5.2 使用MAT分析内存导出文件 31](#_Toc10322382)

[3.5.3 发现热点/瓶颈并改进、改进前后的性能对比分析 35](#_Toc10322383)

[3.5.4 在MAT内使用OQL查询内存导出 36](#_Toc10322384)

[3.5.4.1 CircularOrbit 的所有对象实例 36](#_Toc10322385)

[3.5.4.2 大于特定长度 n 的 String 对象：我选择长度为10 37](#_Toc10322386)

[3.5.4.3 大于特定大小的任意类型对象实例：我选择大小为100 000 38](#_Toc10322387)

[3.5.4.4 PhysicalObject（及其子类）对象实例的数量和总占用内存大小 38](#_Toc10322388)

[3.5.4.5 所有包含元素数量大于 100 的 Collections 实例 39](#_Toc10322389)

[3.5.4.6 我感兴趣的其他查询 40](#_Toc10322390)

[3.5.5 观察jstack导出程序运行时的调用栈 41](#_Toc10322391)

[3.5.6 使用设计模式进行代码性能优化 43](#_Toc10322392)

[3.6 Git仓库结构 44](#_Toc10322393)

[4 实验进度记录 45](#_Toc10322394)

[5 实验过程中遇到的困难与解决途径 46](#_Toc10322395)

[6 实验过程中收获的经验、教训、感想 46](#_Toc10322396)

[6.1 实验过程中收获的经验和教训 46](#_Toc10322397)

[6.2 针对以下方面的感受 46](#_Toc10322398)

# 实验目标概述

本次实验通过对 Lab4 的代码进行静态和动态分析，发现代码中存在的不符 合代码规范的地方、具有潜在 bug的地方、性能存在缺陷的地方（执行时间热点、内存消耗大的语句、函数、类），进而使用第 4、7、8 章所学的知识对这些问题加以改进，掌握代码持续优化的方法，让代码既“看起来很美”，又“运行起来 很美”。

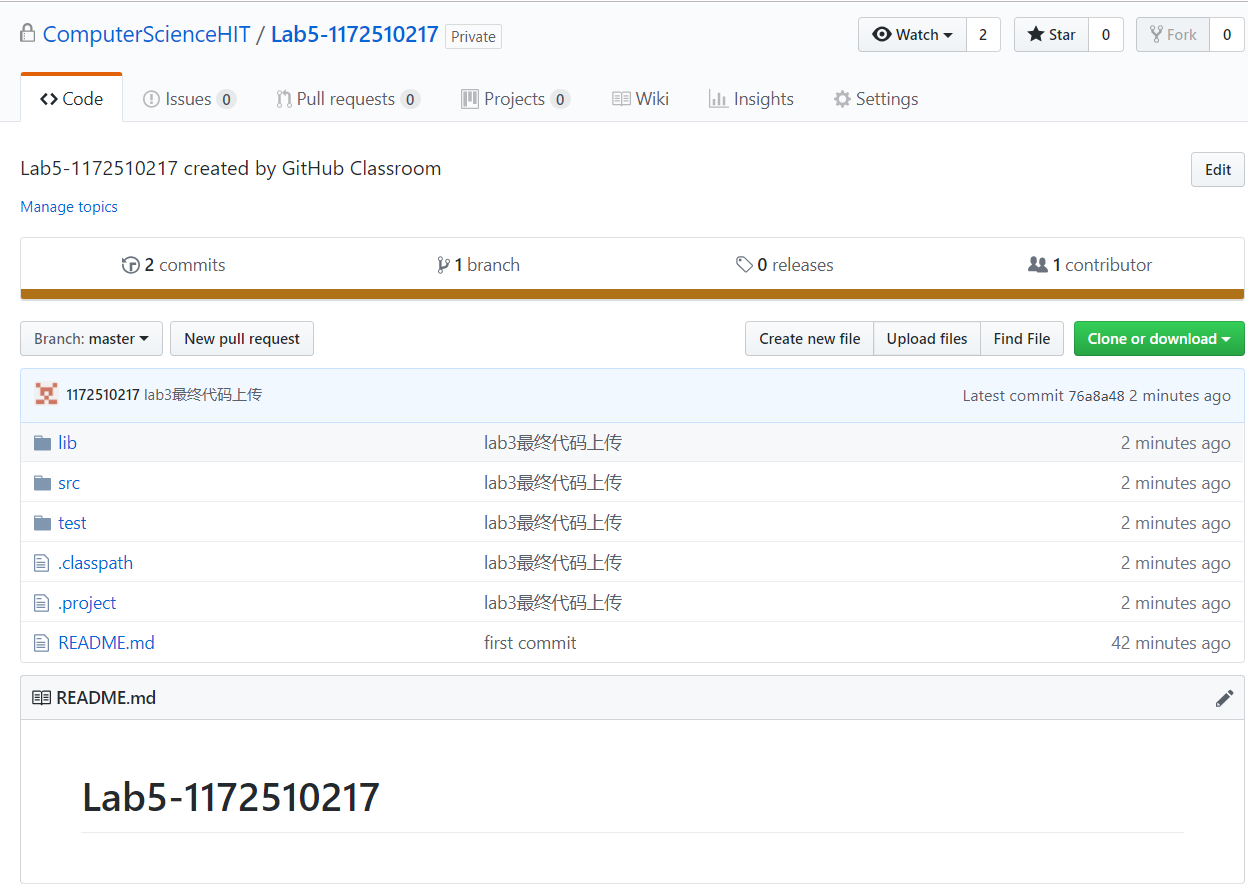
具体训练的技术包括：

* 静态代码分析（CheckStyle和SpotBugs）
* 动态代码分析（Java命令行工具jstat、jmap、jcmd、VisualVM、JMC、JConsole 等）
* JVM内存管理与垃圾回收（GC）的优化配置 
* 运行时内存导出(memory dump)及其分析（Java命令行工具jhat、MAT）
* 运行时调用栈及其分析（Java 命令行工具 jstack）；
* 高性能 I/O
* 基于设计模式的代码调优
* 代码重构

# 实验环境配置

* **配置建立本地仓库**

1. echo "# Lab5-1172510217" >> README.md
2. git init
3. git add README.md
4. git commit -m "first commit"
5. git remote add origin <https://github.com/ComputerScienceHIT/Lab5-1172510217.git>
6. git push -u origin master
7. 将lab4代码推送到远程仓库

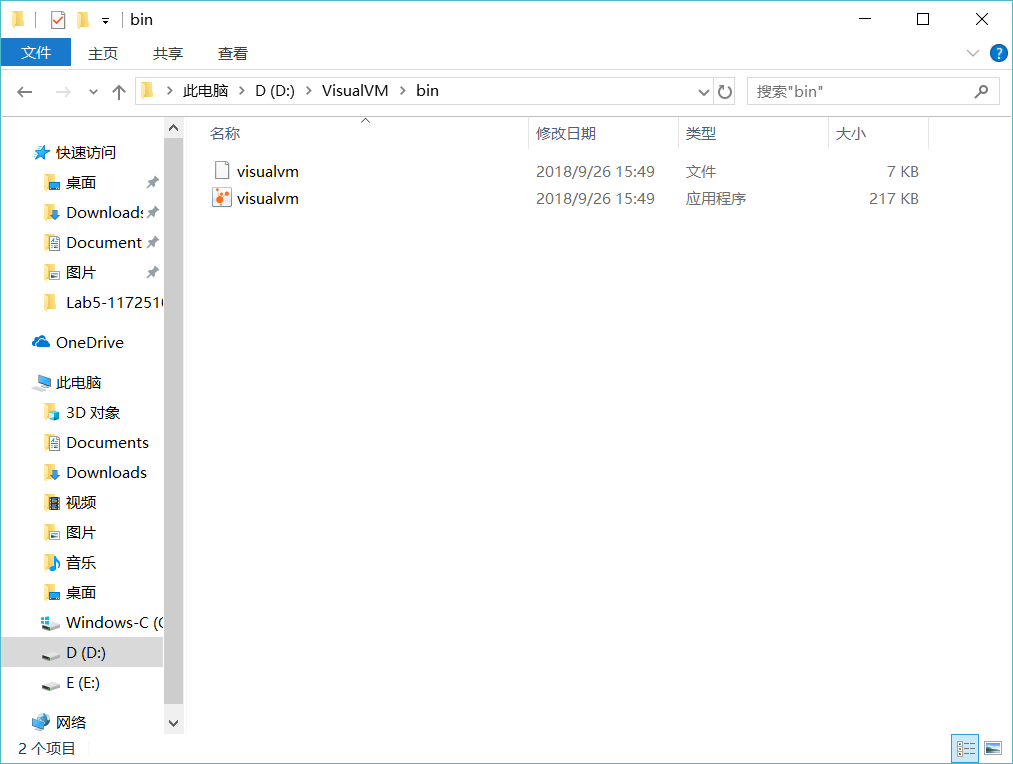


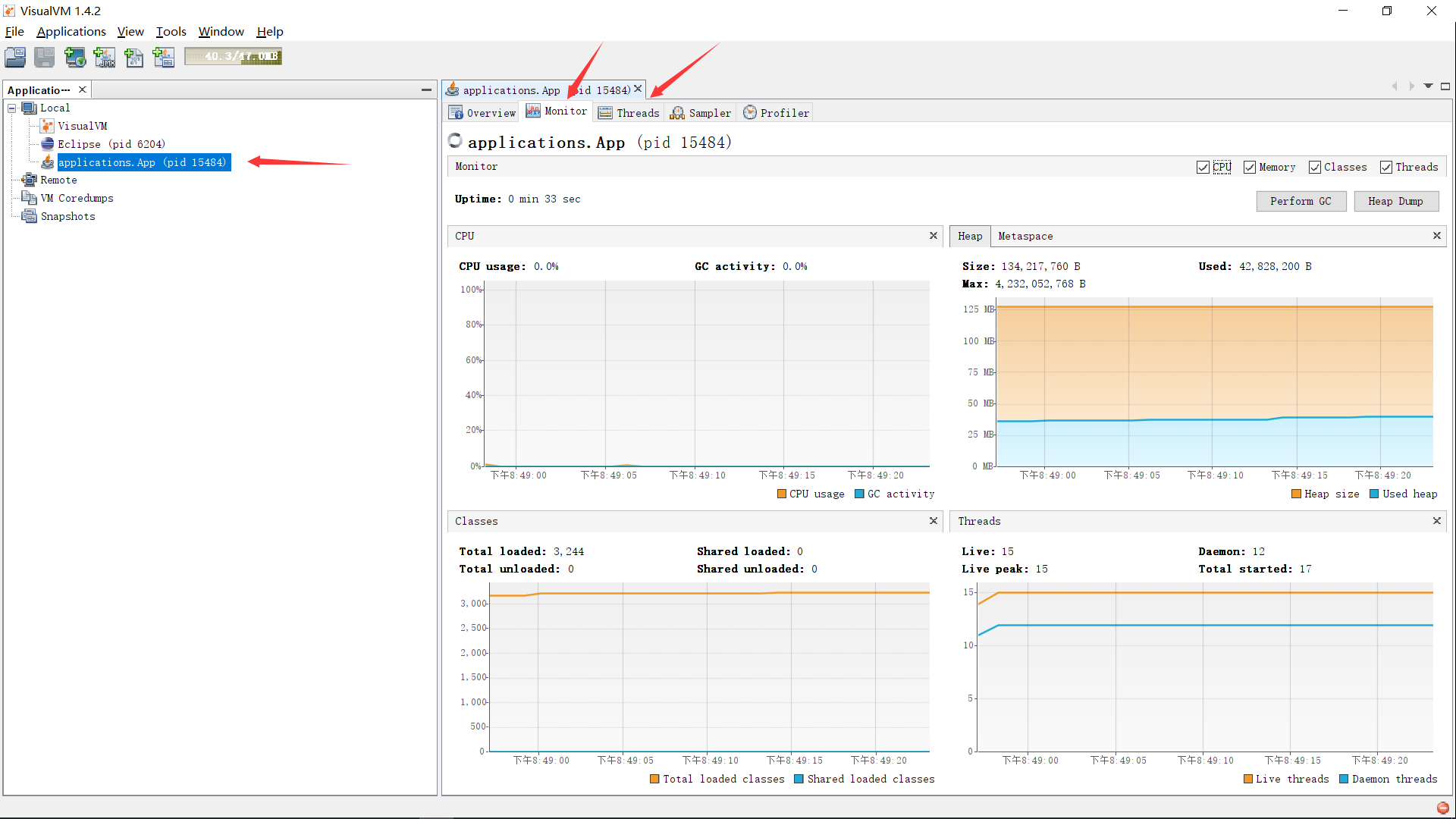
* **本地仓库地址**

<https://github.com/ComputerScienceHIT/Lab5-1172510217>

* 配置VisualVM

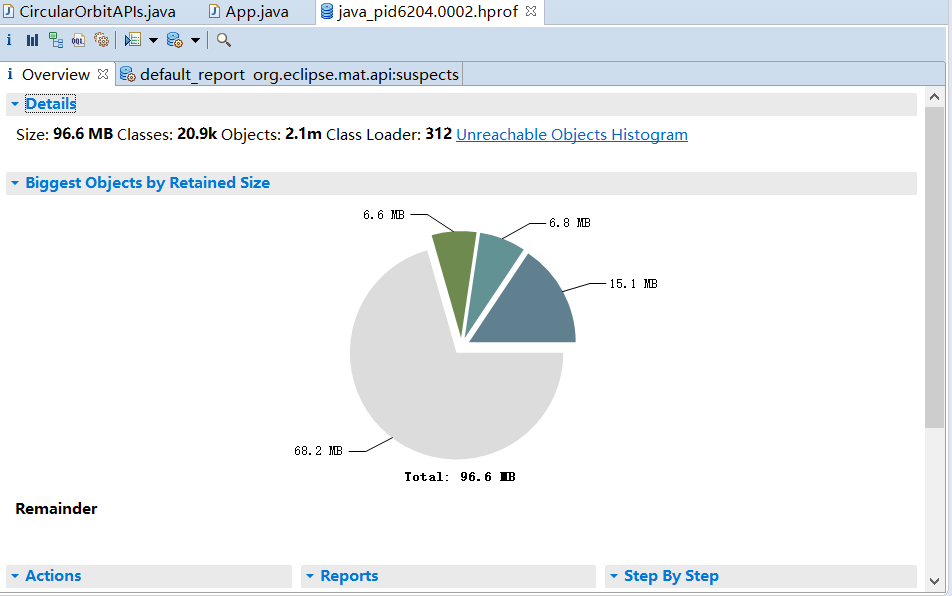
1. 从网站上下载最新版VisualVM压缩包
2. 解压缩此文件
3. 在bin文件中找到visualvm可执行程序文件，如截图
4. 运行此程序，然后运行lab5中的GUI主程序，可以看到我们的App程序，双击即可生成相应的动态详细信息。如截图。





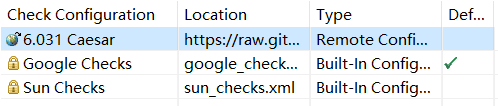
* 配置MemoryAnalyzer。

1. 在Eclipse Market Place中搜索MemoryAnalyzer
2. 点击安装即可，安装完成如截图

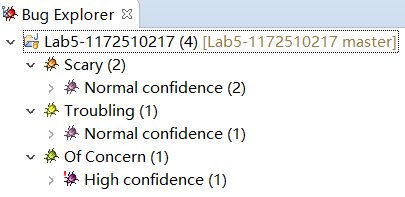


* 配置CheckStyle。

1. 在Eclipse Market Place中搜索CheckStyle
2. 点击安装即可。
3. 选择默认的style：有Googel checks和Sun Checks以及默认的Caesar风格



* 配置SpotBugs。已在lab4中配置完成。



# 实验过程

## Static Program Analysis

### 人工代码走查（walkthrough）

**两点说明**

* 我使用的是Google的代码规范进行checkStyle检测。
* 一共检测到了3511处不规范之处，运行情况如截图。
* 不规范比例分布如下表

|  |  |
| --- | --- |
| 不规范项 | 个数 |
| 总和 | 3511 |
| 代码缩进问题 | 2998 |
| javadoc第一句缺少一个结束日期 | 150 |
| 本行字符数X个：行字符超100个 | 93 |
| Javadoc缺少 | 54 |
| 名称X必须匹配表达式‘X’ | 52 |
| 导入语句X字典顺序错误 | 31 |
| 其他一些错误 | 84 |

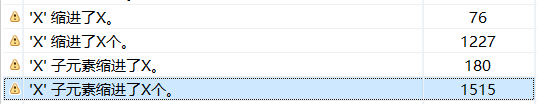




#### 代码缩进问题（2998处）

**1，注：代码缩进问题包括以下4个问题**

* ‘X’缩进了X；
* ‘X’缩进了X个；
* ‘X’子元素缩进了X；
* ‘X’子元素缩进了X个；



**2，解决办法**

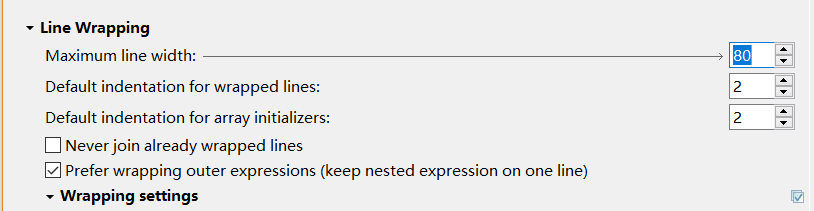
* 将所有的默认缩进换为2空格缩进。采用快捷键CTRL+SHIFT+F进行设置
* 将tab键换位2空格代替。采用快捷键CTRL+SHIFT+F进行设置

**3，解决结果**

* 风格提示处由3511变为464。
* 风格提示窗口不再含有代码缩进问题。

#### 本行字符数X个：一行字符数超过100个

* 解决办法：在eclipse的window->preference->codeStyle->Line Wrapping中进行具体的每行字符数设置即可



* 解决结果：

1. 规范风格提示变为373处
2. 风格提示窗口不再有行字符超过限制的问题

* 具体来看：将下图图1所代表的一类问题的行长度进行默认格式修改，改为图2所示。



图1



图2

#### javadoc第一句缺少一个结束日期

* 解决办法及示例：在注释第一行后面加上字符.即可。截图右边为已修改

* 解决结果

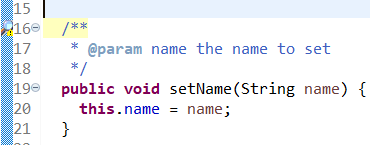
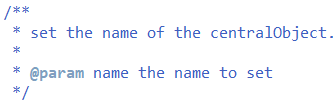
1. 代码风格提示减少了150处
2. 风格提示窗口不再含有javadoc第一句缺少结束日期的提示

#### Javadoc缺少

**此问题包括：缺少摘要javadoc+缺少javadoc**

* 解决办法及示例

1. 针对缺少摘要javadoc：为每一个javadoc添加摘要，修改后如截图

1. 针对缺少javadoc：为每一个方法书写javadoc，修改后如截图



* 解决结果

1. 代码风格提示减少了54处
2. 风格提示窗口不再含有javadoc缺少的提示。

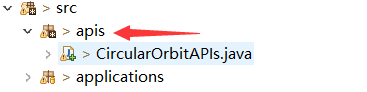
#### 名称X必须匹配表达式‘X’

**此问题包括：包名+方法+变量名+参数名+成员名 不规范**

* **解决办法及示例**

1. 针对包名不规范：右键refactor->rename将所有的包名出现的地方全修改为小写字母格式。

如原来我有名为APIS的包，后来命名重构为apis即可，如截图



1. 针对方法名不规范：pattern：^[a-z][a-z0-9][a-zA-Z0-9\_]\*$。快捷键Shift+Alt+R对方法名进行重命名，如截图所示。





1. 针对变量名不规范：快捷键Shift+Alt+R进行局部变量重命名，如截图（将数组A重命名为a即可）



1. 针对参数名不规范：快捷键Shift+Alt+R进行方法参数重命名，如截图（将数组A重命名为a即可）





1. 针对成员名不规范：pattern：^[a-z][a-z0-9][a-zA-Z0-9]\*$。快捷键Shift+Alt+R进行成员名重命名，如截图。



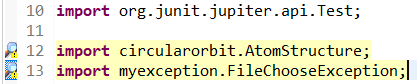


* **解决结果**

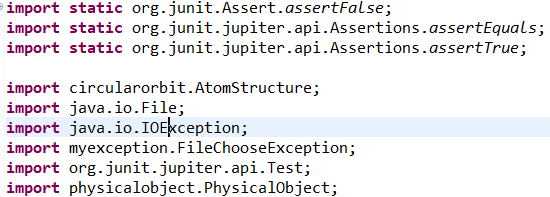
1. 代码规范提示地方变为325处
2. 代码规范窗口不再存在命名不符合规范的问题

#### 导入语句X字典顺序错误

* 解决办法及示例：将顺序调整为正确的顺序即可，如截图



图一：调整前



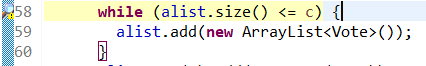
图二：调整后

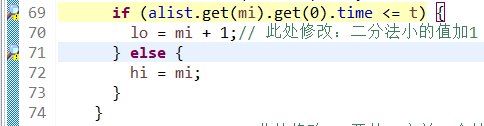
* 解决结果：

1. 代码规范提示地方变为288处
2. 代码规范窗口不再存在导入顺序不合规范的问题

#### 其他一些错误

* while、if、else结构必须使用大括号：结构加上大括号即可



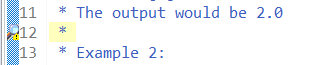


* 静态导入应该与其余的导入组分隔一个空行：加上一个空行即可



* Javadoc注释空行后应有 <p> 标签：执行如下修改（右图为修改后）

格式化操作可以执行：window-preference-codestyle-javadocs-取消勾选Format HTML tags，然后添加标签<p>并对代码执行：source-format即可

* 名称中不能出现超过2个连续大写字母：快捷键重命名即可（右图为修改后）

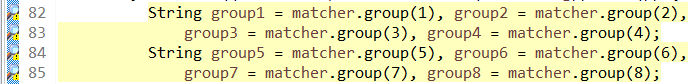
 

* 不应使.\*形式的导入：修改导入为具体的一个个的导入（下面的图为修改后）





* 每个变量定义必须在声明处，且在同一行：将代码修改为一行一个变量声明





* 大括号与后面注释间应有空格：注释前大括号后添加空格（修改后为下图右）



* 变量声明及其使用之间不超过三行：变量在使用时的前一行定义（两张图片下面的一张为修改后）





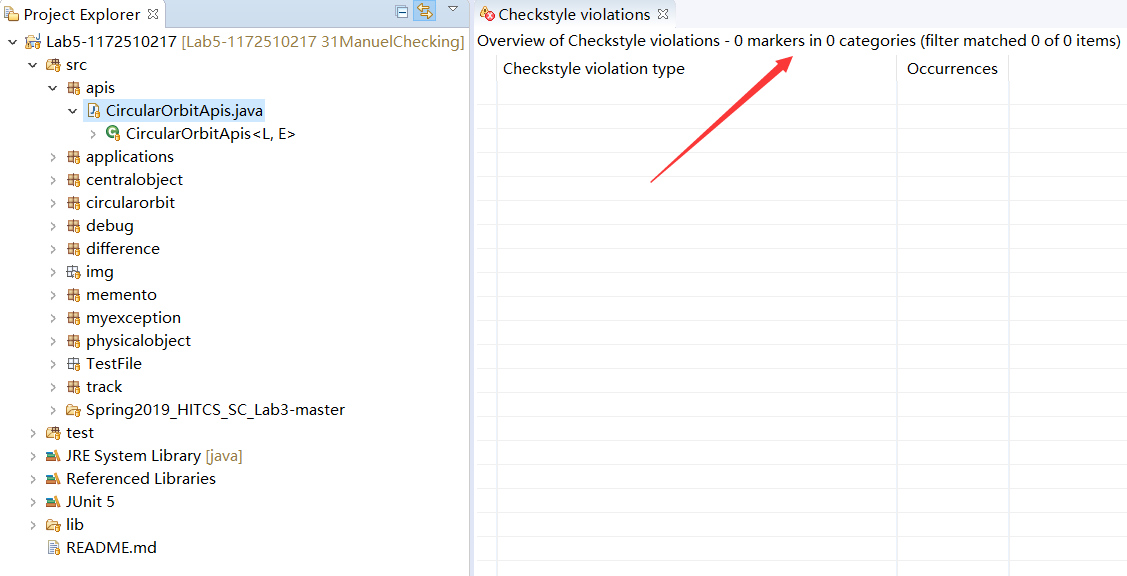
* @标签后应有非空说明：对每一个@标签添加说明（修改后如较下面的图片）





#### 最终结果（解决所有规范问题）

* 下图为最终checkstyle截图



* 合并分支+推送分支

1. 在31ManuelChecking分支输入指令切回主分支：git checkout master
2. 合并本地分支：git merge 31ManuelChecking
3. 推送分支：git push

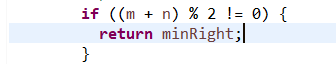
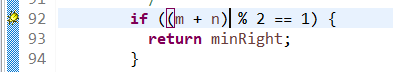
### 使用CheckStyle和SpotBugs进行静态代码分析

* 关于CheckStyle

代码风格已在上文3.1.1进行了详细的介绍和讲解。

* 关于SpotBugs

1. 取余对负数不成立：应改为 (m + n)%!=1（修改后为右图）



1. 对象判断不可用= =：应用equals方法（修改后如右图）





1. 不使用Double.compareTo方法：使用Double的compare方法（下图）



1. 不可以使用直接new String()：改为String string = “”（如右图）

**解决了部分潜在的bug（约10个！！！）**

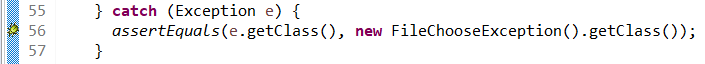


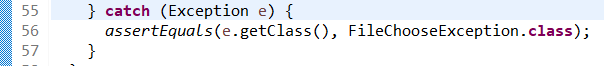
1. 不建议使用System.exit：删除掉使用其他策略



1. 分配一个对象，只是为了得到它的类：改用类名.class获取类（如下图）

**解决了大部分潜在的bug（约40个！！！）**





1. String字符串连接最好用StringBuilder：使用Stringbuilder即可（如下图）





* 分支合并+分支推送

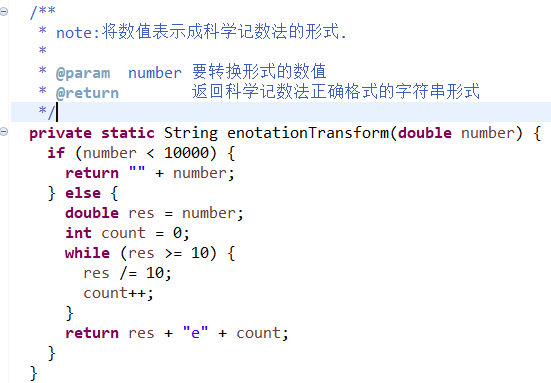
1. 在31ToolChecking分支输入指令切回主分支：git checkout master
2. 合并本地分支：git merge 31ToolChecking
3. 推送分支：git push

## Java I/O Optimization

### 新功能-持久化储存系统信息

* 在circularorbit包中添加新功能：主要是遍历

1. 在StellarSystem中添加新方法：saveSystemInfoInFile和enotationTransform
2. 其中enotationTransform：将编译器存储的科学记数法表示的数字转化为语法要求的格式；

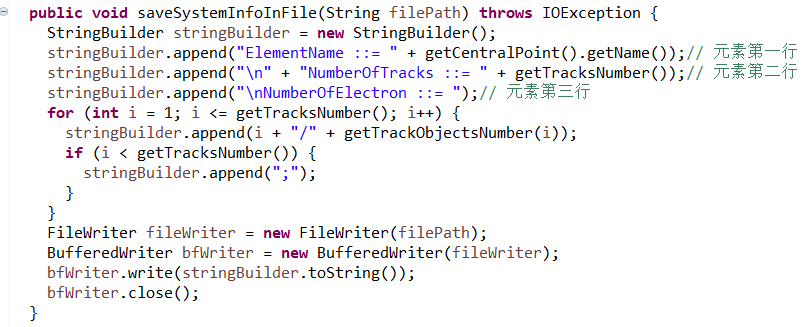


1. savaSystemInfoInFile：将系统按照lab3语法格式输出。



1. 在AtomStructure添加新方法：saveSystemInfoInFile

savaSystemInfoInFile：将系统按照lab3语法格式输出。



1. 在SocialNetworkCircle添加新方法：saveSystemInfoInFile

savaSystemInfoInFile：将系统按照lab3语法格式输出。



* 分支合并+分支推送

1. 在32AddOutput分支输入指令切回主分支：git checkout master
2. 合并本地分支：git merge 32AddOutput
3. 推送分支：git push

### 多种I/O实现方式

#### 实现三个I/O方式+具体实现方法

* **实现方法**

1，设计了一个抽象类public interface IoStrategy {，其中有6个方法，分别对应3个系统的读和写。

**public** **void** readFileAndCreateSystem(StellarSystem stellarSystem, File file)

**public** **void** readFileAndCreateSystem(AtomStructure atomStructure, File file)

**public** **void** readFileAndCreateSystem(SocialNetworkCircle socialNetworkCircle, File file)

**public** **void** saveSystemInfoInFile(StellarSystem stellarSystem, String filePath)

**public** **void** saveSystemInfoInFile(AtomStructure atomStructure, String filePath)

**public** **void** saveSystemInfoInFile(SocialNetworkCircle socialNetworkCircle, String filePath)

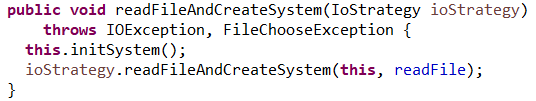
2，设计了3个具体的IO策略，实现该抽象类

**public** **class** NioStrategy **implements** IoStrategy

**public** **class** BufferedIoStrategy **implements** IoStrategy

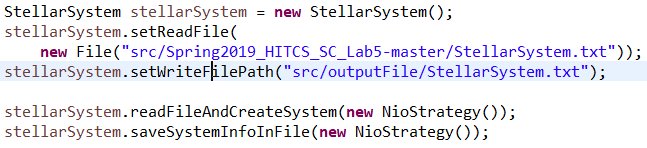
**public** **class** ScannerIoStrategy **implements** IoStrategy

3，在CircularOrbit的3个子类中各实现读和写的方法，传入参数为IoStrategy，如截图。





1. 客户端具体的调用方法如截图

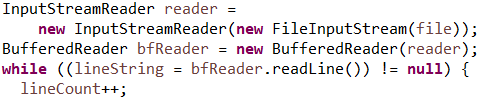


行星系统选择Nio的文件读取写入方式

* BufferedIoStrategy

此方法是前面实验中一直在使用的方法。读取思想是：按行读取处理数据。典型的操作如截图。

基本写入思想是：使用StringBuilder的append方法，构建完成输出字符串后，整个的输出。

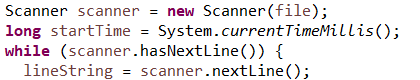


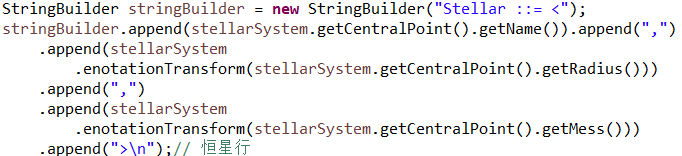


* ScannerIoStrategy

基本读取思想是：按行读取数据+按行处理数据，典型操作如截图。

基本写入思想是：使用StringBuilder的append方法，构建完成输出字符串后，整个的输出。

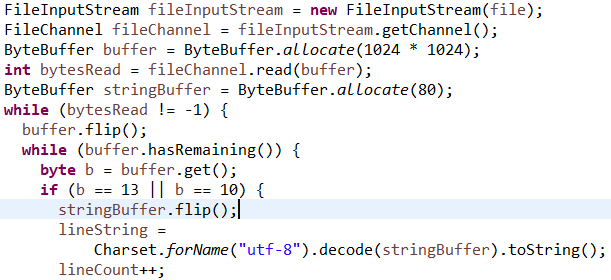


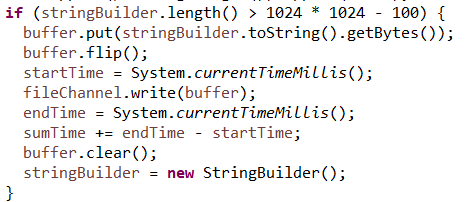


* NioStrategy

基本读取思想：按行读取处理数据。这个按行读取数据的方法与上述两个读取方式有差异，因为是字节读取，所以需要判断换行符的存在以及文本结尾存在。

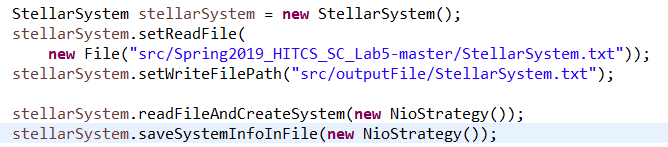
基本写入思想：根据buffer可用容量进行输出，当构建的StringBuilder的字符串字节长度即将大于容量时，进行写出。





#### strategy设计模式实现策略之间的切换

客户端通过传入具体的策略即可实现不同策略的的切换,如截图



### 多种I/O实现方式的效率对比分析

#### 收集I/O时间的方式

* 通过注入少量代码的方式收集时间:注入终端输出代码通过计算时间差得到读写的近似时间,如截图.





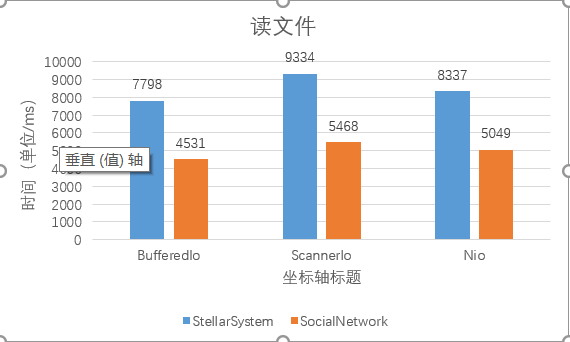
* 为了保证数据的准确性,进行了多次实验,选取其中数据波动不太明显的三组作为参考组
* 具体的执行结果如截图.

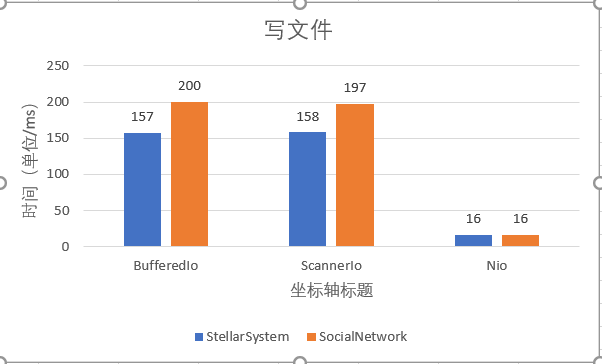
#### 表格方式对比性能

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | StellarSystem.txt | SocialNetworkCircle.txt |
| BufferedIoStrategy | 读文件 | 7798ms | 4531ms |
| 写文件 | 157ms | 200ms |
| ScannerIoStrategy | 读文件 | 9334ms | 5468ms |
| 写文件 | 158ms | 197ms |
| NioStrategy | 读文件 | 8337ms | 5049ms |
| 写文件 | 16ms | 16ms |

#### 图形方式对比性能

总体来说:读取文件由于是按行处理,所以说三者差别不大;但是写出文件可以明显地看到Nio策略具有明显的优势.





## Java Memory Management and Garbage Collection (GC)

**常用：在控制台输入jps可以获取进程pid，便于下面一系列命令的输入**

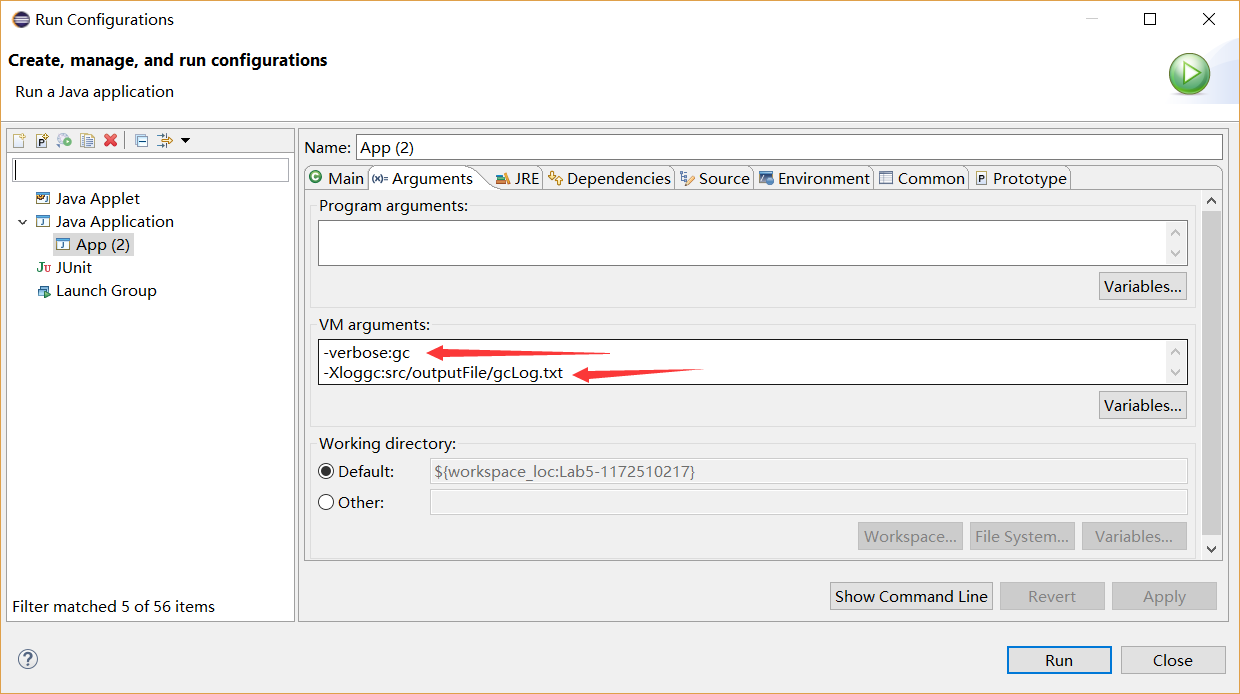
### 使用-verbose:gc参数

* **设置相关参数**

1，-verbose:gc参数

2，-Xloggc:src/outputFile/gcLog.txt参数

3，-XX:+PrintGCDetails参数

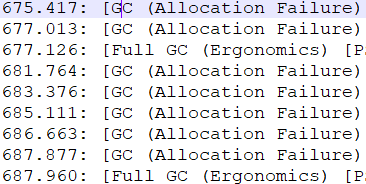
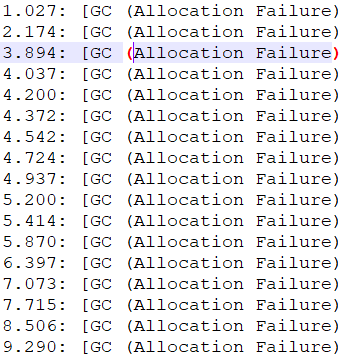


* **分析输出gc文本结果**

1. Minor GC和Full GC发生的频率

Minor GC在8.2s的时间内发生了17次，大约2次/s；

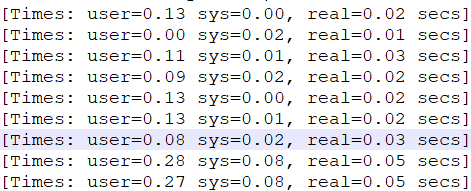
Full GC大于10s进行一次，大约0.1次/s。



1. 两种GC单次耗费的时间

Minor GC单次耗费时间不唯一，从0.01s到0.08s不等。如截图

Full GC单词耗费时间也不唯一，第一次耗费时间较少为0.02s，第二次为0.45s，第三次为0.96s，如截图



Full GC







1. 每次GC前后heap中各区域占用情况的变化

任取一次Minor GC，分析其堆中各区域变化情况（下面加深色的为源数据）

[PSYoungGen: 64483K->2021K(67072K)] 105303K->44811K(186880K), 0.0027480 secs]

年轻代内存使用由64483K降低到2021K，总空间大小为67072K；

整个堆内存的使用由105303K降低到44811K，总空间大小为186880K

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| GC前 | 年轻代已用 | 年轻代未用 | 老年代已用 | 老年代未用 |
| 单位K | 64483 | 2589 | 40820 | 78988 |
| GC后 | 年轻代已用 | 年轻代未用 | 老年代已用 | 老年代未用 |
| 单位K | 2021 | 65051 | 42790 | 77018 |

任取一次Full GC，分析其堆中各区域的变化情况（下面加深色的为源数据）

[PSYoungGen:224K->0K(579584K)]

[ParOldGen:294055K->293506K(487424K)] 294279K->293506K(1067008K),

[Metaspace: 13462K->13462K(1060864K)],1.0579956 secs]

年轻代内存使用由224K降到0K，总空间大小为579584K；

老年代内存使用由294055K降到293506K，总空间大小为487424K；

堆总内存由294279K降到293506K，总空间大小为1067008K；

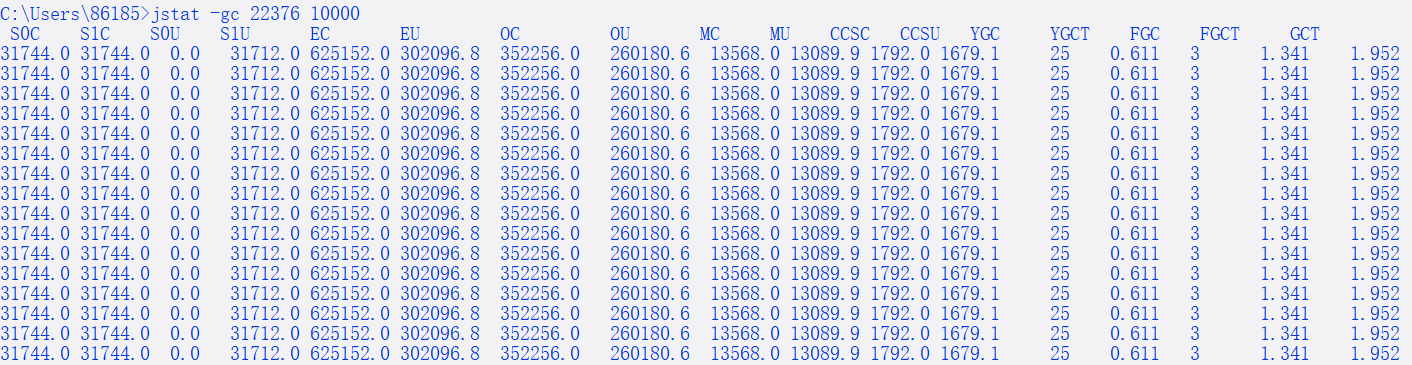
Metaspace内存使用不变为13462K，总空间大小为1060864K。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| GC前 | 年轻代已用 | 年轻代未用 | 老年代已用 | 老年代未用 | Metaspace已用 | Metaspace未用 |
| 单位K | 224 | 579360 | 294055 | 193369 | 13462K | 13462K |
| GC后 | 年轻代已用 | 年轻代未用 | 老年代已用 | 老年代未用 | Metaspace已用 | Metaspace未用 |
| 单位K | 0 | 579584 | 293506 | 193918 | 1047402 | 1047402 |

### 用jstat命令行工具的-gc和-gcutil参数

* 使用工具的方法：jstat -gc 进程pid 监控周期(ms)

我的输入是：jstat -gc 22376 10000（每隔10s输出一次gc情况）



其中各部分代表的含义如下

S0C：第一个幸存区的大小，S1C：第二个幸存区的大小

S0U：第一个幸存区的使用大小，S1U：第二个幸存区的使用大小

EC：伊甸园区的大小，EU：伊甸园区的使用大小

OC：老年代大小，OU：老年代使用大小

MC：方法区大小，MU：方法区使用大小

CCSC:压缩类空间大小，CCSU:压缩类空间使用大小

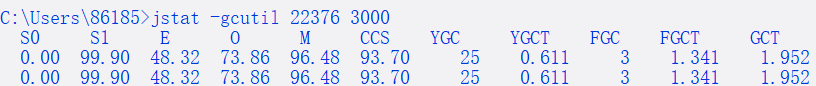
YGC：年轻代垃圾回收次数，YGCT：年轻代垃圾回收消耗时间

FGC：老年代垃圾回收次数，FGCT：老年代垃圾回收消耗时间

GCT：垃圾回收消耗总时间

* 使用工具的方法：jstat -gcutil 进程pid 监控周期(ms)

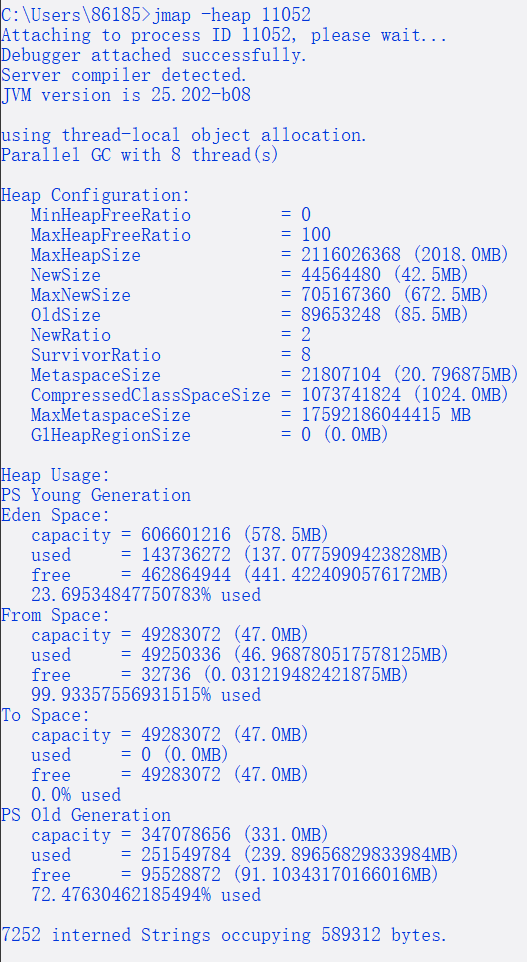
我的输入是：jstat -gcutil 22376 10000（每隔10s输出一次gc情况）



### 使用jmap -heap命令行工具

* 使用工具的方法：jmap -heap 进程pid

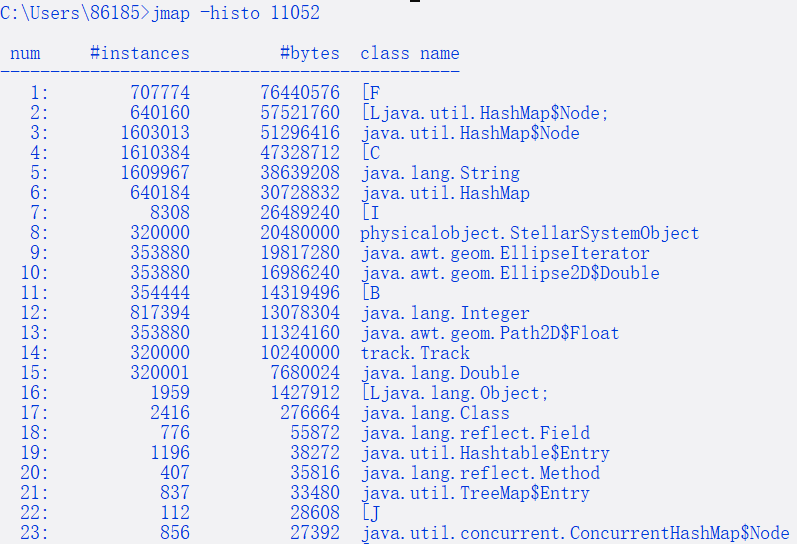
我的输入是：jamp -heap 11052



### 使用jmap -histo命令行工具

* 使用工具的方法：jmap -histo 进程pid

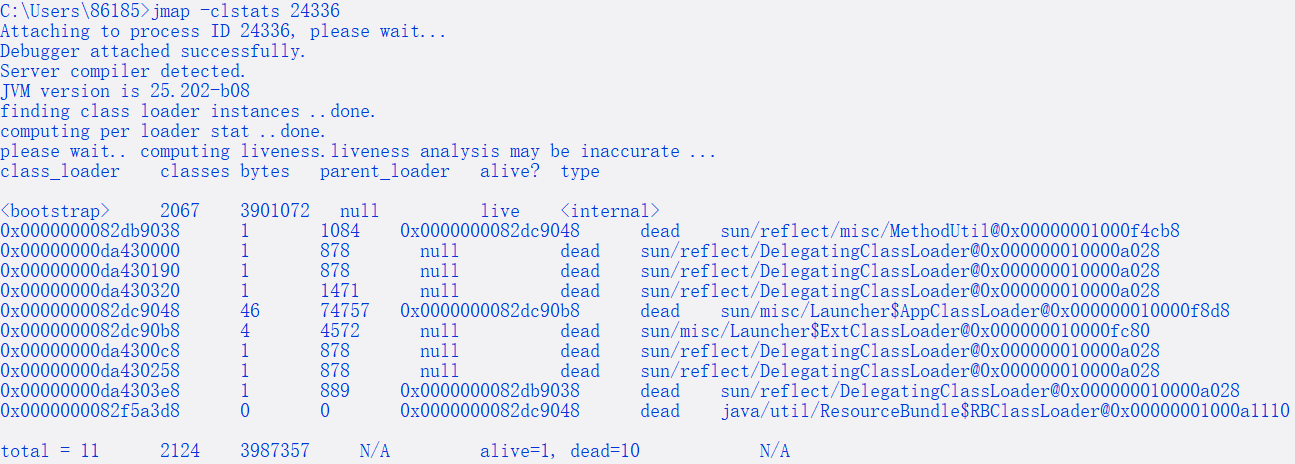
我的输入是：jamp -histo 11052



### 使用jmap -clstats命令行工具

* 使用工具的方法：jmap -clstats进程pid

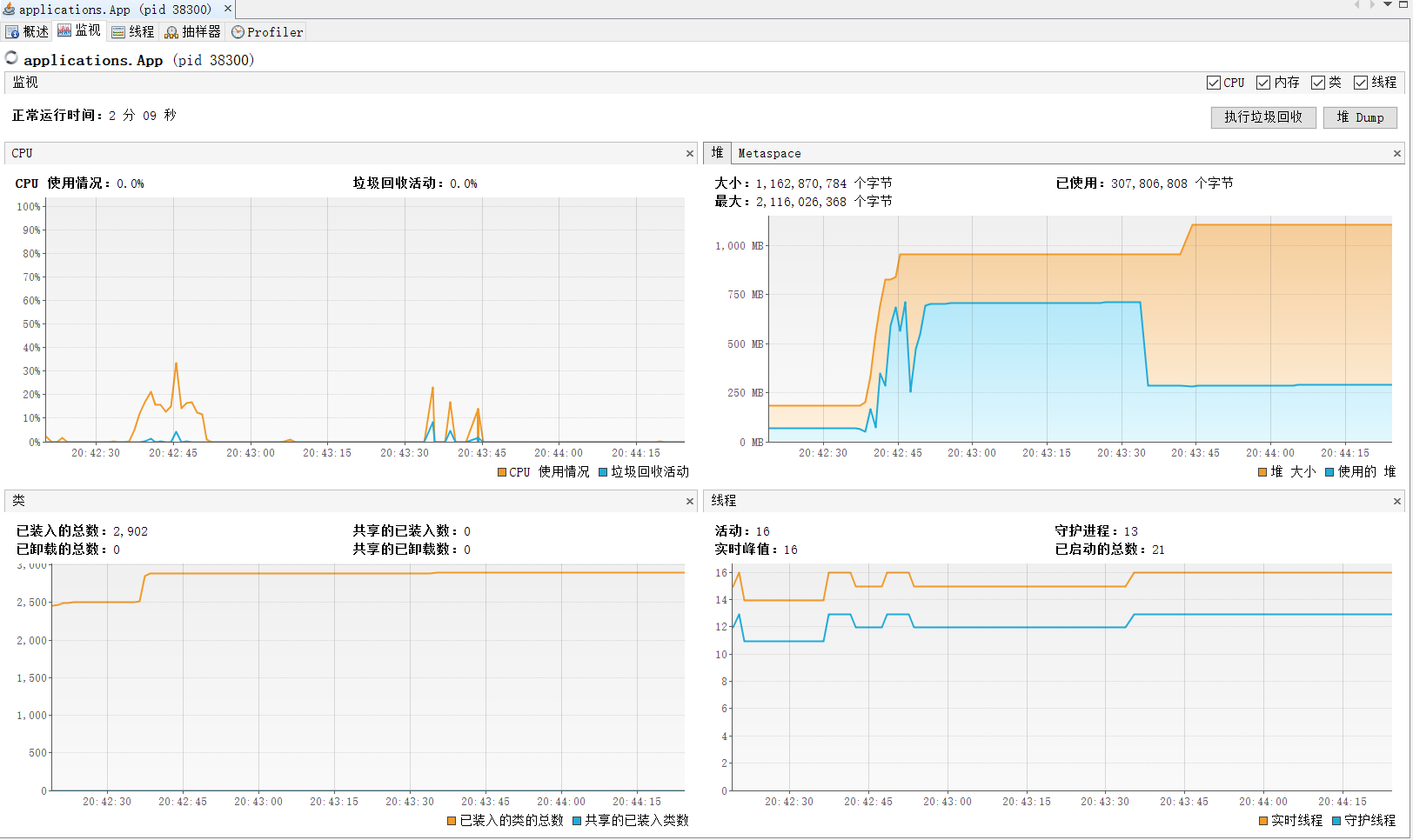
我的输入是：jamp -clstats 11052



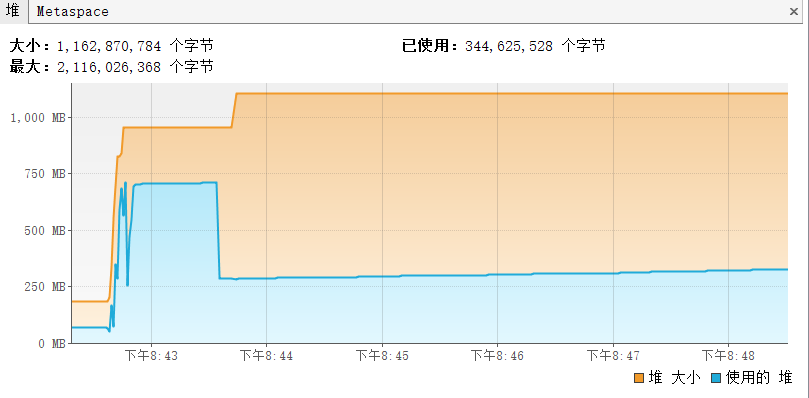
### 使用jmap -permstat命令行工具

此命令为JDK8以前的版本使用，我的JDK版本为JDK8，命令已换为3.3.5节内容。

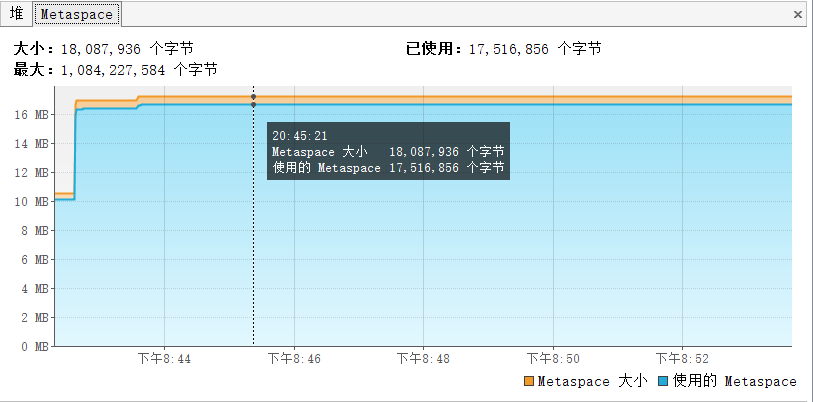
### 使用JMC/JFR、jconsole或VisualVM工具（我选择VisualVM工具）



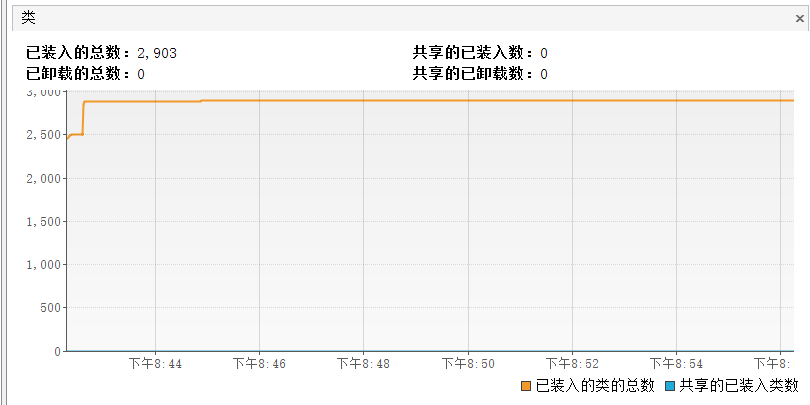
堆内存最大为2116026368字节，当前大小为1162870784字节，已使用344625528字节。



metaspace最大为1084227584字节，当前大小为18087936字节，已使用17516856字节。

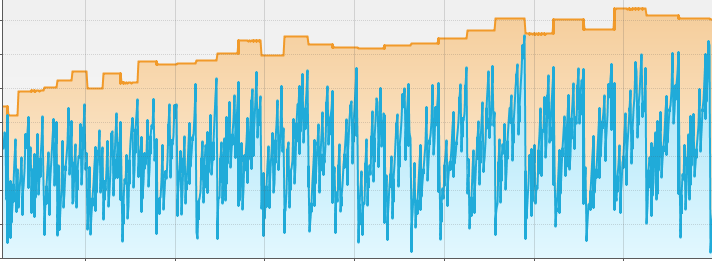


装载的类实例为2903个，如下截图。



### 分析垃圾回收过程

* minor GC频率比较频繁：适当增加年轻代的内存空间大小来降低minor GC的频率。



* 同时发现Full GC也相对比较频繁：由于执行一次Full GC的时间比较短，可以考虑适当增加老年代的内存空间。

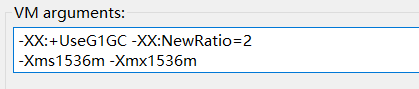
### 配置JVM参数并发现优化的参数配置

* 最终我选择的虚拟机配置如下：

-XX:+UseG1GC -XX:NewRatio=2

-Xms1536m -Xmx1536m

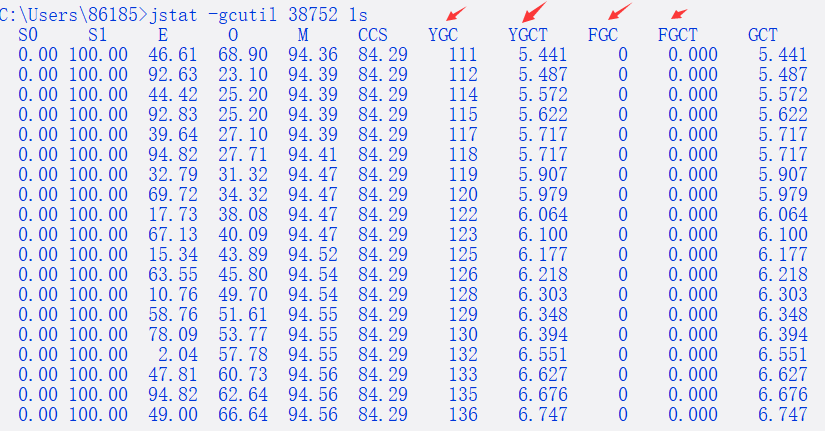
* 选择的垃圾回收算法是G1GC，老年代和新生代的比例为2，总空间初始值为1536M，最大值为1536M。



* 具体表现

在相当长的测试时间内，完全没有发生Full GC的操作；

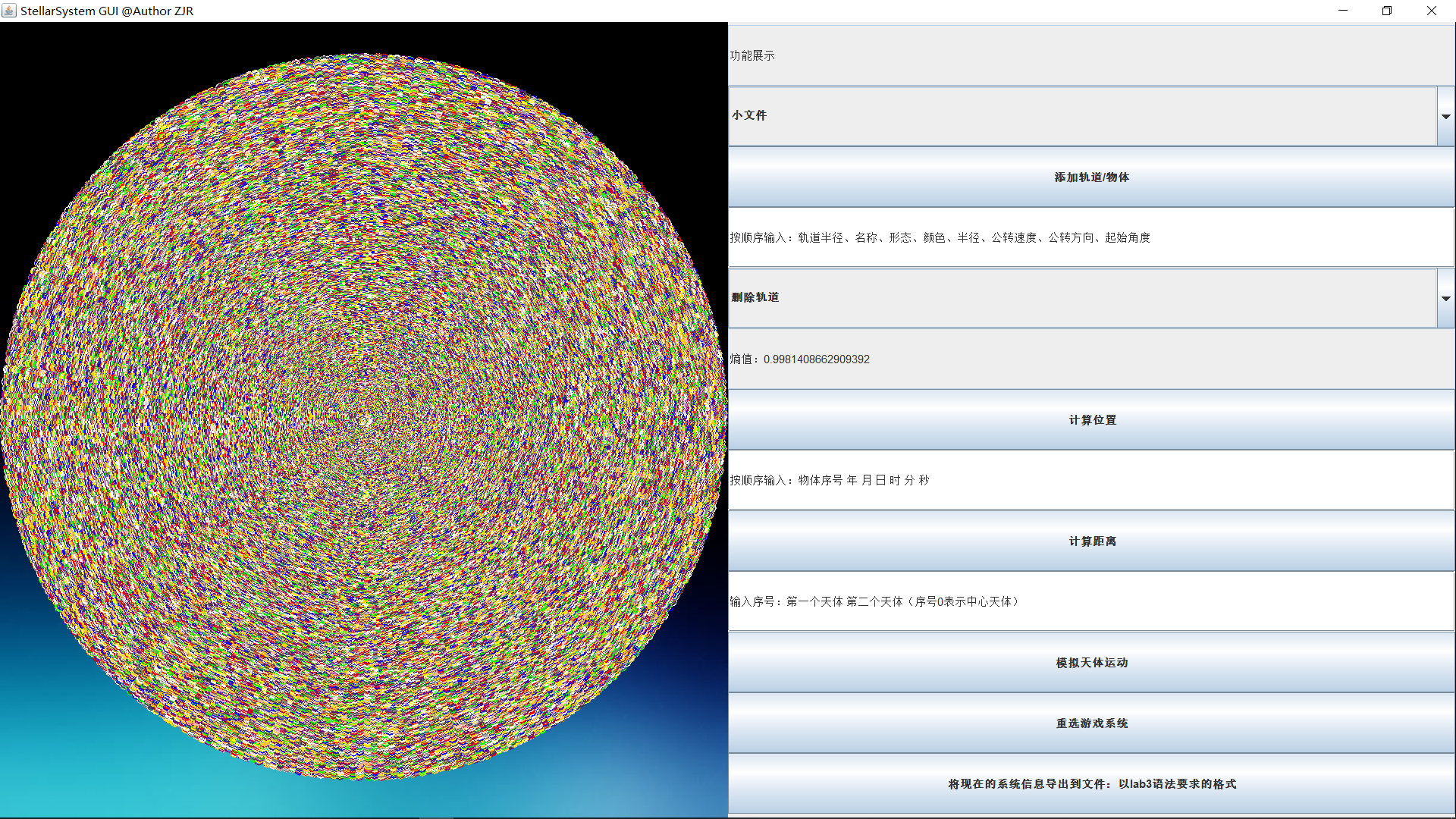
minor GC频率比较合适，且每次时间为50ms左右，较为正常；



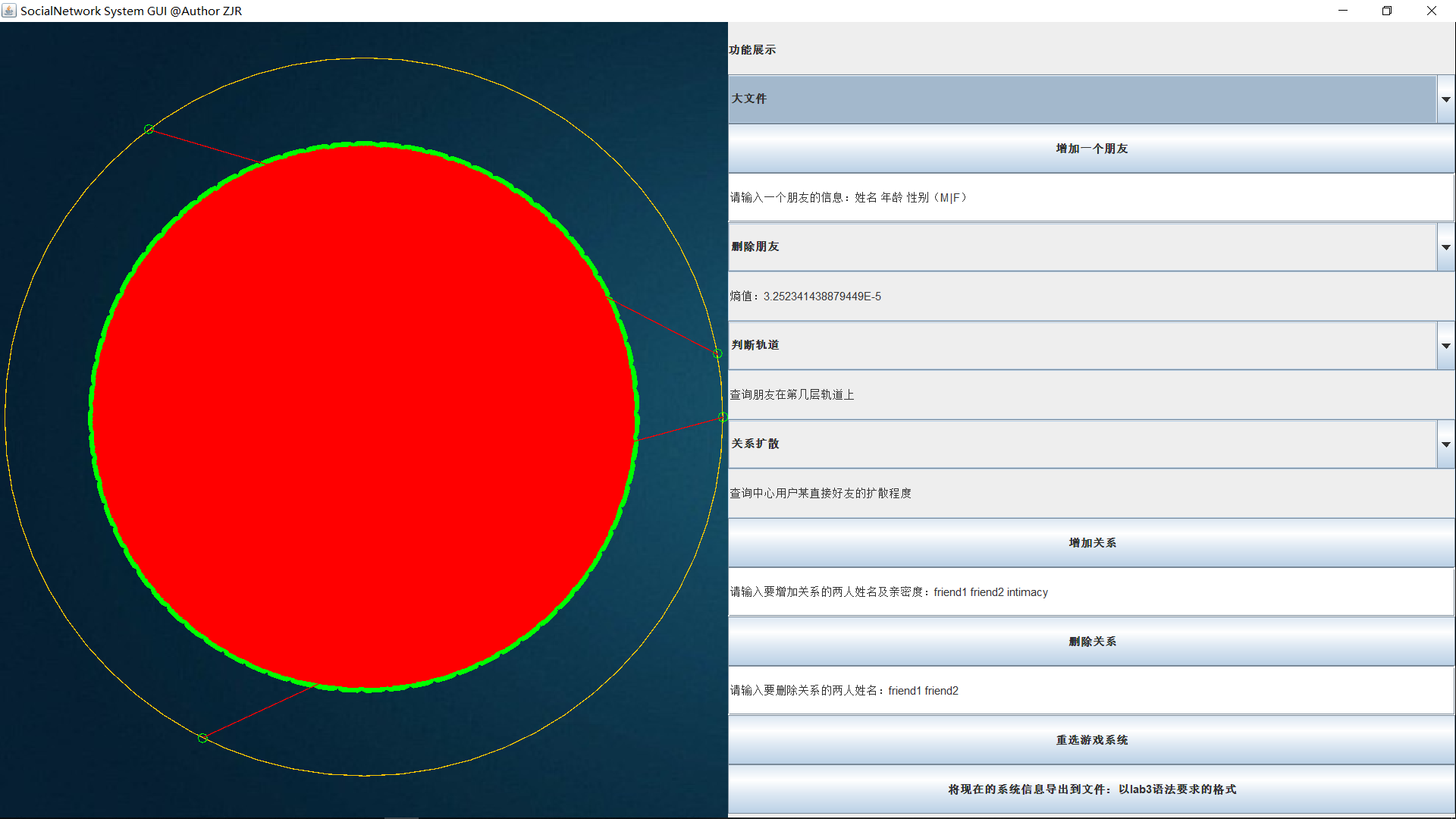
## Dynamic Program Profiling

**综述，下图1是我的超大文件的GUI图形界面，我选择的工具是VisualVM**

**行星系统如下**

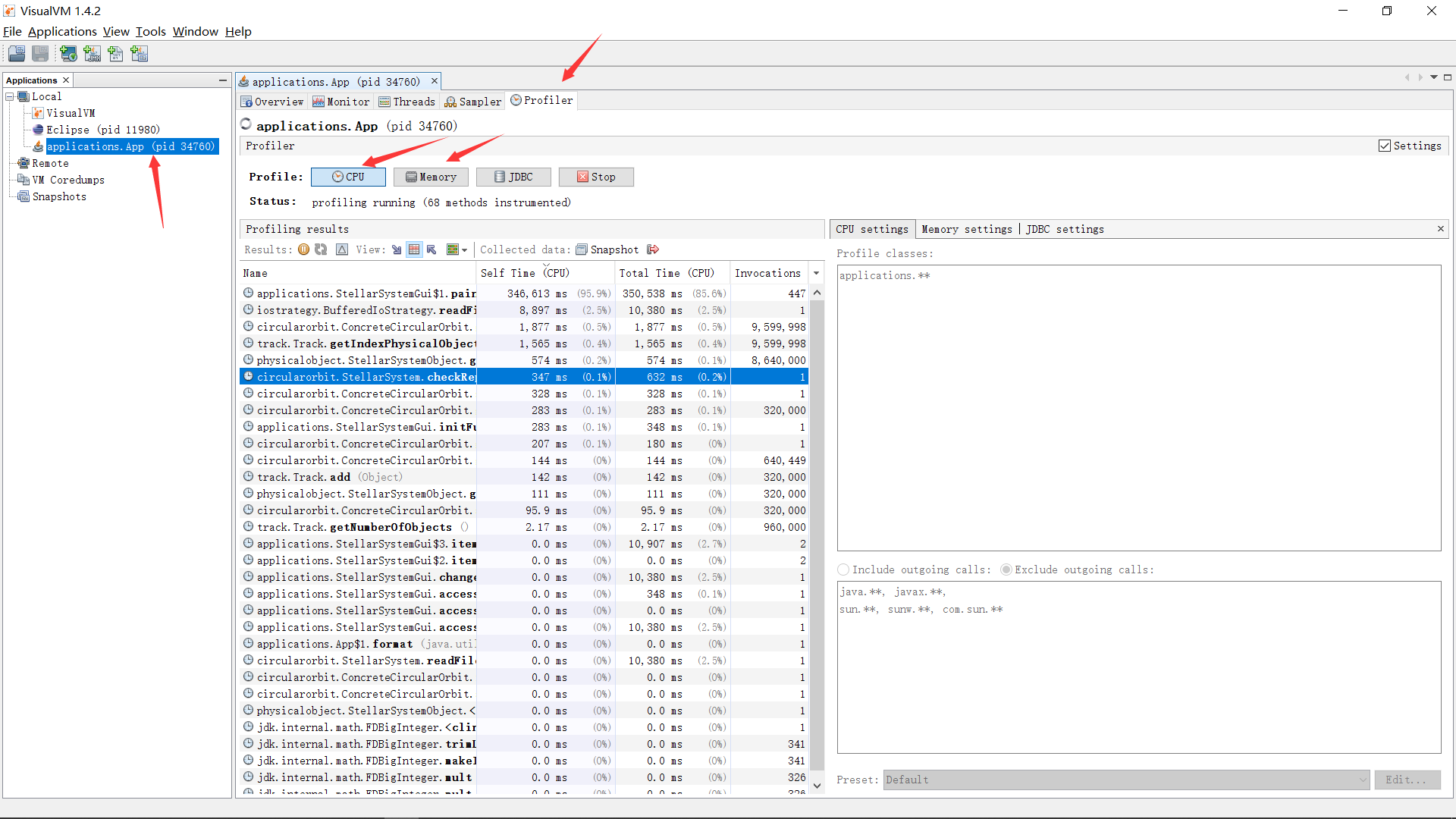


**社交系统如下**



### 使用VisualVM进行CPU Profiling

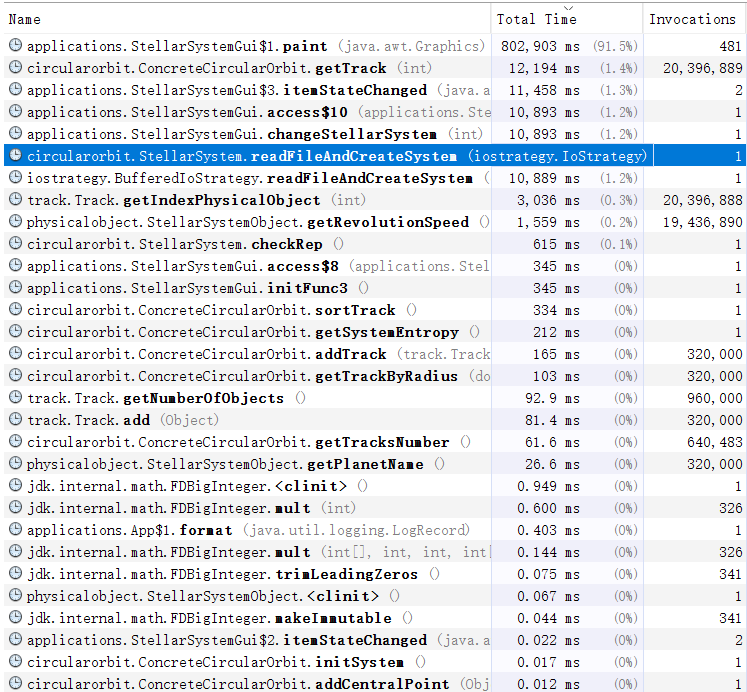
* **VisualVM的CPU Profiling界面**



* **得到的结果分析**

1. 我要监控的类是App类，这是我的客服端主类。

我选择读取文件路径是：src/Spring2019\_HITCS\_SC\_Lab5-master/StellarSystem.txt



1. 图表形式展示执行的方法所耗费的时间、比例

|  |  |  |
| --- | --- | --- |
| paint | 158,253ms | 96.6% |
| getTrack | 2,508ms | 1.5% |
| itemStateChanged | 921ms | 0.6% |
| getIndexPhysicalObject | 571ms | 0.3% |
| sortTrack | 326ms | 0.2% |
| access$8 | 320ms | 0.2% |
| initFunc3 | 320ms | 0.2% |
| getRevolutionSpeed | 301ms | 0.2% |
| getSystemEntropy | 271ms | 0.2% |
| getNumberOfObjects | 51.0ms | 0% |
| getTracksNumber | 25.1ms | 0% |
| App$1.format | 0.328ms | 0% |
| deleteTrack | 0.063ms | 0% |
| access$7 | 0.005ms | 0% |

1. 耗时方法分析
2. 经过分析发现，耗时最多的是GUI画板的paint方法，占据时常96.6%：因为这个画板是实时更新的，所以占比最大，因为他的程序周期是最长的，从用户选择读取这个超大文件开始，就开始运行。
3. 其次是ConcreteCircularOrbit类的getTrack方法。因为在画图的的时候疯狂调用获取轨道的方法，所以占据时长较长。
4. itemStateChanged耗时其次。是正常行为，因为接受来自用户端的鼠标输入。

**相关的解决办法**

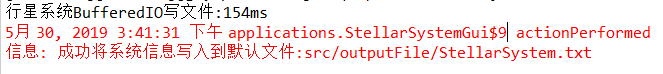
1. 发现画板是实时更新的。

做了相应的修改：画板只有在改变时才会更新。

C:\Users\86185\AppData\Roaming\Tencent\Users\2584363094\QQ\WinTemp\RichOle\LE~1KKVM1TY``Z0ODK`IS3O.png

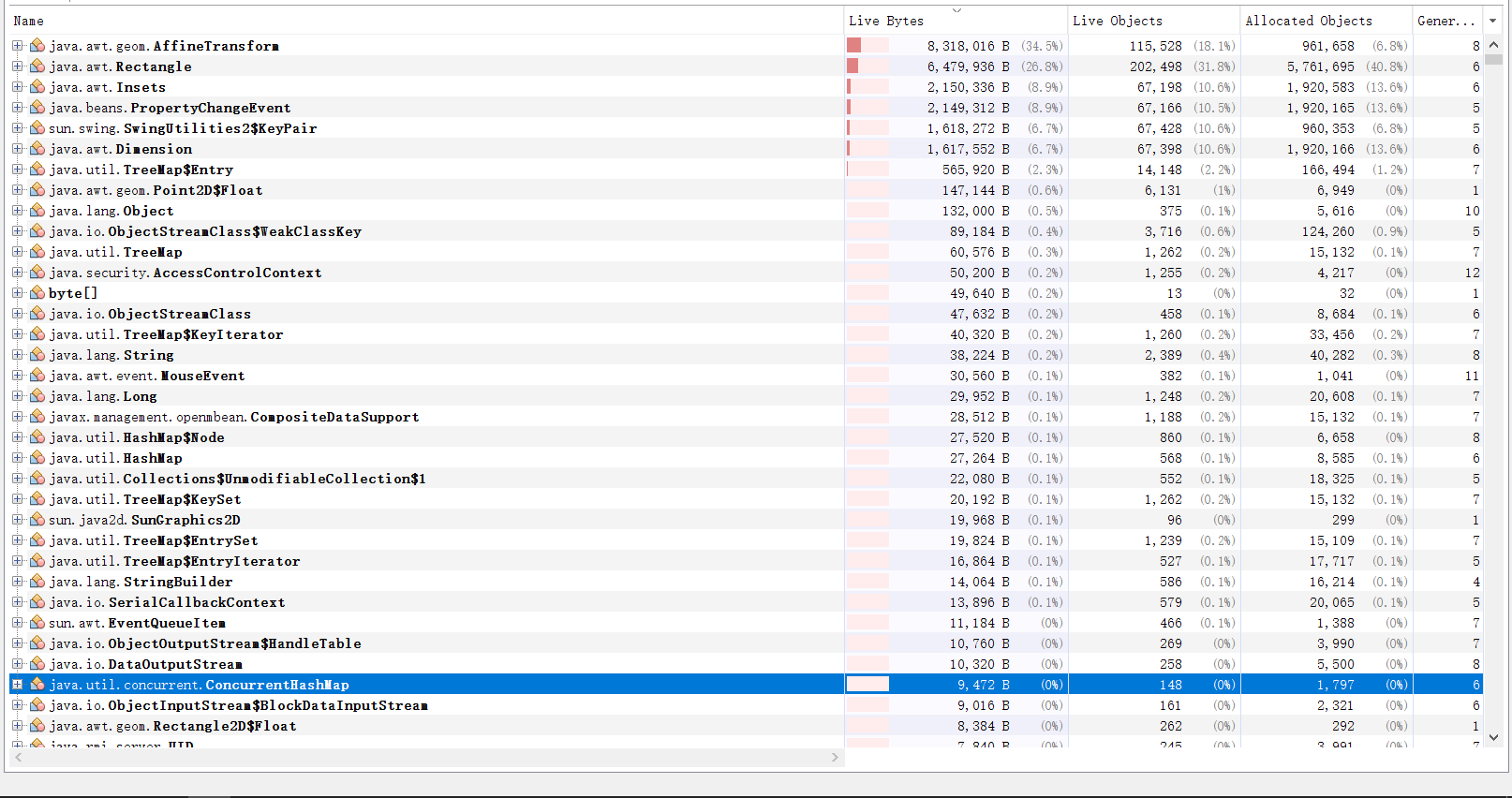
b）进行a）的修改后，发现将信息储存耗时较长。

原因是因为忘记了将3.2节做的IO优化分支合并到当前master上。合并分支后发现saveSystemInfoInFile耗时大幅下降，直观表现如终端截图。



### 使用VisualVM进行Memory profiling

* **VisualVM的memory的profile界面截图**

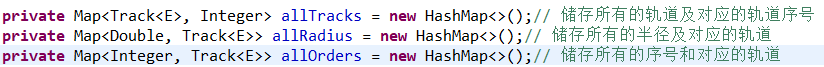


* **得到的结果统计**

|  |  |
| --- | --- |
| 对象 | 分配对象个数 |
| java.awt.Rectangle | 5,761,695 (40.8%) |
| java.awt.Insets | 1,920,583 (13.6%) |
| java.awt.Dimension | 1,920,166 (13.6%) |
| java.lang.String | 40,282 (0.3%) |
| java.lang.Long | 20,608 (0.1%) |
| java.lang.StringBuilder | 16,214 (0.1%) |
| java.util.HashMap | 8,585 (0.1%) |

* **结果分析**

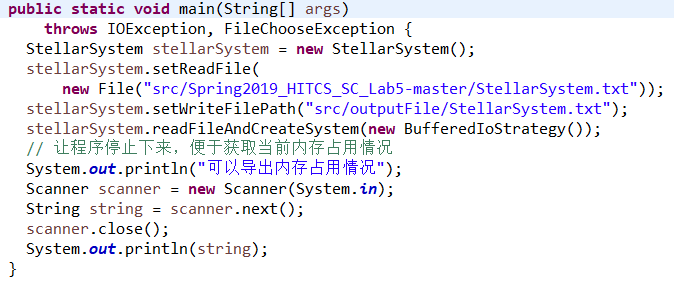
1. 发现java.awt库中的类对象最多。是因为我选择使用GUI将这个超大文件进行行星系统绘制，占据了大量的空间内存资源。
2. 发现String对象也很多，大约有4万个。因为行星名、行星状态、行星颜色、输出系统信息的文件路径、公转方向以及日志的输入message等都是构建的String对象。
3. StringBuilder对象也很多，大约16000个。因为输出系统信息的时候构建的是StringBuilder对象，输出科学记数法的时候也是构建的String Builder对象。
4. HashMap对象数目达8585个。为了提高读取文件、构建系统、查找信息等操作的速度，许多数据结构都是使用的hashMap，如截图



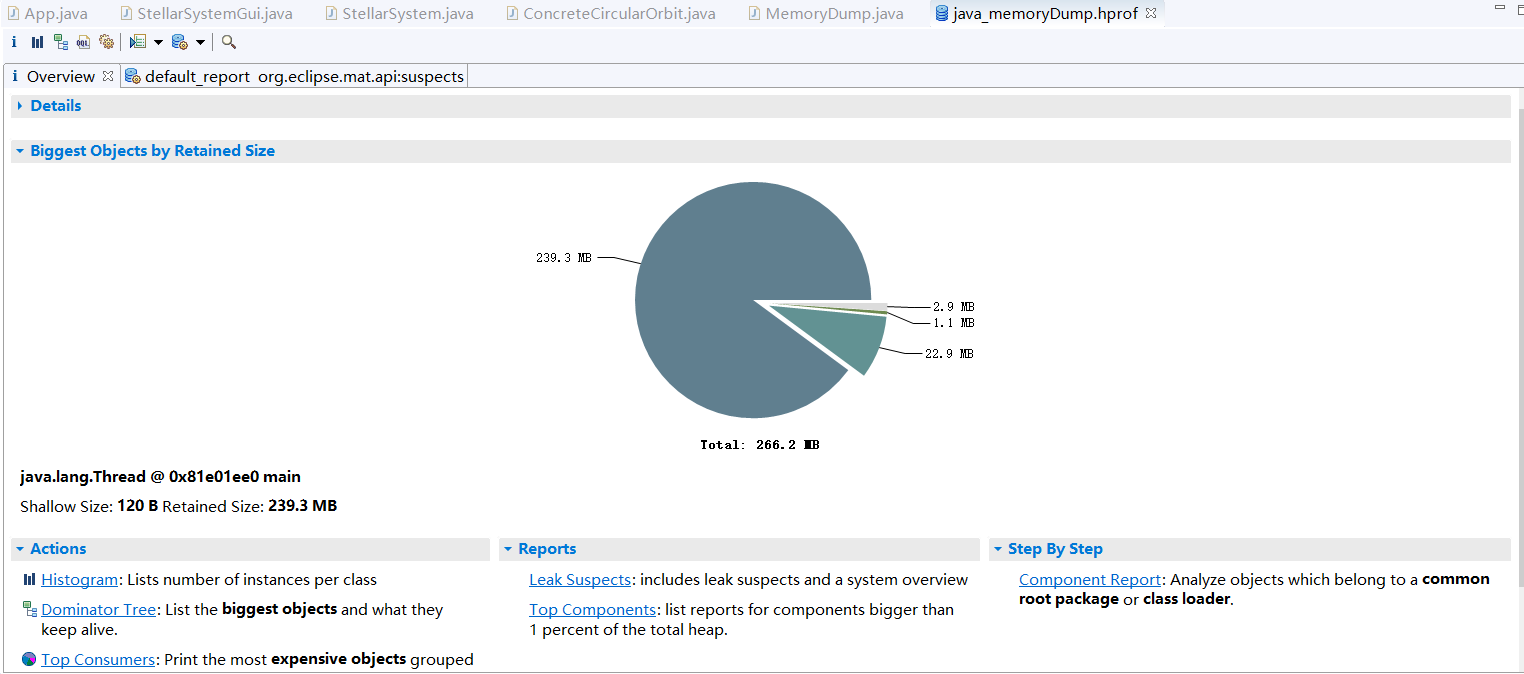
## Memory Dump Analysis and Performance Optimization

### 内存导出

1. 建立一个专门的类recordiotime.MemoryDump：便于导出构建完成大文件系统后的内存占用情况；
2. 程序中使用简单的Scanner代码：保证程序可以等待用户进行内存导出。

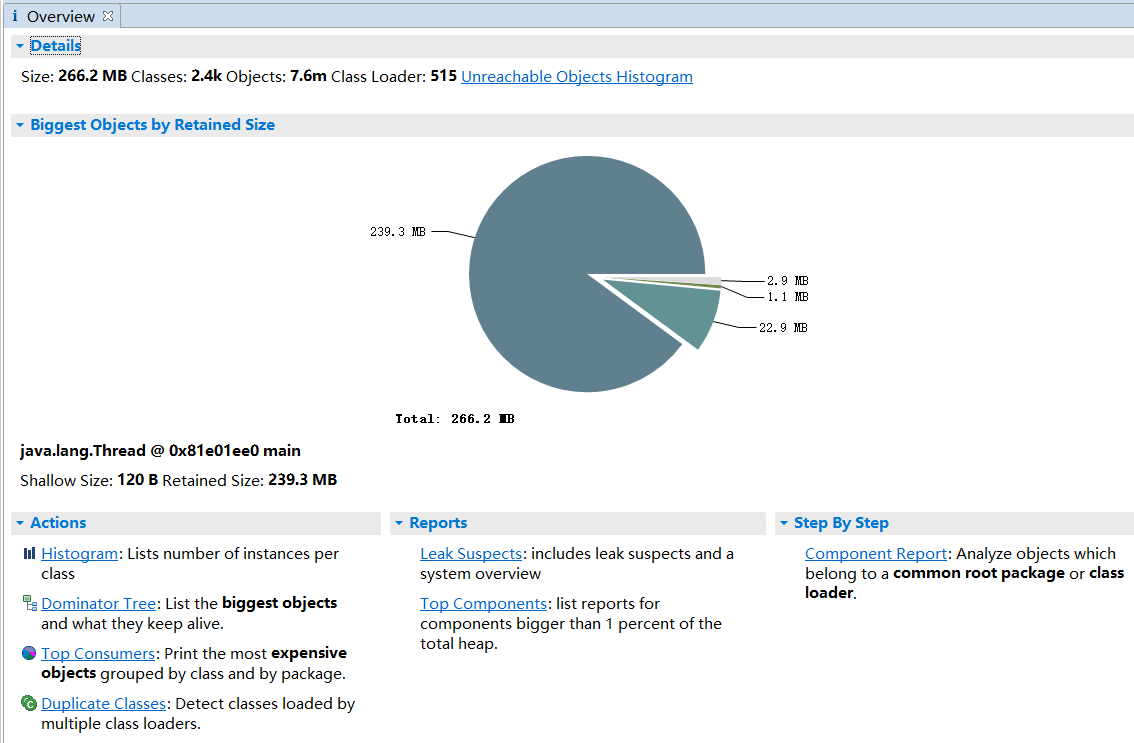


1. 使用eclipse的内存导出功能导出内存状态：项目右键->import->HeapDump->选择当前执行程序即可

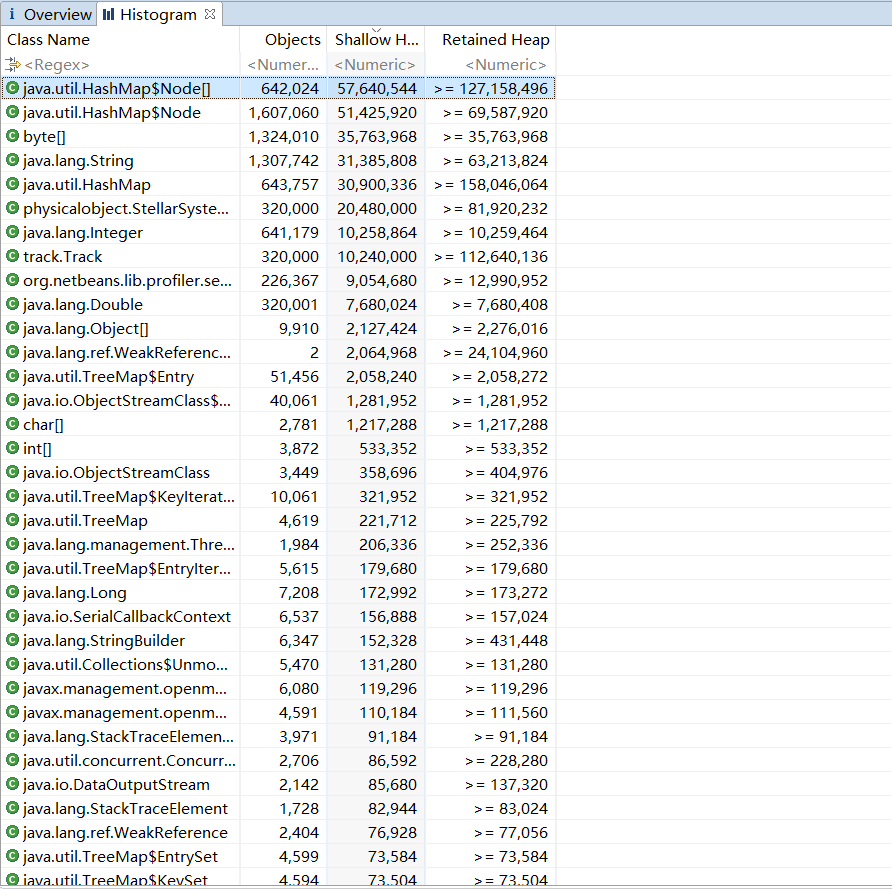


### 使用MAT分析内存导出文件

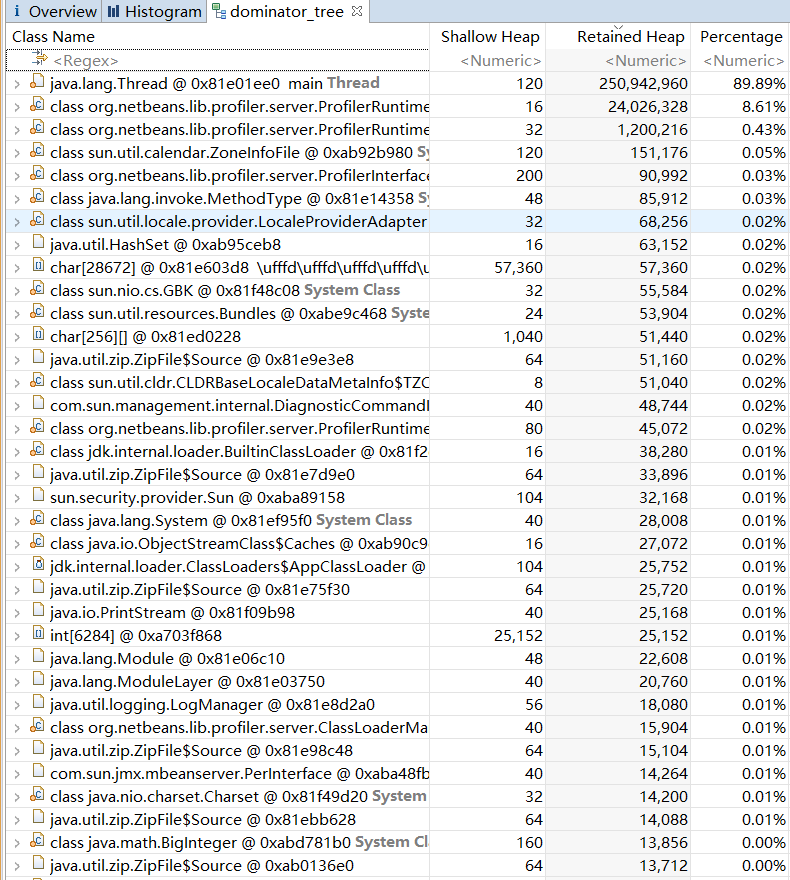
1. 使用eclipse的mat打开上一步生成的heap dump文件：file->openfile->选择内存导出文件即可；
2. overview视图



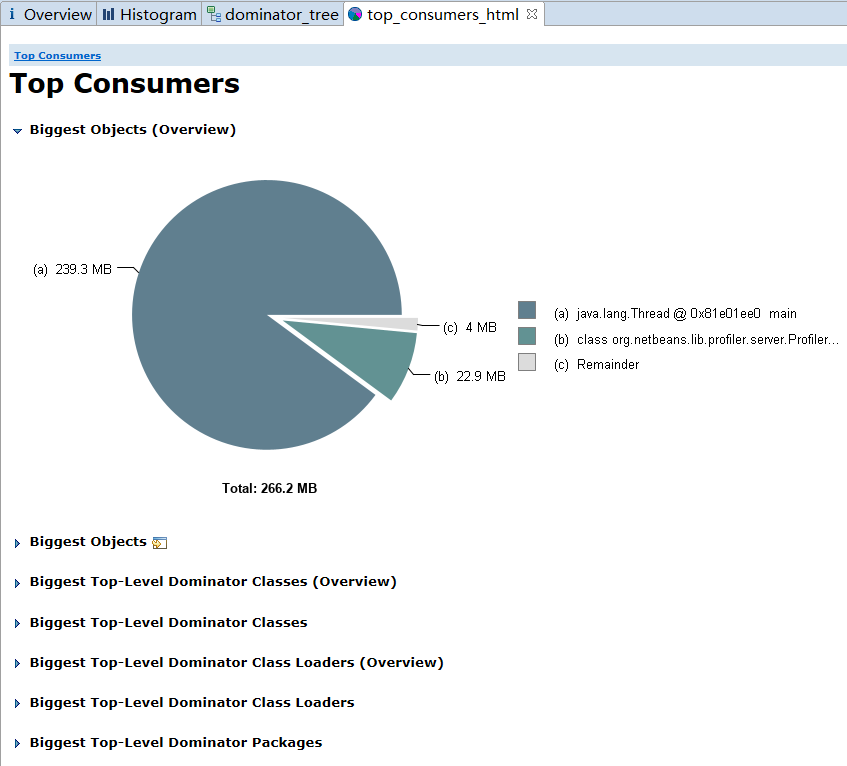
1. histogram视图



1. dominator tree视图



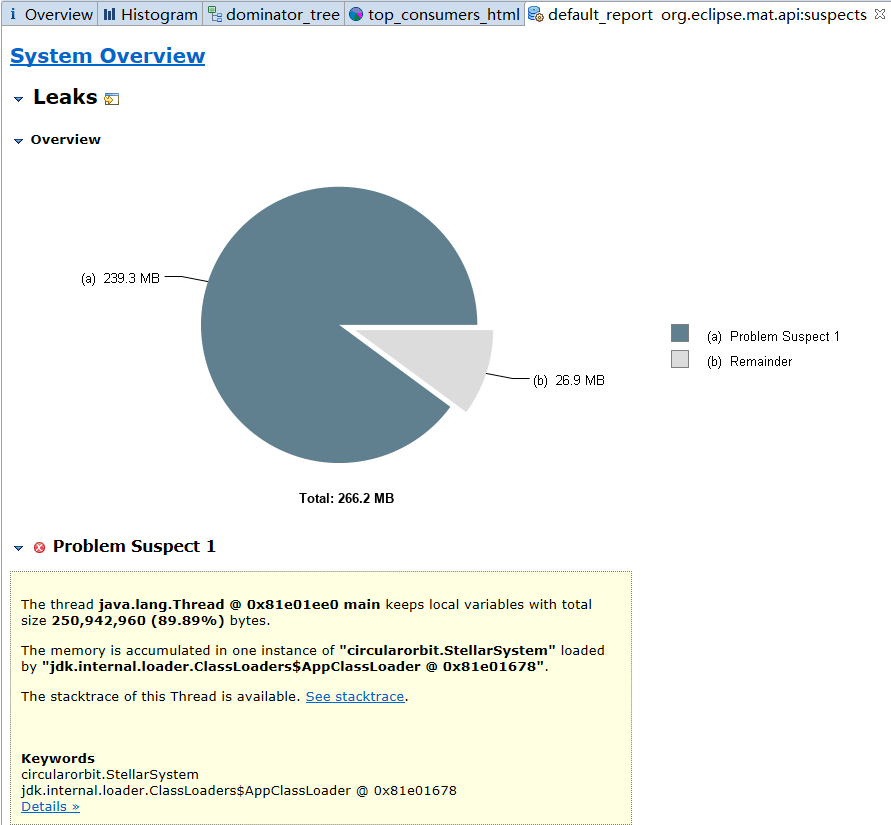
1. top consumers视图



1. leak suspects report视图

发现存在一个可能的内存泄漏情况

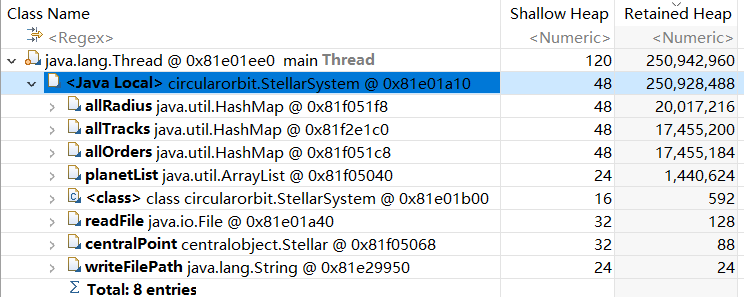
The thread **java.lang.Thread @ 0x81e01ee0 main** keeps local variables with total size **250,942,960 (89.89%)** bytes



### 发现热点/瓶颈并改进、改进前后的性能对比分析

1. 发现热点/瓶颈

发现三个数据结构hashMap以及一个ArrayList消耗内存retainedHeap较多。



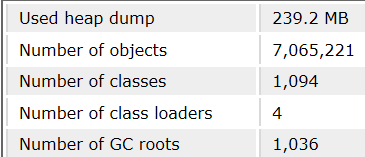
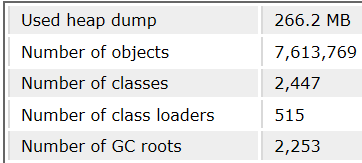
1. 改进内存占用热点

由于构建系统必须的是轨道物体、轨道、恒星以及他们相应的属性，因此这些内存是很难降低的，可以降低的是数据结构的节点数（比如hashMap的节点数、比如String的对象数目等）

我的操作有：删除了用处不大的数据结构planetList（原本用于储存所有的行星）+将部分类的属性中的String初始设为null，直到使用时才进行引用。

1. 改进前后对比

热点代码部分已经没有了planetList，整体占用情况大幅下降，下面为对比图，发现内存使用足足减少了**27MB**，相当于总内存占用的**10%**；对象数目减少了60万左右；类数目减少了也大幅减少。



左图为修改前，右图为修改后

### 在MAT内使用OQL查询内存导出

#### CircularOrbit 的所有对象实例

* OQL查找的输入

SELECT \* FROM INSTANCEOF circularorbit.CircularOrbit

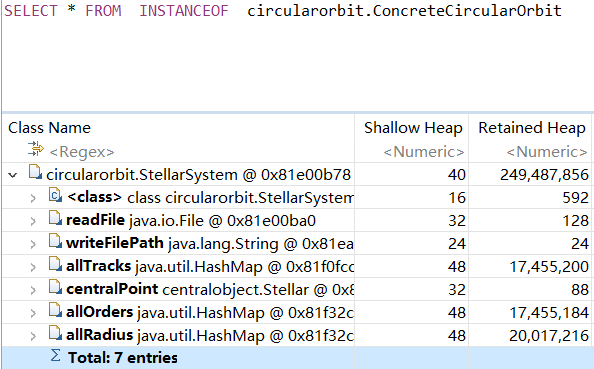
查询结果为空，因为并没有它的示例，如截图。



* OQL查找的输入

SELECT \* FROM INSTANCEOF circularorbit.ConcreteCircularOrbit

查询结果是找到了一个实例：为StellarSystem对象，如截图

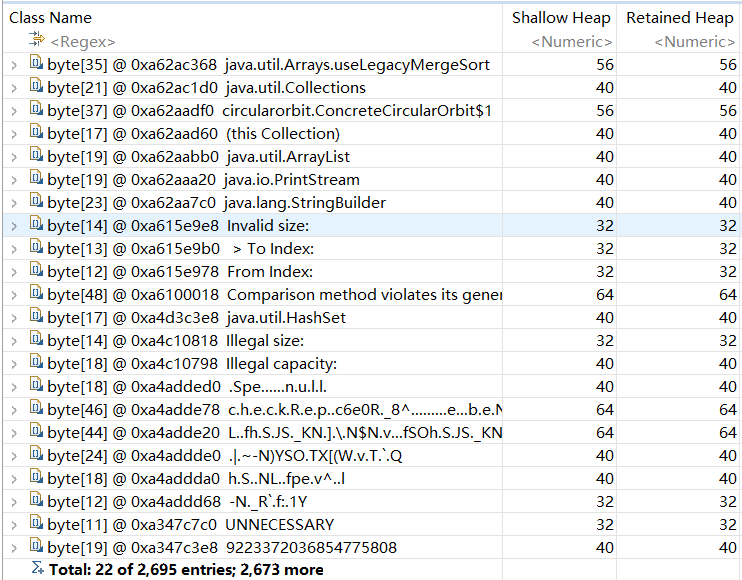


#### 大于特定长度 n 的 String 对象：我选择长度为10

* OQL查找输入为

SELECT OBJECTS s.value FROM java.lang.String s WHERE [s.value.@length>10](mailto:s.value.@length%3e10)

* 查询结果如截图



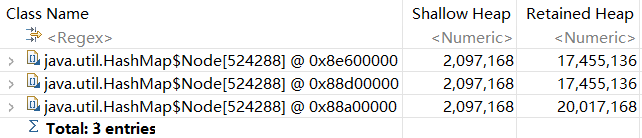
#### 大于特定大小的任意类型对象实例：我选择大小为100 000

* OQL查找输入为

SELECT \* FROM instanceof java.lang.Object o WHERE o.@usedHeapSize >= 100000

* 查询结果

搜索到了三个HashMap数据结构，如截图



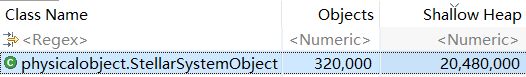
#### PhysicalObject（及其子类）对象实例的数量和总占用内存大小

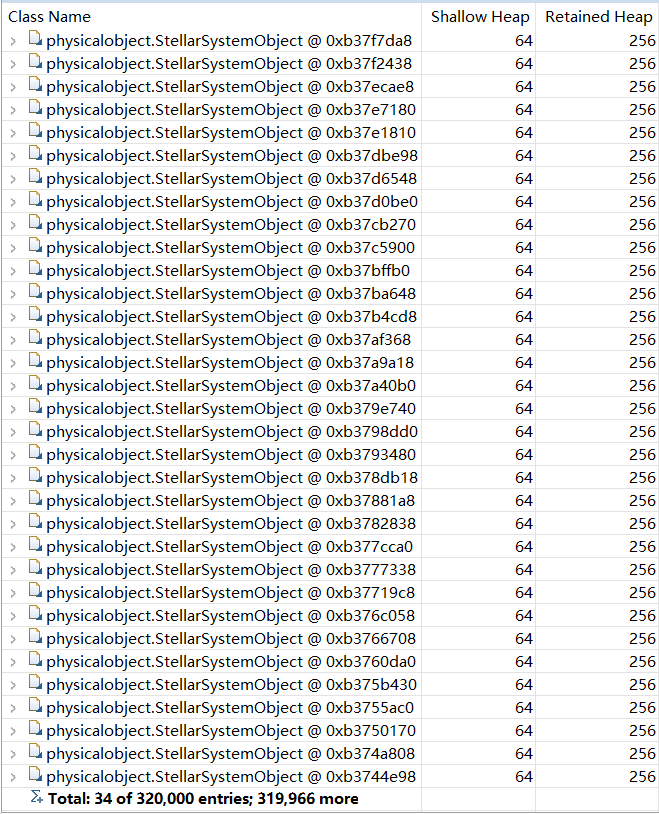
* OQL查找输入为

SELECT \* FROM instanceof physicalobject.PhysicalObject

* 查询结果

查询到的轨道物体对象实例有320 000个，分别对应超大文件中的320 000个行星。所有对象总大小为256\*320000 = 20 480 000，如截图



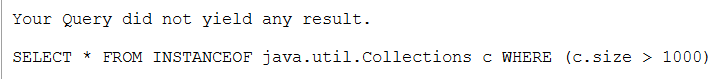


#### 所有包含元素数量大于 100 的 Collections 实例

* OQL查找输入为

SELECT \* FROM INSTANCEOF java.util.Collections c WHERE c.size > 1000

* 查询结果如截图

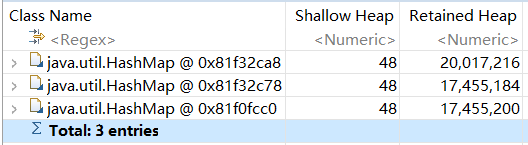


* OQL查找输入为

SELECT \* FROM INSTANCEOF java.util.HashMap map WHERE map.size > 1000

* 查询结果

查询到了3个HashMap的示例，如截图



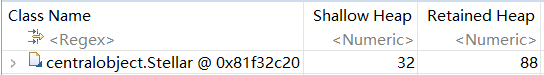
#### 我感兴趣的其他查询

* OQL查找centralobjetc（及其子类）对象实例的数量和总占用内存大小

查询输入为

SELECT \* FROM INSTANCEOF centralobject.CentralObject

查询结果：找到了一个示例，对应行星系统中的中心天体。如截图

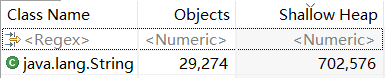


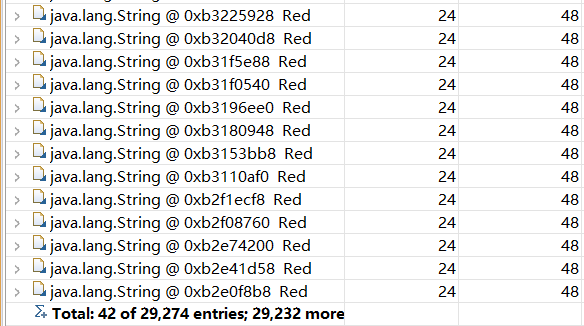
* OQL查找字串为“Red”的String对象的数目和总占用内存大小

查询输入为

SELECT \* FROM java.lang.String s WHERE toString(s).equals("Red")

查询结果：找到了29274个String对象，如截图





### 观察jstack导出程序运行时的调用栈

**首先应该获取进程的pid：**利用windows的资源管理器；

**其次导出调用栈状态：**在cmd中键入jstack pid即可获取。

**批注：**在读取文件并进行GUI绘图的时候已经进行了合法性判断+增删轨道/物体等操作

* 读取超大文件+图形化时导出



* 删除轨道+重画GUI时导出（主要是GUI是runnable）



* 增加轨道即增加物体+GUI重绘（主要是GUI是runnable）

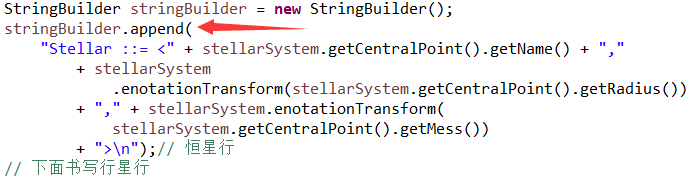


* 天体动态化（主要是GUI是runnable）

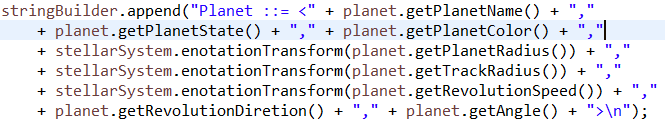


### 使用设计模式进行代码性能优化

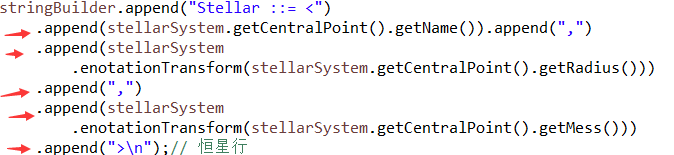
* StringBuilder优化：将需要大量进行字符串连接运算的地方改用StringBuilder类，如截图。



* StringBuilder进一步优化：StringBuilder类的append方法传入的参数中也不可以使用字符串的拼接操作，全部改为append操作如截图。



图一：可以将传入append方法中的带有+的字串全改为append操作



* 实现原型设计模式：让CircularOrbit类继承Cloneable接口，并实现deep copy并在三个具体的子类中进行了相应的实现。如截图



* 一些方法的私有属性不进行初始化：设为null，如截图

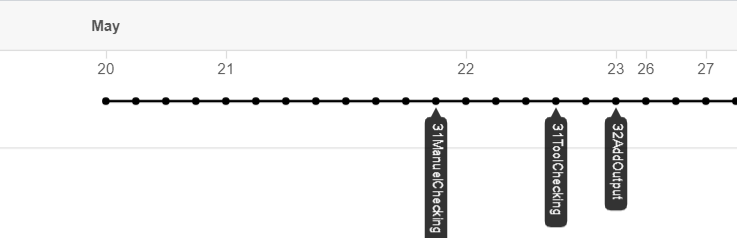


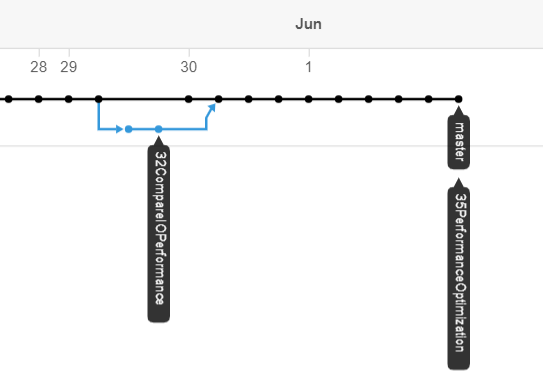
* 优化社交系统先广确定系统所有朋友所处轨道的方法：原本是对系统中所有的朋友均调用getDistance方法，时间复杂度为n^2；现在改为只调用一次，但在这一次里面充分利用每一次搜索的结果，只进行一次广搜，时间复杂度为o(n)。如截图



## Git仓库结构

实验各分支之间的关系通过github的insight界面显示，如下图。





# 实验进度记录

|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 时间段 | 计划任务 | 实际完成情况 |
| 2019-05-20 | 下午 | 完成仓库配置 | 完成 |
| 2019-05-20 | 晚上 | 完成插件配置 | 完成 |
| 2019-05-21 | 下午+晚上 | 完成checkStyle | 完成 |
| 2019-05-23 | 下午+晚上 | 完成3.1节 | 完成 |
| 2019-05-24 | 上午+晚上 | 完成3.2节 | 未完成，完成模式的框架 |
| 2019-05-26 | 全天 | 完成3.2节 | 完成各种策略的书写 |
| 2019-05-27 | 下午+晚上 | 完成3.3节 | 遇到问题，无法解决 |
| 2019-05-28 | 晚上 | 完成3.3节 | 仍无法解决问题 |
| 2019-05-29 | 晚上 | 完成3.4节 | 完成 |
| 2019-05-30 | 下午+晚上 | 完成3.5节 | 设计模式部分未完成 |
| 2019-05-31 | 下午 | 完成3.5节 | 完成 |
| 2019-06-01 | 全天 | 完成3.3节 | 完成 |

# 实验过程中遇到的困难与解决途径

|  |  |
| --- | --- |
| 遇到的难点 | 解决途径 |
| gc的终端显示中没有Full GC | jdk版本问题，选择安装JDK8 |
| 社交系统建立系统超级慢 | 定位到问题在广搜上，优化了调用方式和建立系统方式 |
| 维护了许多数据结构 | 通过.jprof的分析逐渐减少无用的数据结构 |

# 实验过程中收获的经验、教训、感想

## 实验过程中收获的经验和教训

* 对java的命令行操作不太熟悉
* 对设计模式的应用还是不太熟练

## 针对以下方面的感受

1. 代码“看起来很美”和“运行起来很美”，二者之间有何必然的联系或冲突？哪个比另一个更重要些吗？在有限的编程时间里，你更倾向于把精力放在哪个上？
2. 诸如SpotBugs和CheckStyle这样的代码静态分析工具，会提示你的代码里有无数不符合规范或有潜在bug的地方，结合你在本次实验中的体会，你认为它们是否会真的帮助你改善代码质量？
3. 为什么Java提供了这么多种I/O的实现方式？从Java自身的发展路线上看，这其实也体现了JDK自身代码的逐渐优化过程。你是否能够梳理清楚Java I/O的逐步优化和扩展的过程，并能够搞清楚每种I/O技术最适合的应用场景？
4. JVM的内存管理机制，与你在《计算机系统》课程里所学的内存管理基本原理相比，有何差异？有何新意？你认为它是否足够好？
5. JVM自动进行垃圾回收，从而避免了程序员手工进行垃圾回收的麻烦（例如在C++中）。你怎么看待这两种垃圾回收机制？你认为JVM目前所采用的这些垃圾回收机制还有改进的空间吗？
6. 基于你在实验中的体会，你认为“通过配置JVM内存分配和GC参数来提高程序运行性能”是否有足够的回报？
7. 通过Memory Dump进行程序性能的分析，JMC/JFR、VisualVM和MAT这几个工具提供了很强大的分析功能。你是否已经体验到了使用它们发现程序热点以进行程序性能优化的好处？
8. 使用各种代码调优技术进行性能优化，考验的是程序员的细心，依赖的是程序员日积月累的编程中养成的“对性能的敏感程度”。你是否有足够的耐心，从每一条语句、每一个类做起，“积跬步，以至千里”，一点一点累积出整体性能的较大提升？
9. 关于本实验的工作量、难度、deadline。
10. 到目前为止，你对《软件构造》课程的意见与建议。

* 运行起来很美更重要；
* SpotBugs会提高我的代码质量，但感觉CheckStyle帮助不大，因为有这么多的规范，只需要选中一种规范，就可以达到程序的良好的可读性；
* 我感觉java的自动垃圾回收机制解放了程序员的双手，但是感觉自动垃圾回收机制还是需要程序员不断地根据具体的代码进行相应的调整；
* Memory Dump和VisualVM确实好用呀
* 本实验对于一个java命令行基础较差的我来说，感觉工作量有点大；感觉难度适中。