

**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 | 强文杰 |
| 学号 | 1171000410 |
| 班号 | 1703005 |
| 电子邮件 | [672334335@qq.com](mailto:672334335@qq.com) |
| 手机号码 | 18800421389 |

**目录**

[1 实验目标概述 1](#_Toc3923017)

[2 实验环境配置 1](#_Toc3923018)

[3 实验过程 3](#_Toc3923019)

[3.1 Static Program Analysis 4](#_Toc3923020)

[3.1.1 人工代码走查（walkthrough） 4](#_Toc3923021)

[3.1.2 使用CheckStyle和SpotBugs进行静态代码分析 6](#_Toc3923022)

[3.2 Java I/O Optimization 11](#_Toc3923023)

[3.2.1 多种I/O实现方式 11](#_Toc3923024)

[3.2.2 多种I/O实现方式的效率对比分析 16](#_Toc3923025)

[3.3 Java Memory Management and Garbage Collection (GC) 19](#_Toc3923026)

[3.3.1 使用-verbose:gc参数 19](#_Toc3923027)

[3.3.2 用jstat命令行工具的-gc和-gcutil参数 21](#_Toc3923028)

[3.3.3 使用jmap -heap命令行工具 23](#_Toc3923029)

[3.3.4 使用jmap -clstats命令行工具 25](#_Toc3923030)

[3.3.5 使用jmap -permstat命令行工具 26](#_Toc3923031)

[3.3.6 使用JMC/JFR、jconsole或VisualVM工具 26](#_Toc3923032)

[3.3.7 分析垃圾回收过程 27](#_Toc3923033)

[3.3.8 配置JVM参数并发现优化的参数配置 28](#_Toc3923034)

[3.4 Dynamic Program Profiling 29](#_Toc3923035)

[3.4.1 使用JMC或VisualVM进行CPU Profiling 31](#_Toc3923036)

[3.4.2 使用VisualVM进行Memory profiling 32](#_Toc3923037)

[3.5 Memory Dump Analysis and Performance Optimization 34](#_Toc3923038)

[3.5.1 内存导出 34](#_Toc3923039)

[3.5.2 使用MAT分析内存导出文件 34](#_Toc3923040)

[3.5.3 发现热点/瓶颈并改进、改进前后的性能对比分析 39](#_Toc3923041)

[3.5.4 在MAT内使用OQL查询内存导出 40](#_Toc3923042)

[3.5.5 观察jstack/jcmd导出程序运行时的调用栈 42](#_Toc3923043)

[3.5.6 使用设计模式进行代码性能优化 44](#_Toc3923044)

[4 实验进度记录 46](#_Toc3923045)

[5 实验过程中遇到的困难与解决途径 47](#_Toc3923046)

[6 实验过程中收获的经验、教训、感想 48](#_Toc3923047)

[6.1 实验过程中收获的经验和教训 48](#_Toc3923048)

[6.2 针对以下方面的感受 48](#_Toc3923049)

# 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），对其进行持续的改进和优化。

需要使用的工具和方法主要有以下几种：

1. 静态代码分析（CheckStyle 和 SpotBugs）

2. 动态代码分析（Java 命令行工具 jstat、jmap、jcmd、VisualVM、JMC、

JConsole 等）

3. JVM 内存管理与垃圾回收（GC）的优化配置

4. 运行时内存导出(memory dump)及其分析（Java 命令行工具 jhat、MAT）

5. 运行时调用栈及其分析（Java 命令行工具 jstack）；

6. 高性能 I/O

7. 基于设计模式的代码调优

8. 代码重构

# 实验环境配置

简要陈述你配置本次实验所需环境的过程，必要时可以给出屏幕截图。

特别是要记录配置过程中遇到的问题和困难，以及如何解决的。

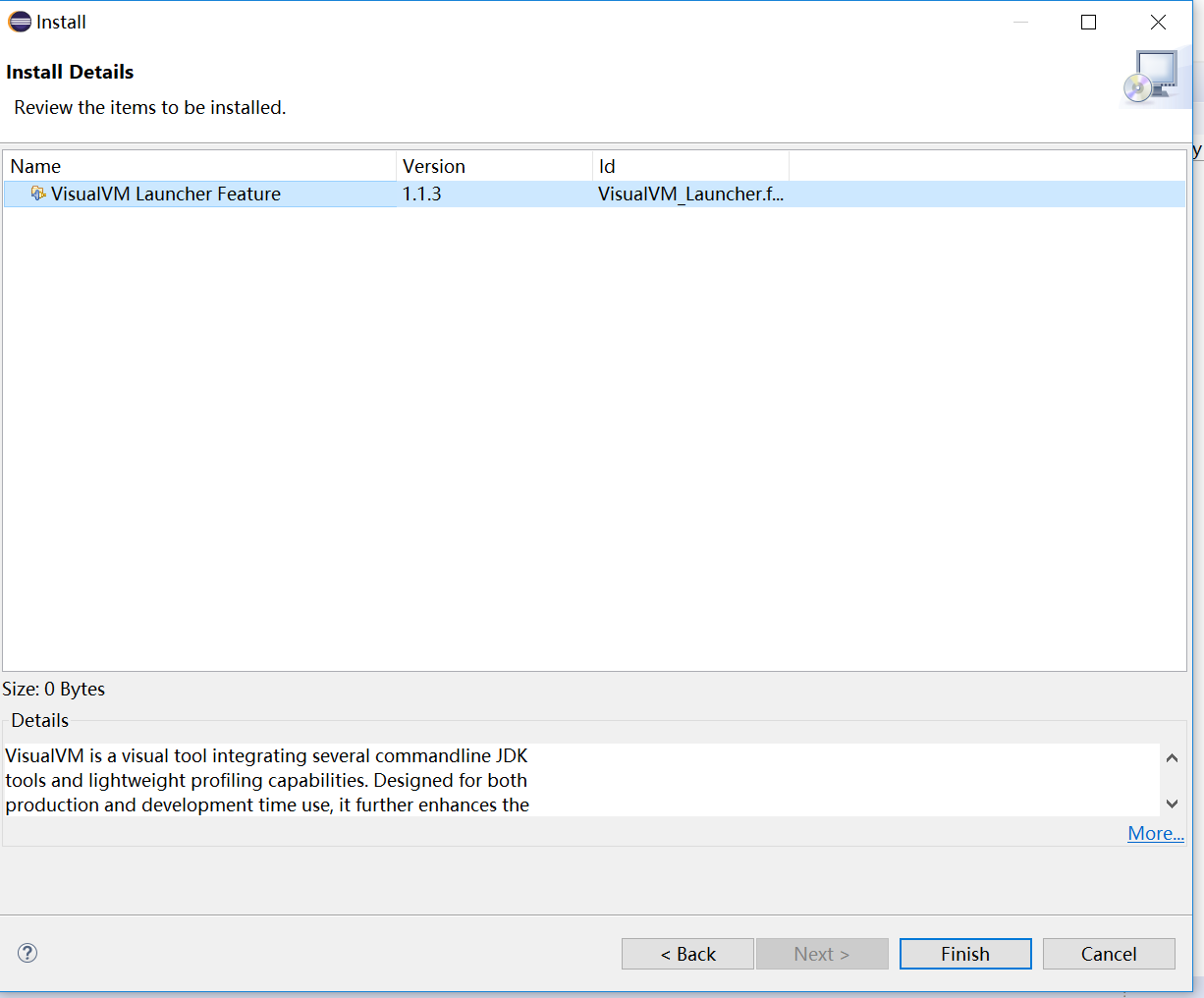
**一. 配置Visual VM**

首先在https://visualvm.github.io/download.html 下载VisualVM启动器和在https://visualvm.github.io/idesupport.html 下载visualVM到Eclipse的插件。

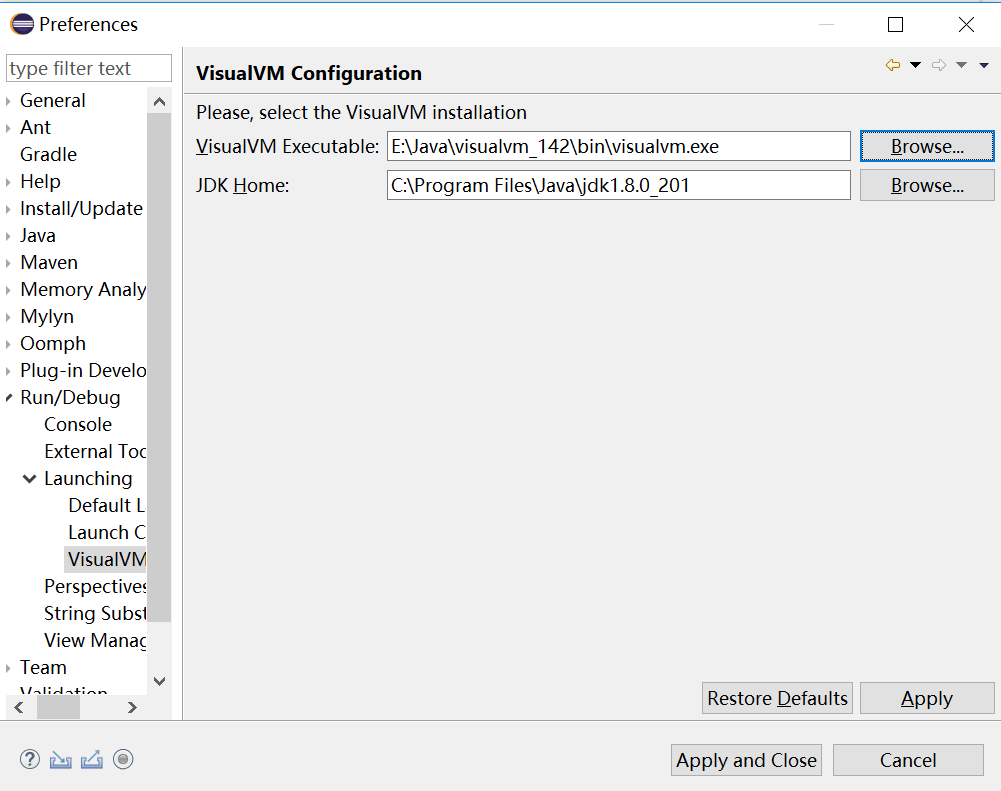
下载完成后将visualvm\_141.zip 解压到当前目录下。

将visualvm\_launcher\_u2\_eclipse.zip 解压到当前目录下。

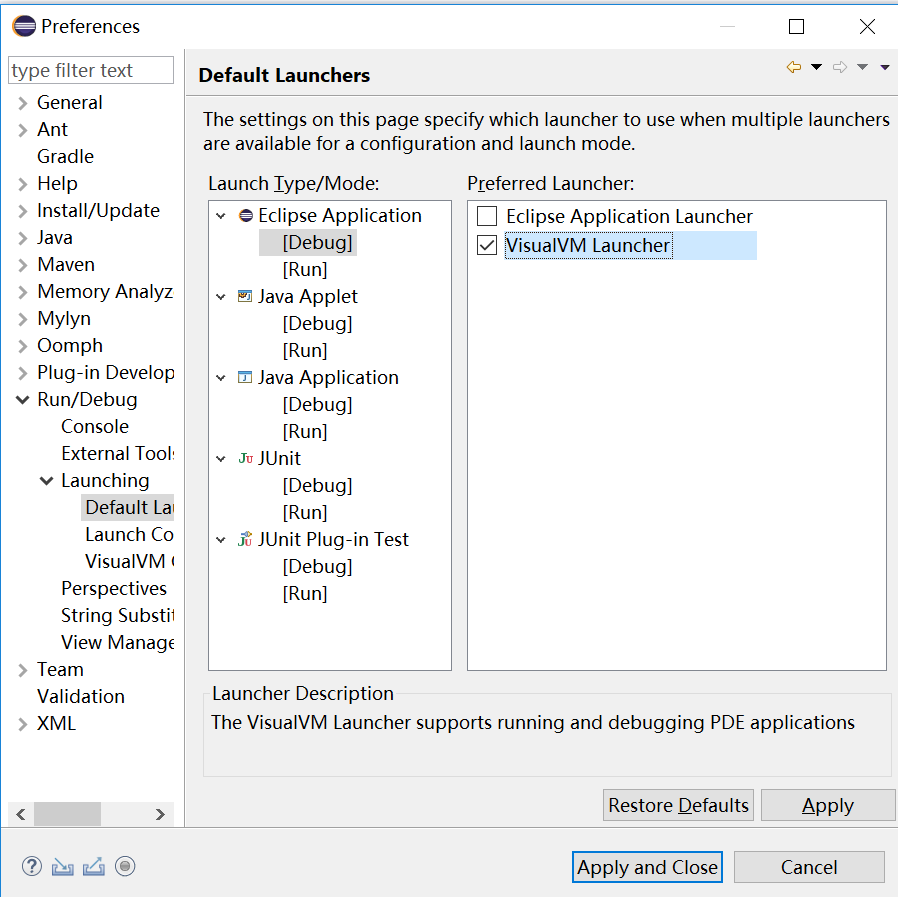
在Eclipse中选择helper下的install new software，在目录选择解压后的visualvm\_launcher\_u2\_eclipse，然后进行安装。



在Eclipse的Window下的Preferences中按如图配置VisualVM的安装目录以及jdk的目录。



配置完成visualVM之后，再进行启动注册的配置，在Default Launchers中选择启动器。



**二. 配置MAT**

在Eclipse中选择helper下的install new software，在网址中输入http://download.eclipse.org/mat/1.8.1/update-site/ 进行安装。

在这里给出你的GitHub Lab5仓库的URL地址（Lab5-学号）。

https://github.com/ComputerScienceHIT/Lab5-1171000410.git

# 实验过程

请仔细对照实验手册，针对每一项任务，在下面各节中记录你的实验过程、阐述你的设计思路和问题求解思路，可辅之以示意图或关键源代码加以说明（但千万不要把你的源代码全部粘贴过来！）。

## Static Program Analysis

### 人工代码走查（walkthrough）

列出你所发现的问题和所做的修改。每种类型的问题只需列出一个示例即可。

**以下选定Google编程规范。**

**#源文件基础**

没有需要修改的地方

**#源文件结构**

1.import语句：删去了import语句中的通配符

2.对类的成员顺序重新排序，修改了ConcreteCircularOrbit中一些混乱的成员顺序。

**#格式**

1.使用大括号(即使是可选的)：将很多if后面省去的大括号重新添加上。

2.对于非空块和块状结构，大括号遵循Kernighan和Ritchie风格。重新调整了大括号的使用情况。

3.块缩进：2个空格。把许多块缩进使用的tab符改成了两个空格。

4.列限制：80或100

对于AtomStructure等继承和实现关系比较复杂的类，单行字符超出限制，换行处理。

5. 把方法体中的组合声明修改，比如int a, b;

6. 修改了C风格的数组声明

在read File方法中，定义了字符串数组，中括号是类型的一部分：String[] args，而非String args[]

7.在MyOrbitScences类中，存在大量的switch语句，缩进与其它块状结构一致，switch块中的内容缩进为2个空格。

8.把类中原来自动换行时缩进的tab符改为至少+4个空格

9.对于空白。在函数体内，语句的逻辑分组间使用了空行。

**#命名约定**

1.根据对所有标识符都通用的规则，把方法中friend\_set这种类型的命名修改成friendSet

2.包名全部小写，连续的单词只是简单地连接起来，不使用下划线。

把APIS包rename为apis，把centralObject包rename为centralobject

3. 类名以UpperCamelCase风格编写；方法名都以lowerCamelCase风格编写。这类型的问题没有发现。

4.对方法中出现的常量啊a，修改其命名为：全部字母大写，用下划线分隔单词。

5. 参数名

把单个字符命名的参数名全部修改为一个单词。

**#编程实践**

1.把MyException类、MyFormatter和ConcreteCircularOrbit类中的合法重写全部加上了@Override

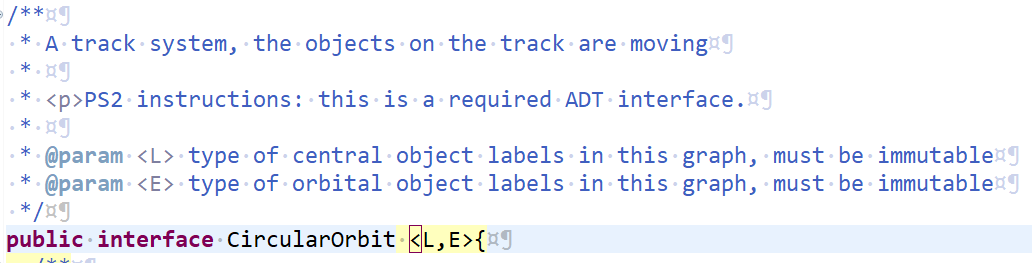
2.对捕获的异常全部做出相应

**#Javadoc**

1.添加<p>。添加的情况：空行(即，只包含最左侧星号的行)会出现在段落之间和Javadoc标记(@XXX)之前(如果有的话)。 除了第一个段落，每个段落第一个单词前都有标签<p>，并且它和第一个单词间没有空格。

2. 标准的Javadoc标记按以下顺序出现：@param, @return, @throws, @deprecated, 前面这4种标记如果出现，描述都不能为空。

如图所示：



### 使用CheckStyle和SpotBugs进行静态代码分析

列出你所发现的问题和所做的修改。每种类型的问题只需列出一个示例即可。

在修改之前很恐怖：



**#源文件基础**

没有需要修改的地方

**#源文件结构**

1.修改import的顺序



2.import中避免使用通配符



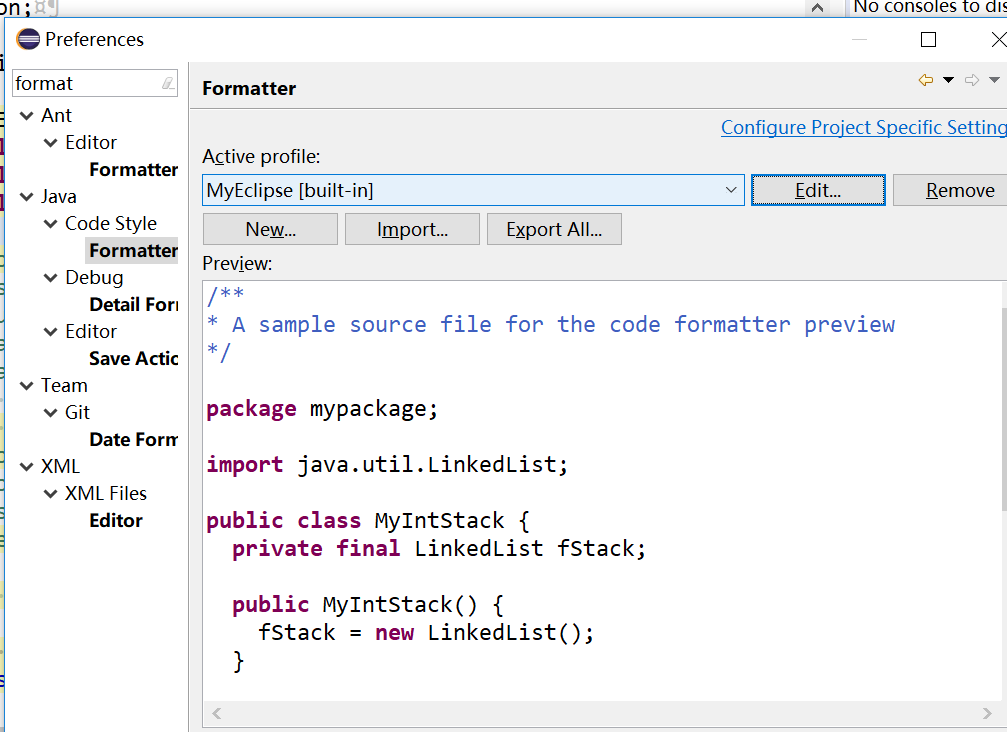
**#格式**

1.每一个变量的定义必须在它的声明处，且在同一行

要求同一行只声明一个变量。

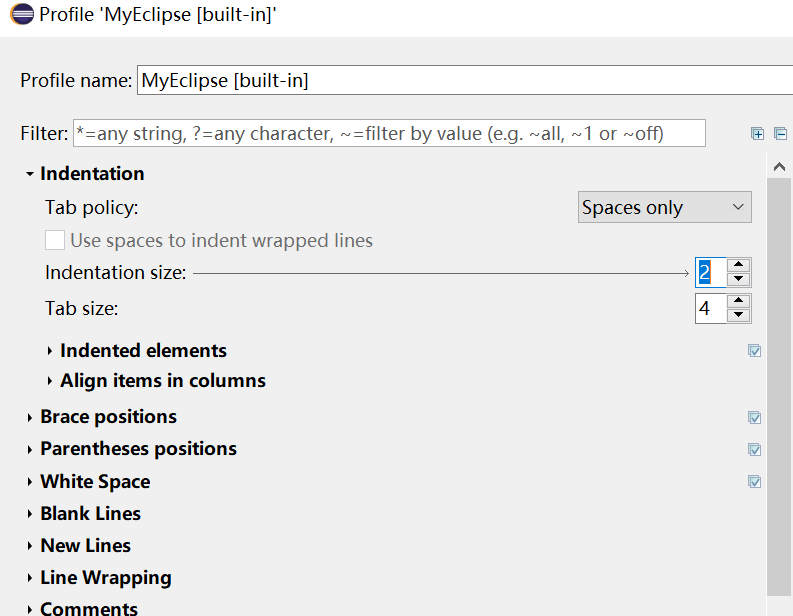
2.行内含有制表符tab，缩进符数量不正确。

这个问题在所有的代码行中都出现了，当然对于这么大量的代码，我们不可能一行行修改，修改方式如下：

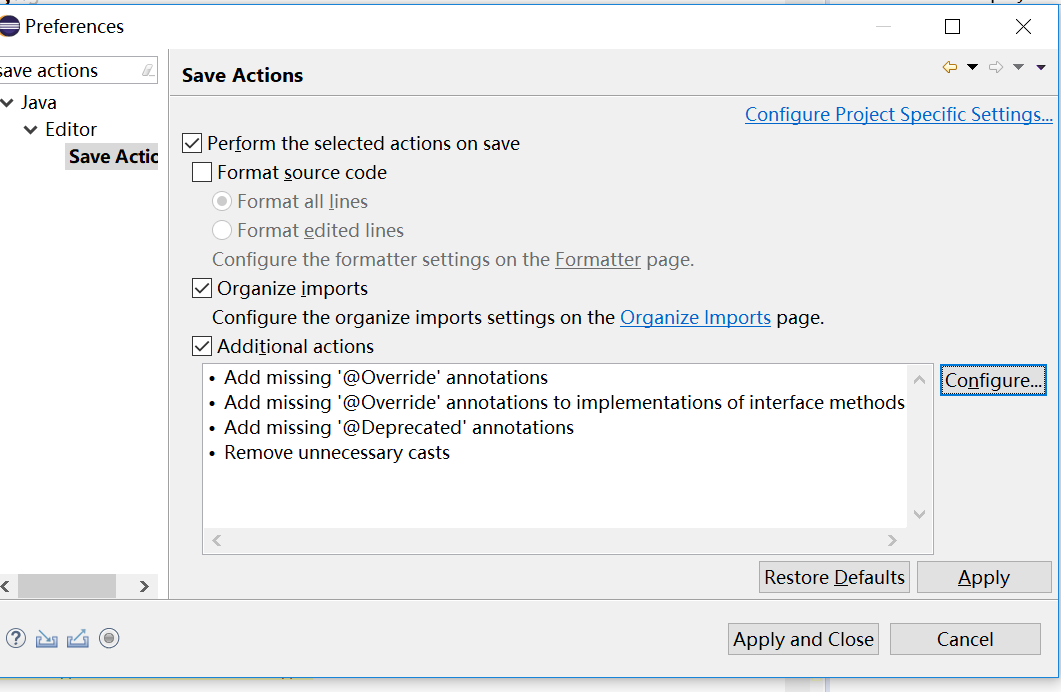


采用的方法为：点击“Java”->“Code Style”-> “Formatter”菜单，在右侧选中模板后点击“Edit”按钮

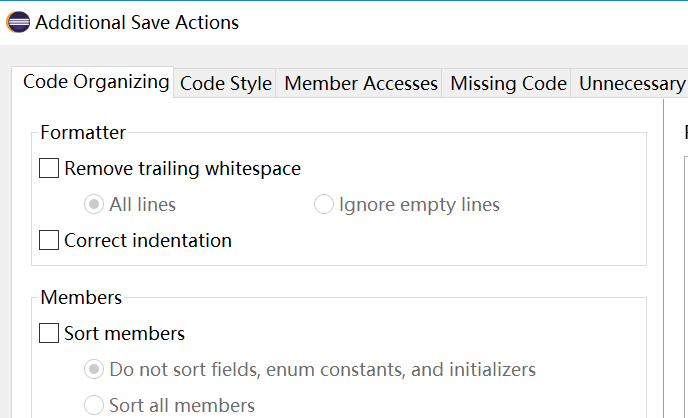
在Tab policy处选择Space only ，并且Indentation Size设置为2，这是为了使4个缩进符变为2个缩进符。



最后勾选上”Perform the selected actions on save”,”Additional actions”,并点击”Configure”进行配置.



使Correcct indentation为未勾选的状态即可。



最后“CTRL + SHIFT + F”快捷键，此问题解决。（怎么感觉在写教程2333）

3. 变量'y'声明及第一次使用距离5行（最多：3 行）。若需要存储该变量的值，请将其声明为final的（方法调用前声明以避免副作用影响原值）。

解决方法：变量声明前增加final

4.缩进符数目不正确

有些无法批量修改的只能一点点改了。

5. WhitespaceAround: '+' is not preceded with whitespace.

“+”左右要留空白

**#命名约定**

1. Local variable name 'E2E' must match pattern '^[a-z]([a-z0-9][a-zA-Z0-9]\*)?$'.

把原本使用的T2E这种局部变量名称修改为t2e

**#编程实践**

1.把MyException类、MyFormatter和ConcreteCircularOrbit类中的合法重写全部加上了@Override

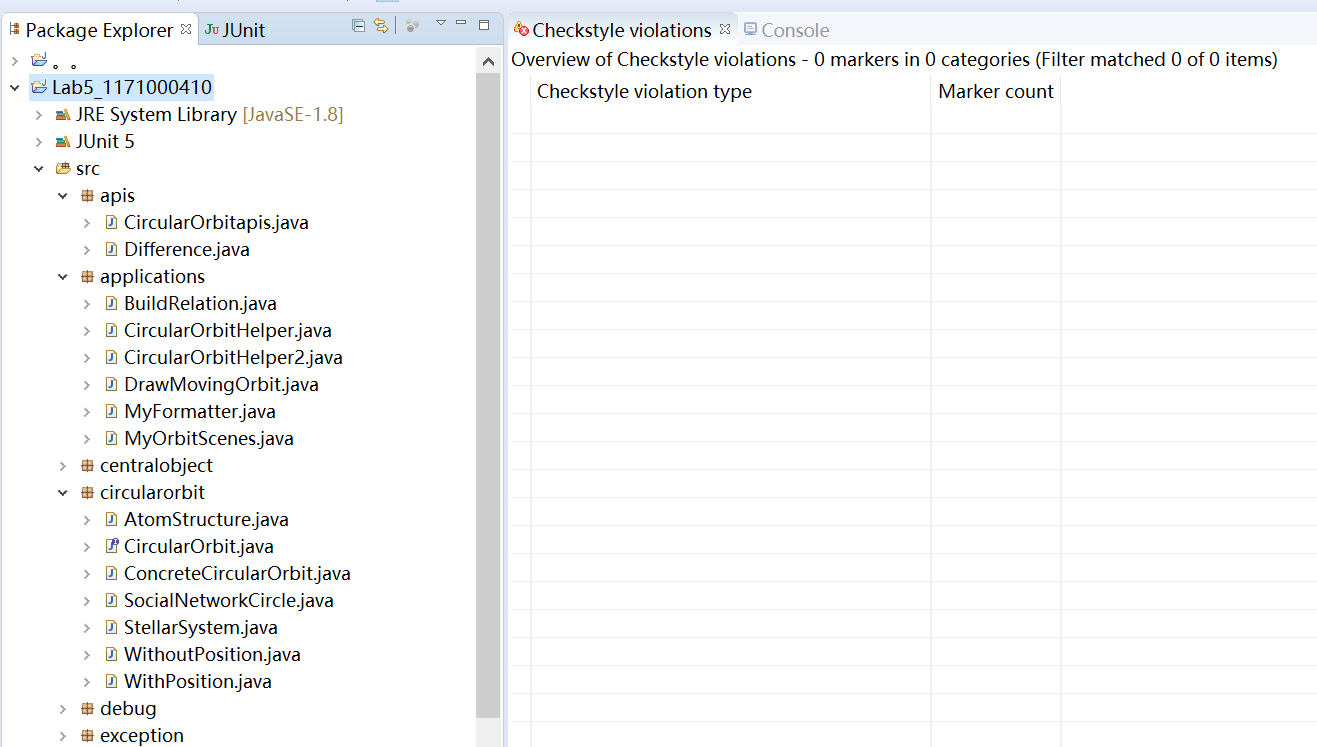
2.对捕获的异常全部做出相应

**#Javadoc**

1.给Javadoc的语句添加结束时期



**结果：手动改完所有的问题后：（警告终于全部消除了）**



对比分析两种工具发现问题的能力和发现问题的类型上有何差异。

**发现问题的能力：**两种工具发现问题的能力来说，Checkstyle运行起来更快，spotbugs运行稍慢一些，二者都能很细致的发现问题并且提示。个人认为spotbugs发现的问题对程序的影响更大，Checkstyle发现的主要是程序员写代码格式等不规范问题，其作用相对小一些。

**发现问题的类型：**Checkstyle发现的大多数是java代码中的格式错误，它能够自动化代码规范检查过程；而spotbugs发现的大多数是代码中可能存在的安全问题，例如对null的防范，或是考虑代码的正确性。

## Java I/O Optimization

### 多种I/O实现方式

**实现了哪些I/O方式来读写文件，具体如何实现的。**

**Input：**

首先为了strategy设计模式的方便实现和对input时间的准确收集，我选择了一次性读完整个文件，然后将文件存在一个string中（list中同理），然后通过分割字符串实现类似按行读文件的操作。

其中将文件存到字符串的操作如下：

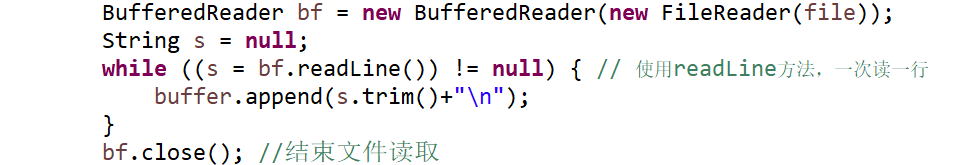
StringBuffer buffer = **new** StringBuffer();

buffer.append(s.trim()+"\n");

String fileStr = buffer.toString();

1.BufferedReader

字符串缓冲读取类。java的IO包使用包装器模式设计的，也就是说用FileReader包装了File，又用BufferedReader包装了FileReader，才可以用BufferedReader。

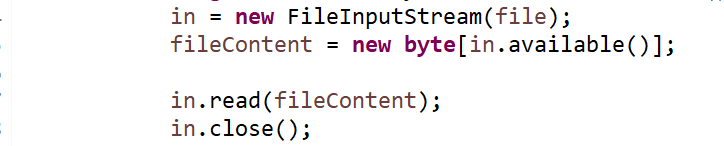


2.InputStream

字节输入流。

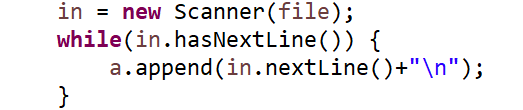
具体步骤如下：

使用File类找到一个文件；通过子类实例化父类对象；进行读操作，所有的内容都读到字符数组之中；读取内容；关闭输出流。



3.Scanner

Scanner经常被我们用于读取控制台的输入，但是我们只需要将他构造函数中的System.in替换成一个File文件，他就可以变成一个读取文件的Scanner了，Scanner的按行读取方法是nextLine()我们在使用它的时候可以在前面判断它时候hasNextLine()。



**Output:**

1.BufferedWriter

首先创建读取字符数据流对象关联所要复制的文件，创建缓冲区对象关联流对象，最后从缓冲区中将字符创建并写入到要目的文件中。

简单说BufferedWriter为我们提供了写字符串的方法，直接write()里面放个字符串就可以了。

最后需要 writer.flush();

writer.close();

2.OutputStream

创建字节输出流对象了做了几件事情：

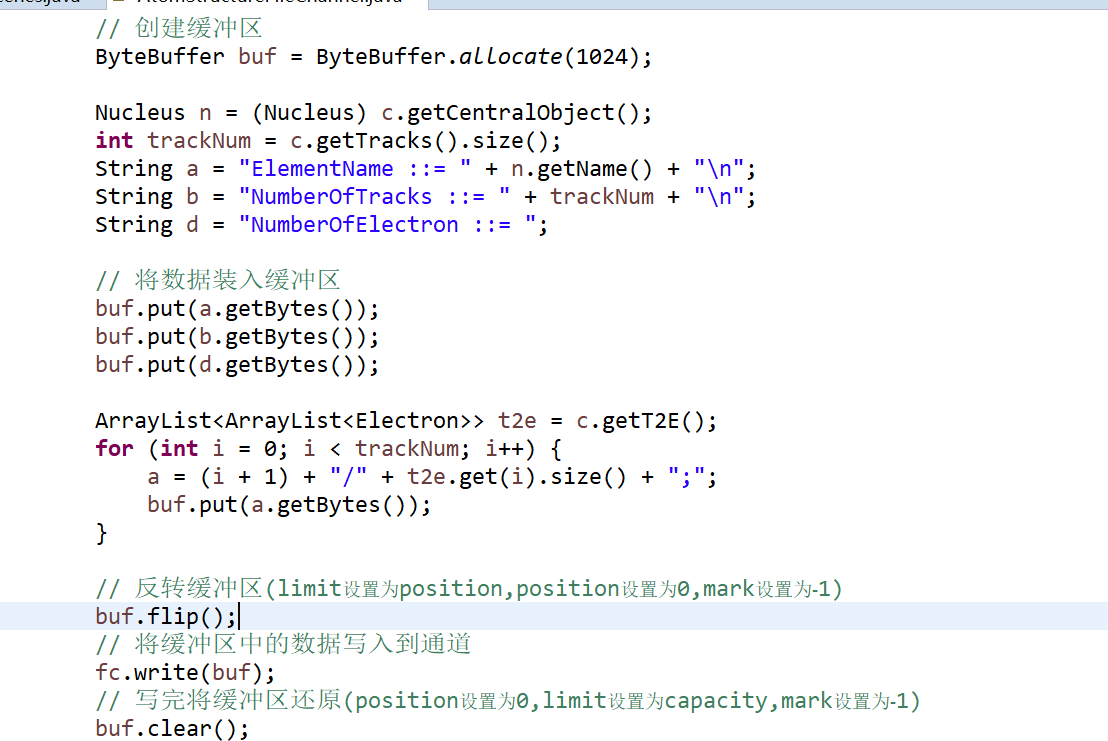
调用系统功能去创建文件；创建outputStream对象；把outputStream对象指向这个文件

3.FileChannel

开启FileChannel：无法直接打开FileChannel通过OutputStream获取FileChannel。

使用FileChannel.write()方法向FileChannel写数据，该方法的参数是一个Buffer。按字节写入。用完FileChannel后需将其关闭。

以下是AtomStructureFileChannel示例：



另外，在写实验中提供的大文件时，经常会出现缓冲区溢出的情况，因此我们需要对缓冲区剩余容量进行估计。写文件之前我给设置ByteBuffer缓冲区大小为0x300000（也就是3M），然后在后面向文件中写的时候。增加如下判断：

if(buf.remaining() < 1024) {

buf.flip();

fc.write(buf);

buf.clear();

}

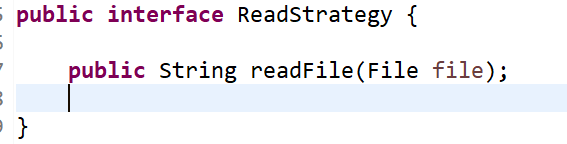
**如何用strategy设计模式实现在多种I/O策略之间的切换。**

因为我程序中Input和Output策略是分开使用的，故这部分也区分Input和Output来说明。

**Input:**

因为对于输入，我的每个策略都返回文件的字符串即可，因此可以借此来提高程序的可复用性，具体的策略只需要3个类，分别代表了三种读文件的方式。

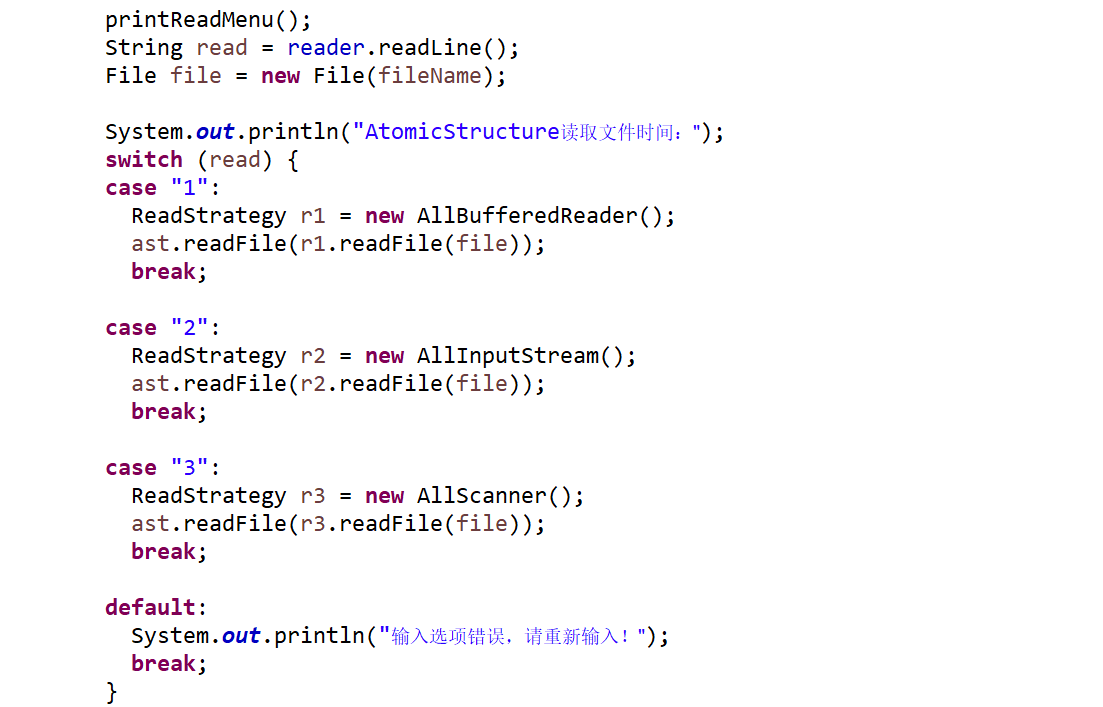
**抽象策略对象(Strategy)：**它可由接口或抽象类来实现。



**具体策略对象(ConcreteStrategy)：**它封装了读文件的不同实现策略，所有策略都需要implements ReadStrategy接口，每一个都根据自己这个方法的特点对方法进行了重写，其中具体的读文件方式已说明。

**环境对象(Context)：**分别在三个应用场景中实现了对抽象策略中定义的接口的引用。我们在使用它的时候，直接声明一个应用场景类，通过用户的不同输入来给Context传入不同的读策略即可，这样我们就能够只使用一个变量而获得多种行为模式了。

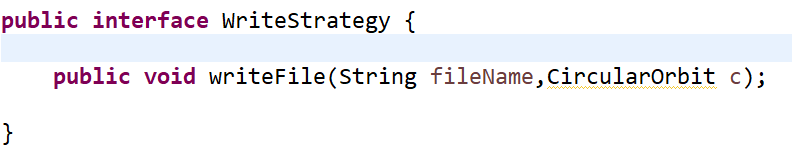
以下为三种读取策略的切换场景。



**Output：**

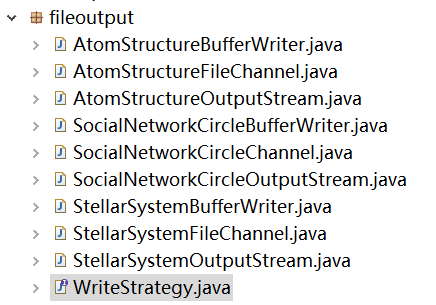
因为三个不同应用场景要实现三种不同的写文件的方式，并且此时没有很好提高复用性的方法，因此具体策略我用9个类来实现。

**抽象策略对象(Strategy)：**它可由接口或抽象类来实现。



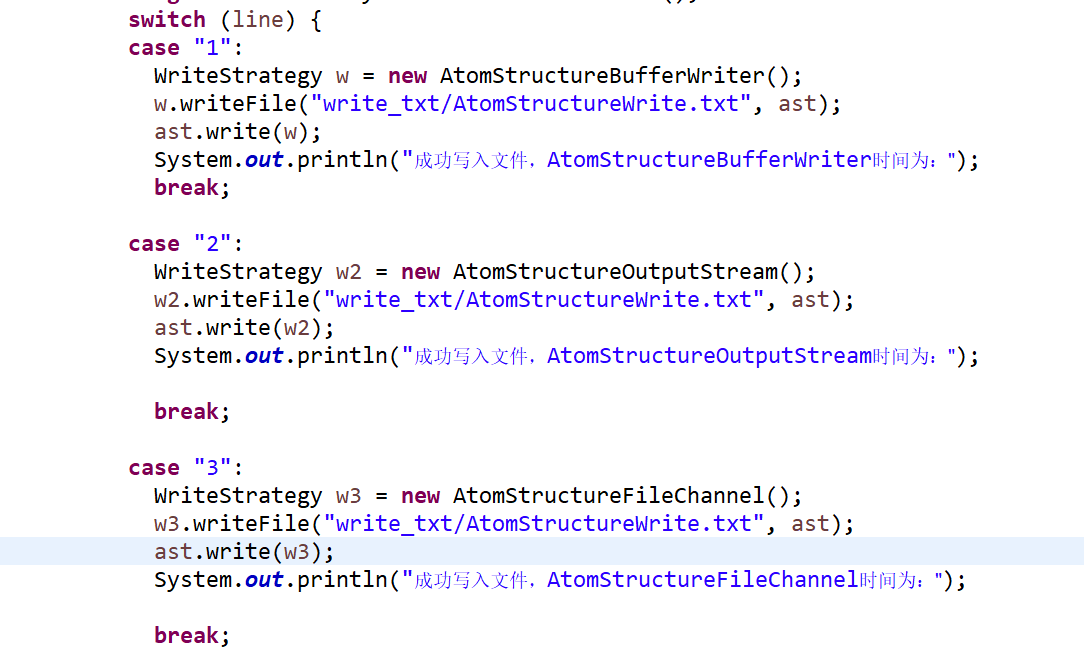
**具体策略对象(ConcreteStrategy)：**它封装了写文件的不同实现策略，所有策略都需要implements WriteStrategy接口，每一个都根据自己这个方法的特点对方法进行了重写，其中具体的读文件方式已说明。

写文件package的结构如下：



**环境对象(Context)：**分别在三个应用场景中实现了对抽象策略中定义的接口的引用。同理，我们在使用它的时候，直接声明一个应用场景类，通过用户的不同输入来给Context传入不同的写策略即可，这样我们就能够只使用一个变量而获得多种行为模式了。

方法：public void write(WriteStrategy writeStrategy)



### 多种I/O实现方式的效率对比分析

如何收集你的程序I/O语法文件的时间。

**1. 收集程序I/O语法文件的时间。**

因为Input阶段就采用了一次性将文件读完的策略，因此Input和Output收集时间的方式类似，在Input或Output开始阶段获取时间，在结束阶段再次获取时间，然后将两次时间相减即可。

具体操作：

**long** startTime = System.*currentTimeMillis*();

**long** finishTime = System.*currentTimeMillis*();

**long** totalTime = finishTime - startTime;

System.***out***.println(totalTime + "ms");

**2.程序优化：**

因为每次写入文件之前都需要在已经建立好轨道的基础上，因此即使我们一次性将文件读完，如果没有很好的建立轨道的策略，在读入大文件并进行I/O对比时，效率也是相当低的。

首先对lab4原本的设置进行修改，因为大文件中是存在非法情况的，因此我们不能像lab4中读到异常情况就重新选择文件，处理策略是读到异常情况，跳过该行的读取，继续读下一行。

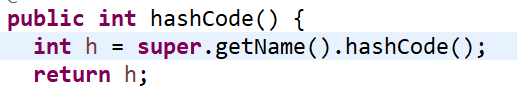
因为程序在一定时间内难以读出SocialNetworkCircle.txt文件并建立轨道，因此主要的优化在SocialNetworkCircle。

**问题一：**因为我们在读到每个Friend时，需要把建立的对象存储起来，存储方式是Hashset，然而hashSet存在一个致命的缺陷，当数据量较小时还没有问题，可当数据量过大时hashSet的存储会出现问题。hashSet集合存在大小，即存储的数据量上限，如果最初没有设置上限大小的话，默认的大小自然不可能有多大。当存储的数据量超出上限时，hashset会自动倍数扩大，这时集合set1就被更大的set2替代，而原本set1的数据自然不会直接出现在set2中，于是在扩大时set1会把存储的所有数据取出放入内存中，再重新放入set2中。这一部分不仅耗费了时间同时占用内存，降低了代码性能。

**优化一：**根据Friend对象最大的数目，初始设置了hashSet的大小是110000，这样向集合中添加的时候，不必一直transfer数据了。

**问题二：**即使我们有了存储Friend的hashSet，在读SocialTie取Friend对象的时候，需要遍历一个很大的集合，根据姓名来取对象，效率依然很低，第一次尝试的时候构建轨道花了2个小时。

**优化二：**既然遍历集合效率很低，我们为什么不采用hashMap取数据的方式呢，时间复杂度只有O(1)。因此我们重写Friend中hashCode方法，



每次在hashMap中寻找对象的时候，只需要根据name的hashCode值来搜索，相当于查找一个数组中指定位置的元素，时间复杂度只有O(1)



如此优化后，程序的运行效率提高了很多，方便下面的统计时间了。

**2.表格方式对比不同I/O的性能。**

由于对读入文件中的异常情况进行了剔除，因此写入的文件并没有读入的文件大。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | AtomStucture\_Medium.txt | StellarSystem.txt | SocialNetworkCircle.txt |
| 读文件 | BufferReader | 0ms | 238ms | 320ms |
| 写文件 | BufferWriter | 3ms | 327ms | 170ms |
| 读文件 | InputStream | 0ms | 24ms | 35ms |
| 写文件 | OutputStream | 2ms | 1556ms | 1123ms |
| 读文件 | Scanner | 7ms | 639ms | 1180ms |
| 写文件 | FileChannel | 2ms | 319ms | 165ms |

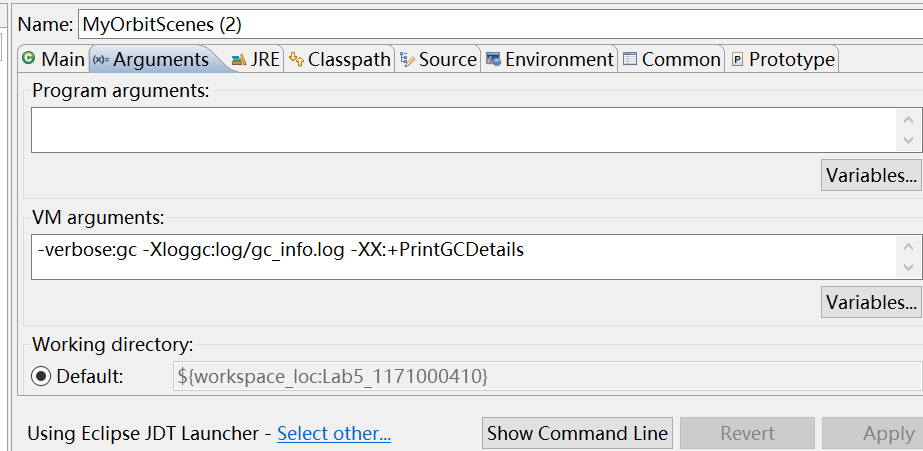
图形对比不同I/O的性能。

由于AtomicStructure\_Medium.txt文件较小，因此读此文件的时间较少，在图像上基本反映不出。

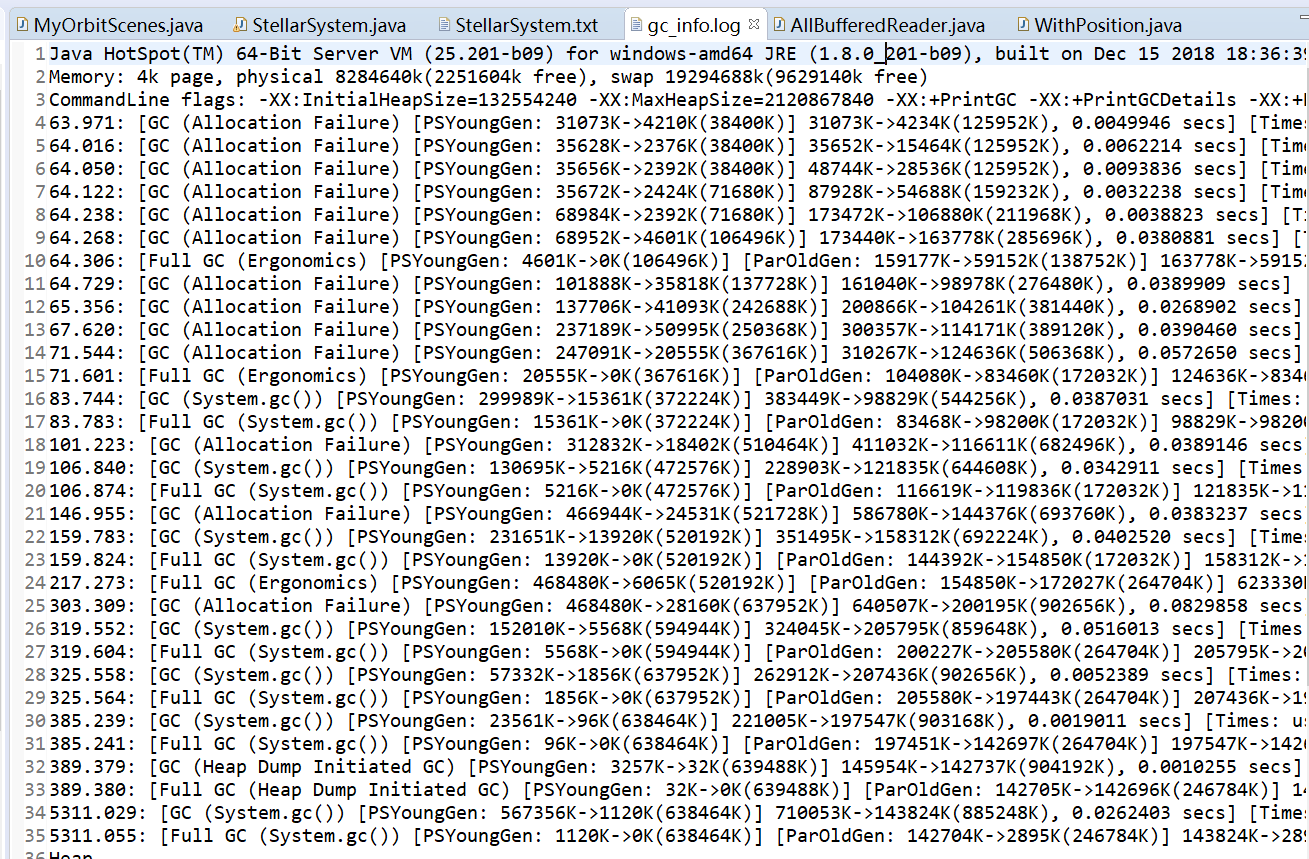
## Java Memory Management and Garbage Collection (GC)

### 使用-verbose:gc参数

配置参数：



日志文件中的部分内容如下：



**分析：**

让我们来挑几条典型的日志进行分析：

第一条：63.971: [GC (Allocation Failure) [PSYoungGen: 31073K->4210K(38400K)] 31073K->4234K(125952K), 0.0049946 secs] [Times: user=0.05 sys=0.02, real=0.01 secs]

1. 63.971：gc发生时，虚拟机运行了多少秒。

2. GC (Allocation Failure) ： 发生了一次垃圾回收，这是一次Minor GC 。注意它不表示只GC新生代，括号里的内容是gc发生的原因，这里的Allocation Failure的原因是年轻代中没有足够区域能够存放需要分配的数据而失败。

3. PSYoungGen: 使用的垃圾收集器的名字。

4. 31073K->4210K(38400K)指的是垃圾收集前->垃圾收集后(年轻代堆总大小)

5. 31073K->4234K(125952K),指的是垃圾收集前后，Java堆的大小（总堆125952K，堆大小包括新生代和年老代）

因此可以计算出年老代占用空间为125952k-38400k = 87552k

6. 0.0049946 secs：整个GC过程持续时间

7. [Times: user=0.05 sys=0.02, real=0.01 secs]分别表示用户态耗时，内核态耗时和总耗时。也是对gc耗时的一个记录。

第二条：106.840: [GC (System.gc()) [PSYoungGen: 130695K->5216K(472576K)] 228903K->121835K(644608K), 0.0342911 secs] [Times: user=0.13 sys=0.00, real=0.03 secs]

先对于第一条的(Allocation Failure) 变成了(System.gc())，说明这是一次成功的垃圾回收。

第三条：83.783: [Full GC (System.gc()) [PSYoungGen: 15361K->0K(372224K)] [ParOldGen: 83468K->98200K(172032K)] 98829K->98200K(544256K), [Metaspace: 9989K->9989K(1058816K)], 0.3036213 secs] [Times: user=1.03 sys=0.00, real=0.30 secs]

对于Full GC：

1 [PSYoungGen: 15361K->0K(372224K)] ：年轻代：垃圾收集前->垃圾收集后(年轻代堆总大小)

2 [ParOldGen: 83468K->98200K(172032K)] ：年老代：垃圾收集前->垃圾收集后(年老代堆总大小)

3 98829K->98200K(544256K), ：垃圾收集前->垃圾收集后(总堆大小)

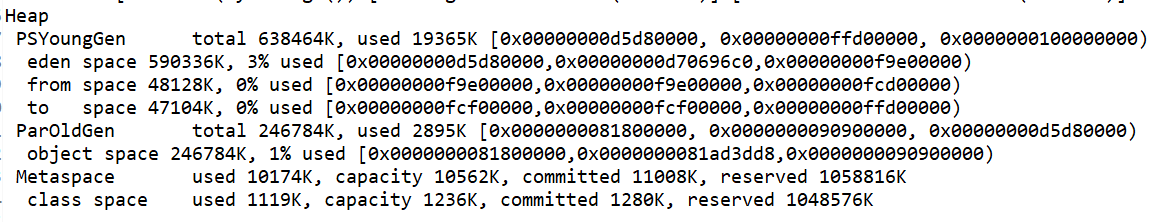
4 [[Metaspace: 9989K->9989K(1058816K)], Metaspace空间信息，同上

5 0.3036213 secs：整个GC过程持续时间

6 [Times: user=1.03 sys=0.00, real=0.30 secs] 分别表示用户态耗时，内核态耗时和总耗时。也是对gc耗时的一个记录。

第四条：71.601: [Full GC (Ergonomics) [PSYoungGen: 20555K->0K(367616K)] [ParOldGen: 104080K->83460K(172032K)] 124636K->83460K(539648K), [Metaspace: 9959K->9959K(1058816K)], 0.2146493 secs] [Times: user=0.64 sys=0.00, real=0.21 secs]

这里可以到full gc的reason是Ergonomics，是因为开启了UseAdaptiveSizePolicy，jvm自己进行自适应调整引发的full gc。



对于底下Heap的输出情况，和上面是完全一致的。

只是增加了堆中每个部分total总大小，used使用情况。

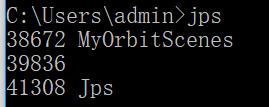
PSYoungGen分为eden space 530336K, 3% used ， from space 48128K, 0% used ， to space 47104K, 0% used三个部分，分别显示了它们的大小和used比例。

ParOldGen分为object space246784K, 1% used，显示了其大小和used比例。

Metaspace中used大小为10174k。

**根据以上日志中显示出垃圾回收的异常情况，在报告的后面说明了处理方法。**

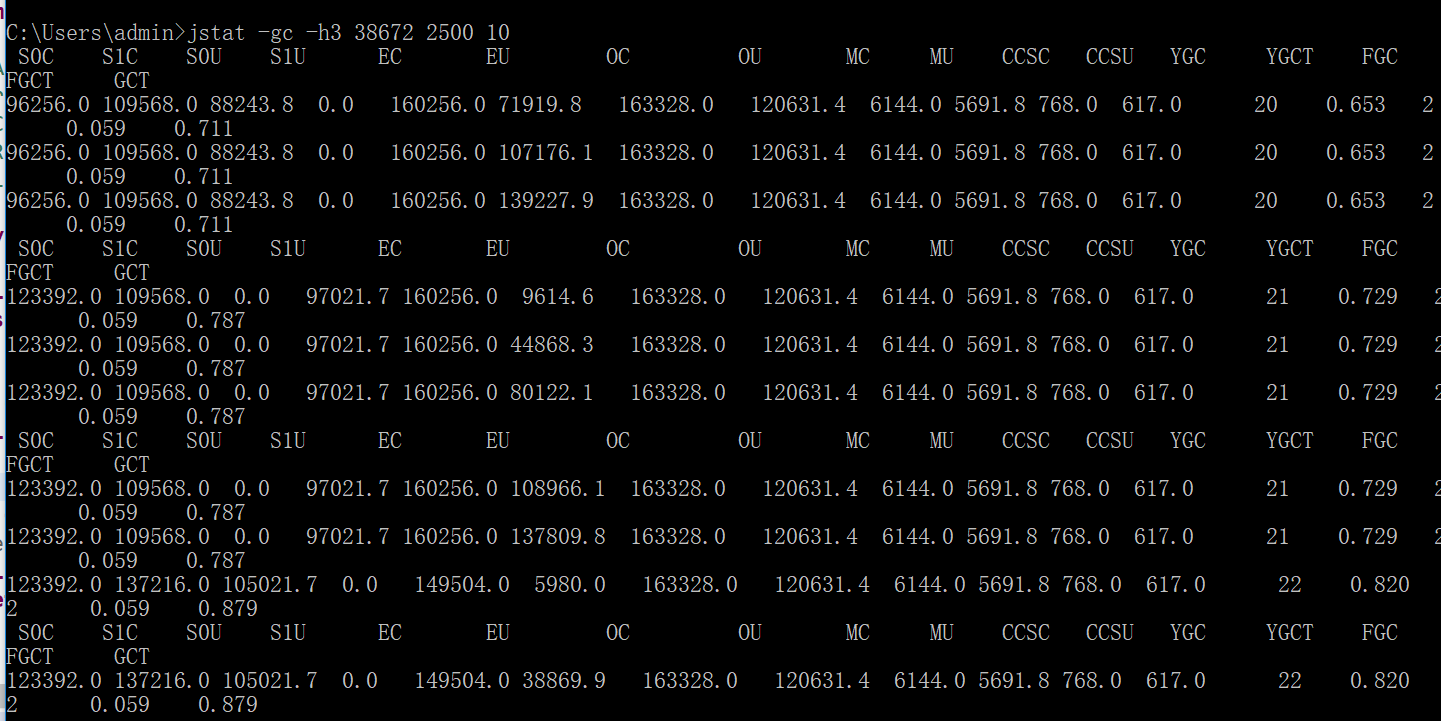
### 用jstat命令行工具的-gc和-gcutil参数



先用jps 命令列出当前的 Java 进程列表，选择进程号38672的进程观察。

jstat -gc -h3 38672 2500 10

**解释**：监控gc的信息，每三行输出一次表头，监控的进程pid为38672，每2500ms输出一次信息，一共输出10次。

****

**分析：**

新生代（Young）：

S0C 第一个幸存者区的总大小；S1C 第二个幸存者区的总大小

S0U第一个幸存者区的已使用的大小；S1U第二个幸存者区的已使用的大小

EC 伊甸区的总大小；EU 伊甸区已使用的大小

老年代（Old）

OC 老年区的总大小；OU 老年区已使用的大小

Metaspace：

MC 方法区的总大小；MU 方法区已使用大小

CCSC压缩类总大小；CCSU 压缩类已使用大小

其他：

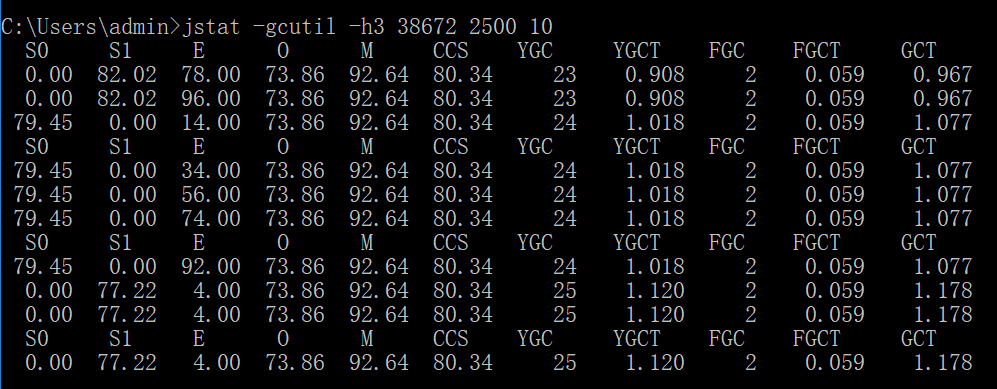
YGC young gc次数；YGCT young gc总时间

FGC full gc 次数；FGCT full gc 总时间

GCT gc的总时间

jstat -gcutil -h3 38672 2500 10

**解释：**监控gc的信息，每三行输出一次表头，监控的进程pid为38672，每2500ms输出一次信息，一共输出10次。



**分析：**

S0、S1 代表两个Survivor区；

E 代表 Eden 区；

O（Old）代表老年代；

M代表Metaspace

YGC（Young GC）代表Minor GC；

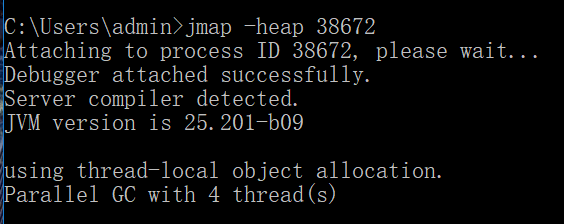
YGCT代表Minor GC耗时；

FGC（Full GC）代表Full GC耗时；

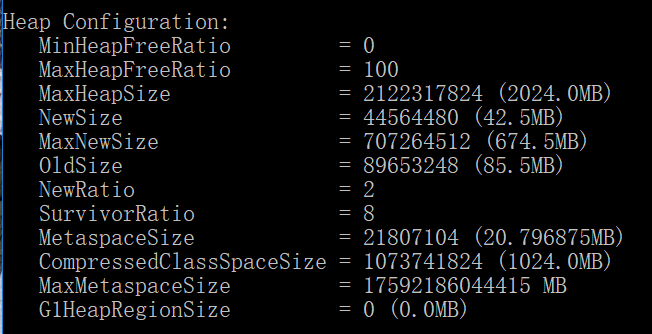
GCT代表Minor & Full GC共计耗时

分析到这儿，程序的gc其实是发生了异常情况的，具体的解决和环境配置在后文说明。

### 使用jmap -heap命令行工具



GC使用的算法：thread-local object allocation.



heap的配置情况：以上多个参数的意义

MinHeapFreeRatio：JVM堆最小空闲比率0

MaxHeapFreeRatio：JVM堆最大空闲比率100

MaxHeapSize：JVM堆的最大大小2122317824

NewSize：JVM堆的‘新生代’的默认大小44564480

MaxNewSize：VM堆的‘新生代’的最大大小707264512

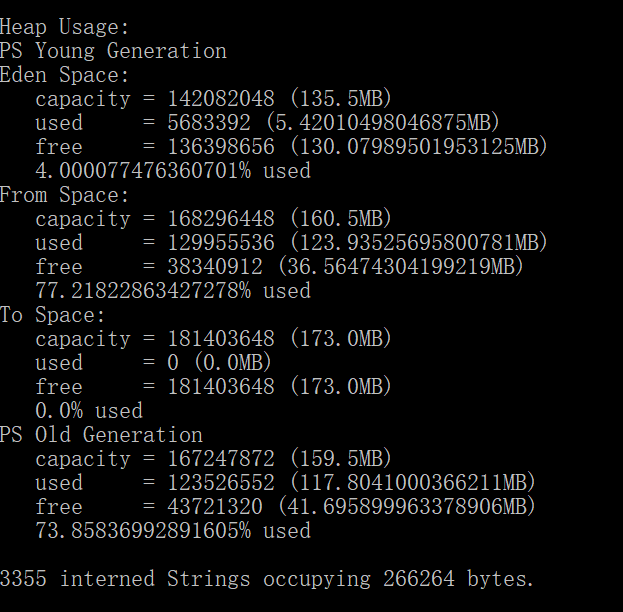
OldSize：JVM堆的‘老生代’的大小89653248

NewRatio：‘新生代’和‘老生代’的大小比率2

SurvivorRatio：年轻代中Eden区与Survivor区的大小比值8

MetaspaceSize=<value>: JVM堆的Metaspace的初始大小21807104

MaxMetaspaceSize=<value>:JVM堆的Metaspace的最大大小



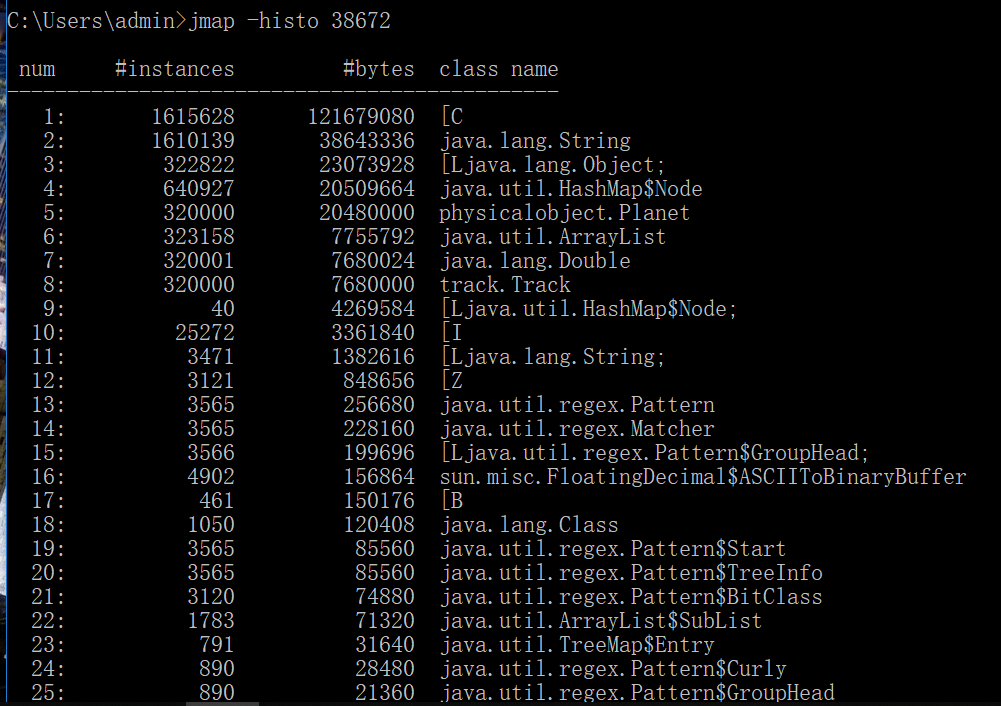
堆的使用情况：

新生代区：Eden区内存分布；From Space: 其中一个Survivor区的内存分布；To Space: 另一个Survivor区的内存分布

年老代区：内存分布

**补充：jmap -histo命令行工具**

说明：instances（实例数）、bytes（大小）、classs name（类名）。它基本是按照使用使用大小逆序排列的。

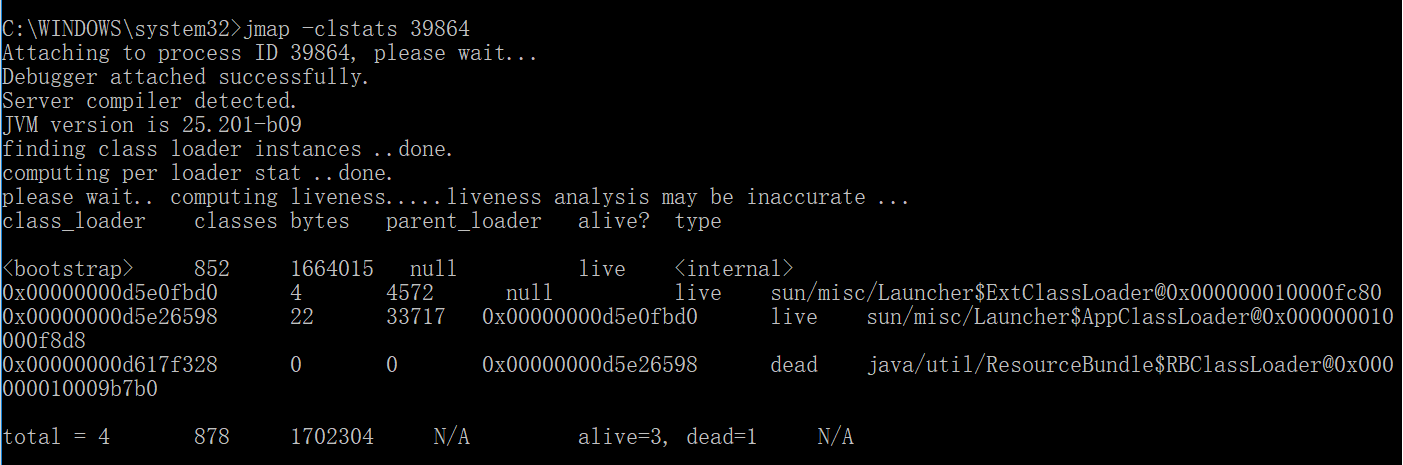


其中：[C is a char[] [I is a int[] [B is a byte[]

### 使用jmap -clstats命令行工具

以管理员身份运行cmd，可以得到如下图。

jmap -clstats 39864打印Java类加载器的智能统计信息，对于每个类加载器而言，它的名称，活跃度，地址，父类加载器，它所加载的类的数量和大小都会被打印。此外，包含的字符串数量和大小也会被打印。



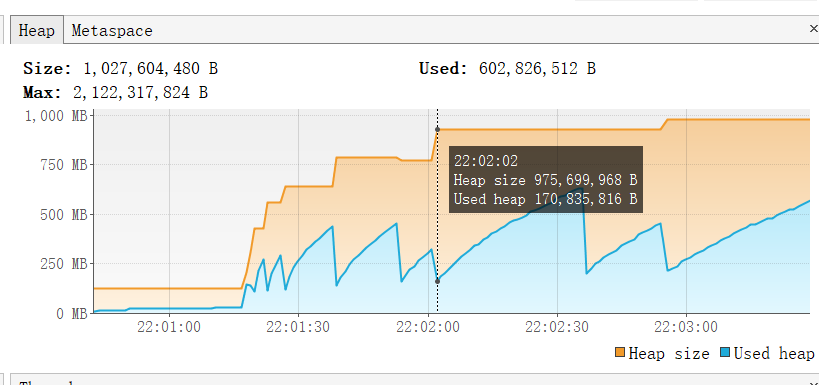
### 使用jmap -permstat命令行工具

JDK版本在8之后，不使用该指令。

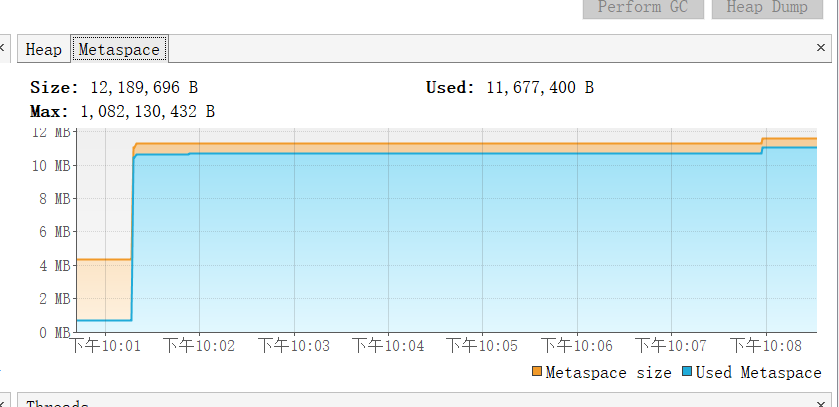
### 使用JMC/JFR、jconsole或VisualVM工具

堆的分配和占用:

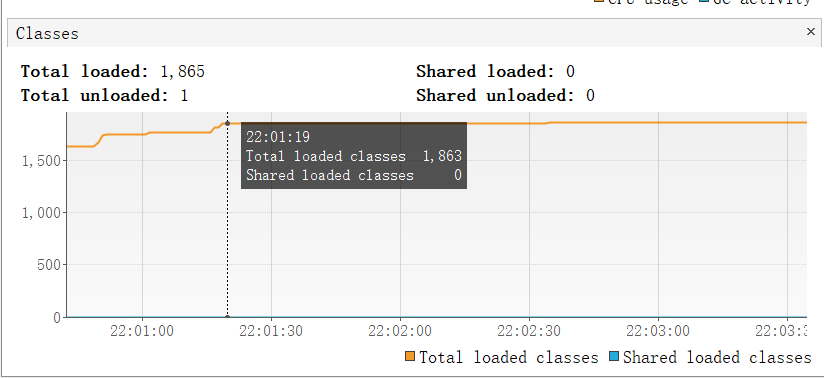
我们可以看出，在readFile开始运行的时候，Heap的使用突然增大。由于构建的对象较多，整体上内存的heap的使用率也是比较高的。

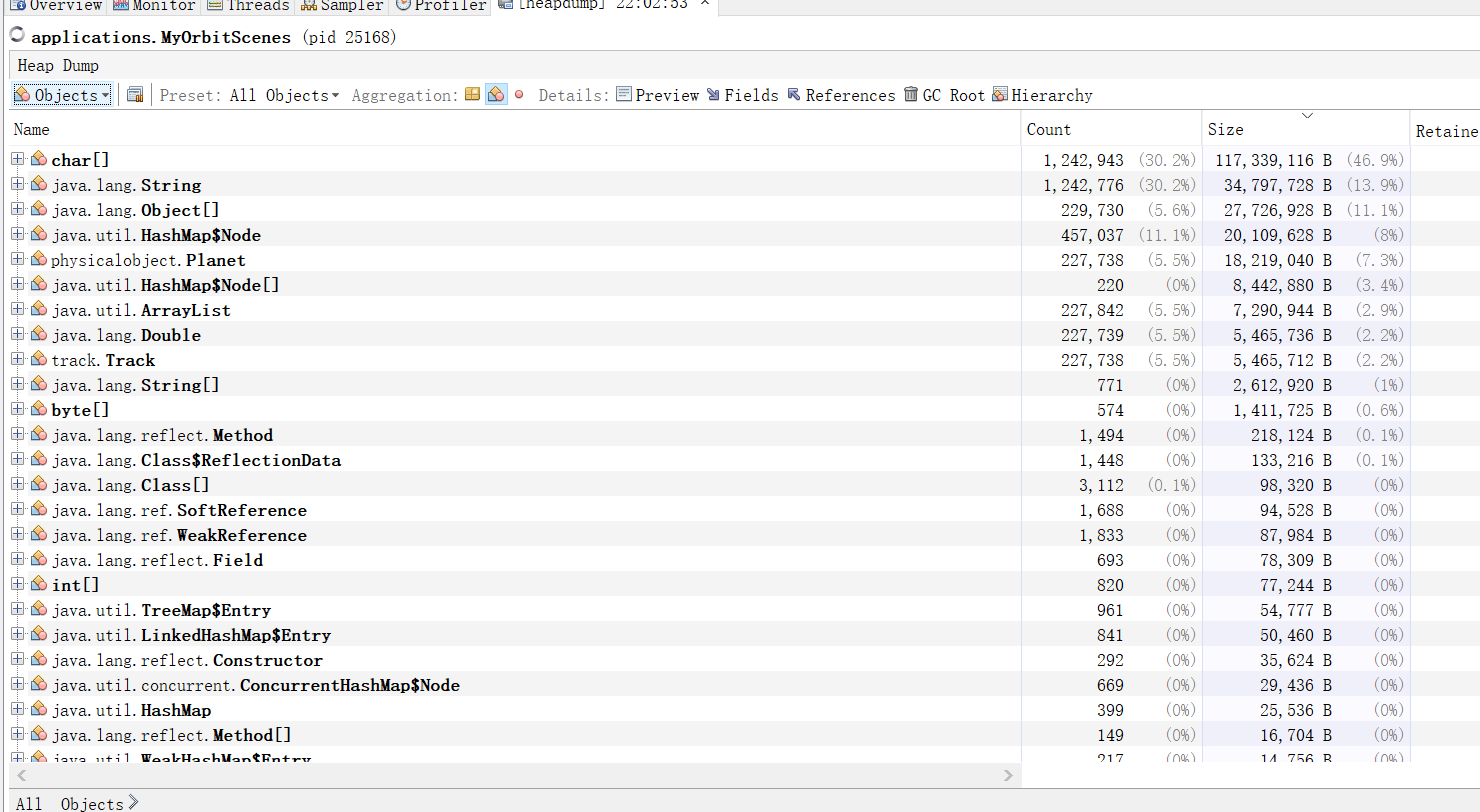


Metaspace情况：



装载的类的实例统计：





### 分析垃圾回收过程

根据分代的垃圾回收策略，我们对垃圾回收的过程可以做出如下的总结。

首先从jstat -gc打印出的结果，我们可以清楚的看到年轻代分区的情况

1. 新生代内存按照8:1:1的比例分为一个eden区和S0、S1区。大部分对象在Eden区中生成，回收时先将eden区存活对象复制到一个S0区，然后清空eden区。

2. 当这个S0区也存放满了时，则将eden区和S0区存活对象复制到另一个S1区，然后清空eden和这个S1区，此时S0区是空的，然后将S0区和s1区交换，即保持S1区为空，如此往复。

3. 这里也解释了为什么命令行jstat -gc每个2500ms打印结果的时候，S0和S1的大小交替变换。

然后再从日志分析结果，当我-XX:NewSize=500m：设置年轻代初始值为500M时。发现gc时大部分的的在年轻代会被回收，大约只有有20M大小的对象会进入老年代。

1. 一方面根据我程序实现时，对象的存活周期大多较短，另一方面，所有新生成的对象首先都是放在年轻代的。年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象。

2. 当survivor1区不足以存放 eden和S0的存活对象时，就将存活对象直接存放到老年代。若是老年代也满了就会触发一次Full GC，也就是新生代、老年代都进行回收

3. 新生代发生的GC也叫做Minor GC，MinorGC发生频率比较高(不一定等Eden区满了才触发)

我们再结合着jmap -heap打印出的结果来分析，以下主要讨论年老代和Metaspace。

1. 在年轻代中经历了N次垃圾回收后仍然存活的对象，就会被放到年老代中。因此，可以认为年老代中存放的都是一些生命周期较长的对象。

2.当老年代内存满时触发Full GC，Full GC发生频率比较低，老年代对象存活时间比较长，存活率标记高。

3. Metaspace用于存放静态文件，如Java类、方法等。结合着日志，我们会发现Metaspace的大小变化总是不大，事实上Metaspace对垃圾回收没有显著影响。Metaspace包含JVM用于描述应用程序中类和方法的元数据，是由JVM在运行时根据应用程序使用的类来填充的。

### 配置JVM参数并发现优化的参数配置

先说明一下参数配置的依据：

1. 在gc日志，我们很明显的看出Minor gc的时候，显示Allocation Failure的原因是年轻代中没有足够区域能够存放需要分配的数据而失败。因此我们需要增加年轻代的大小。刚开始-XX:NewSize=650m：设置年轻代初始值为650M。发现还是不够，于是又加大了1g。

2. 从jstat -gc的打印结果，发现每次ygc之后survivor空间基本是空的，说明新生对象产生快，生命周期也短，原本设计的survivor空间没有派上用场。因此考虑调整young generation分区的大小设置。

参数配置:

-XX:InitialHeapSize=2120867840

-XX:MaxHeapSize=2120867840

-XX:MaxNewSize=2147483648

-XX:NewSize=2147483648

-XX:+PrintGC

-XX:+PrintGCDetails

-XX:+PrintGCTimeStamps

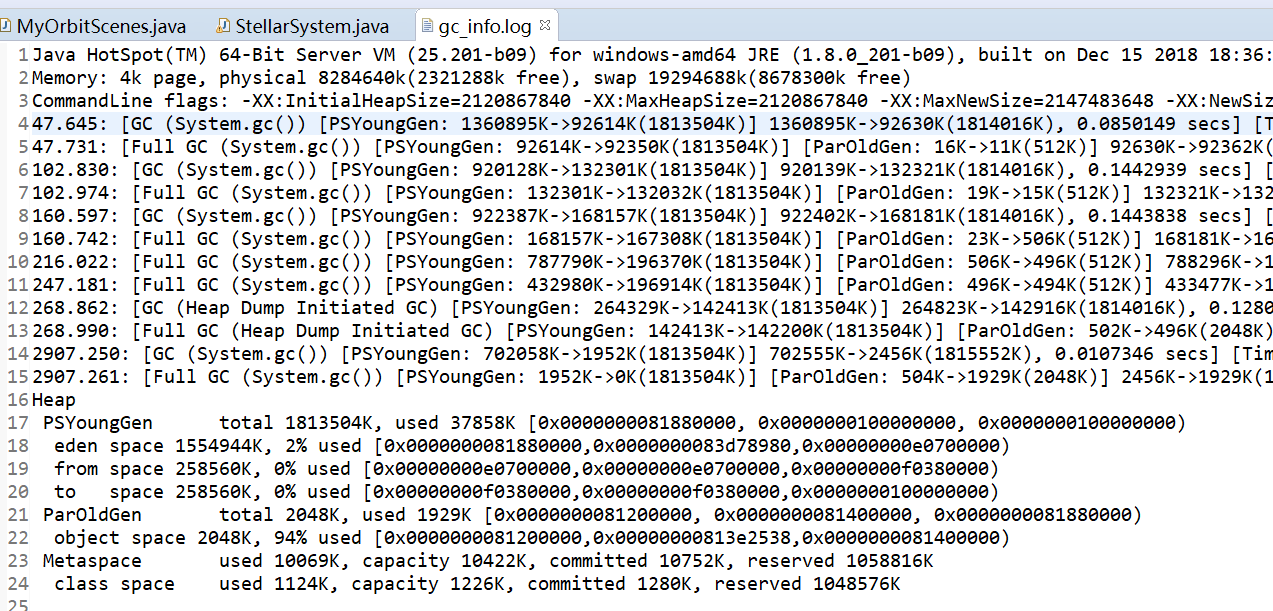
-XX:+UseCompressedClassPointers

-XX:+UseCompressedOops

-XX:-UseLargePagesIndividualAllocation

-XX:+UseParallelGC

重新执行（1）：GC (Allocation Failure)和Full GC (Ergonomics)终于没了，而且我们可以发现垃圾回收的时间减少了很多。

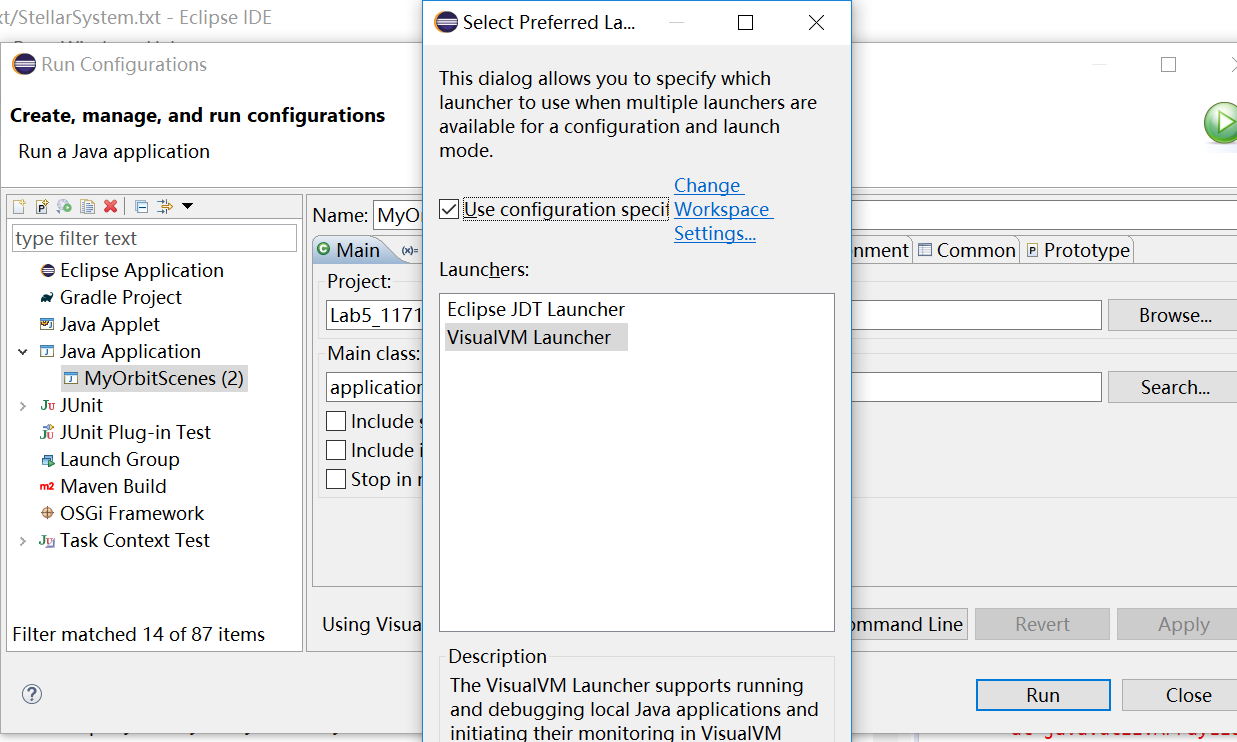


重新执行（2）：

截图太多就不粘了，由于我们增大了young generation，同时old generation会有一定的减小，但是观察YGCT代表Minor GC耗时；FGC（Full GC）代表Full GC耗时；GCT代表Minor & Full GC共计耗时。这些耗时都有减少。

## Dynamic Program Profiling

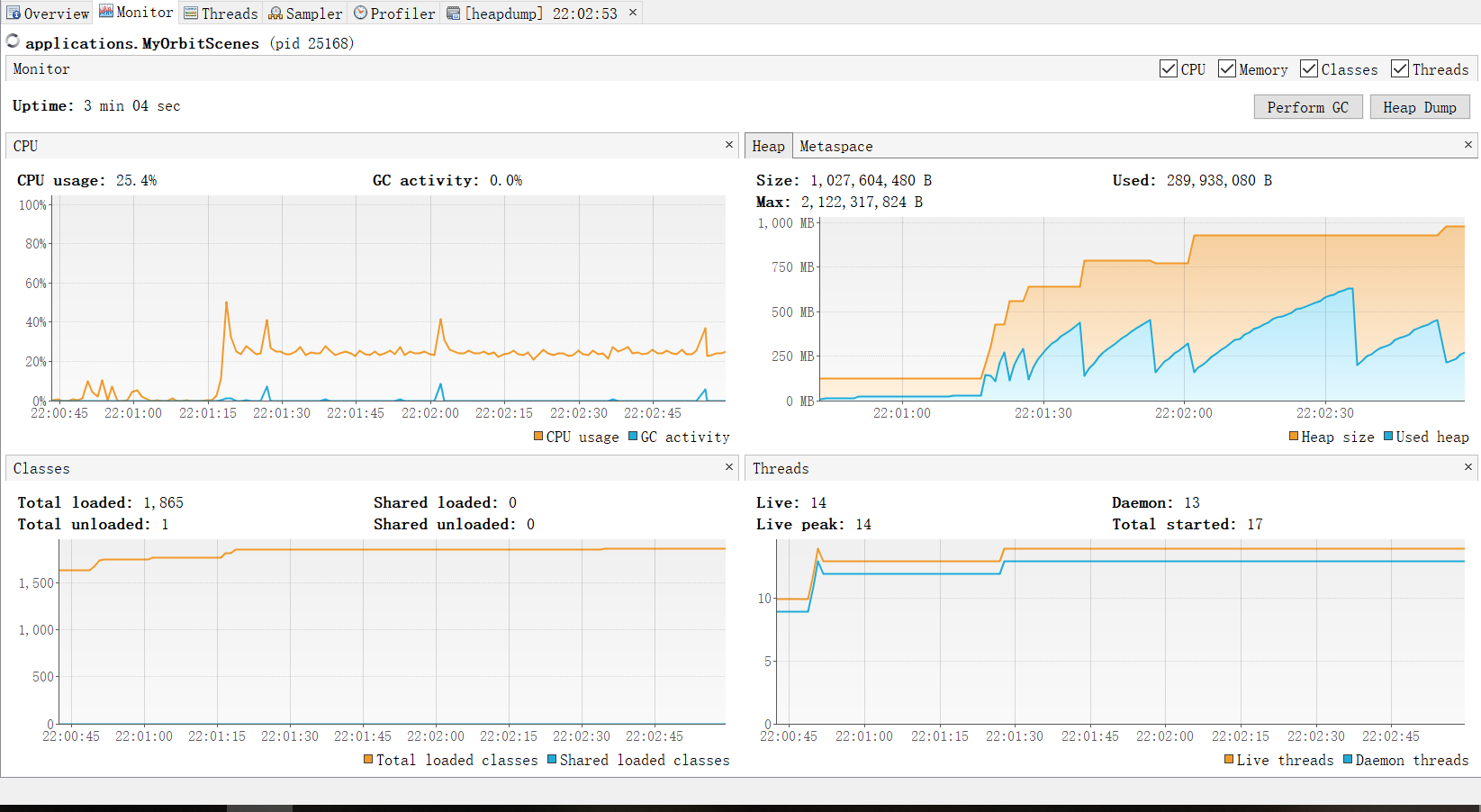
配置环境的部分在前面实验环境配置的部分已经说明，运行VisualVM的方法如下图：



在选择读文件阶段，就将VisualVM profiler 设定为“自动更新结果”，选择CPU Profiling或者memory profiling，从头开始动态监控程序的性能。

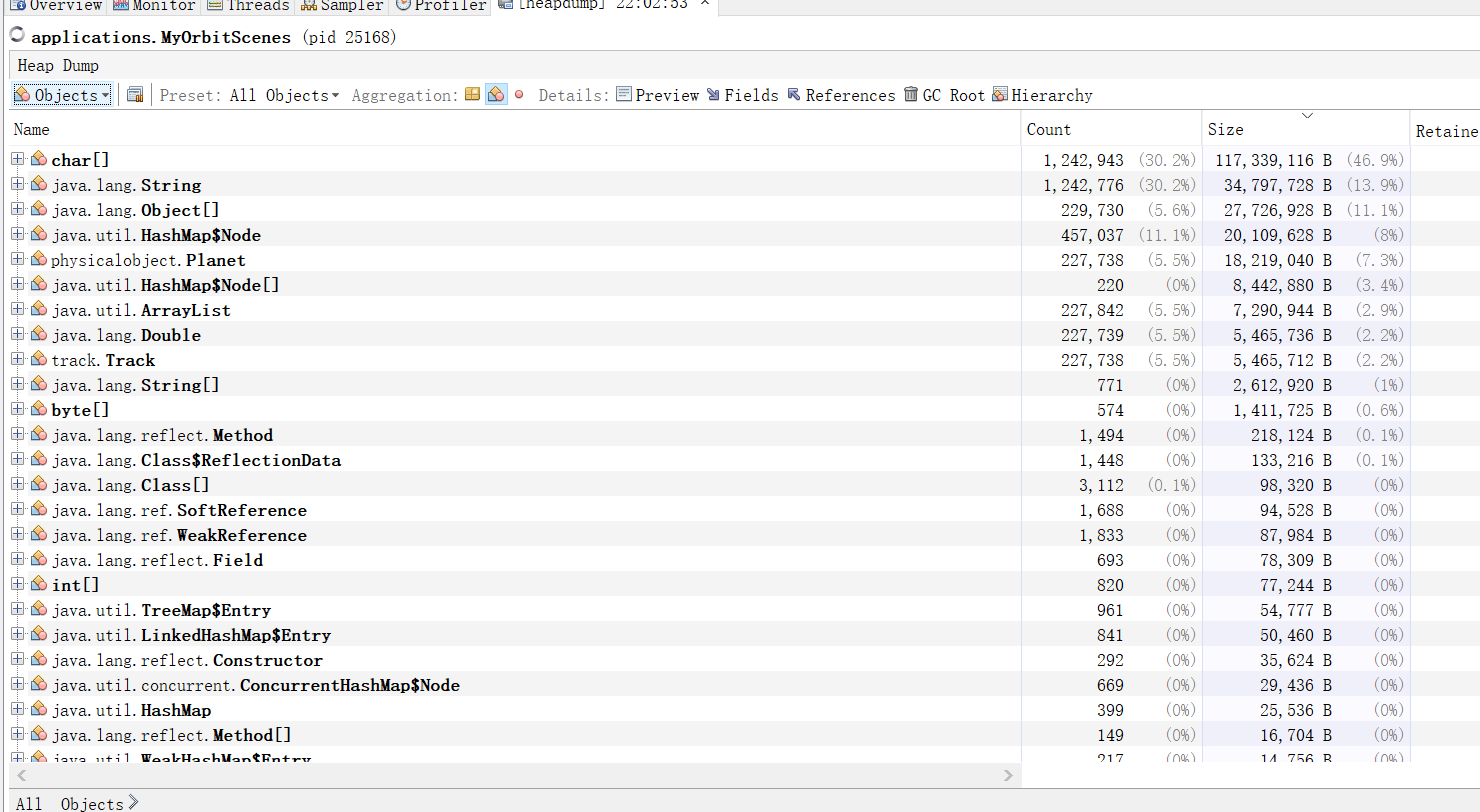
在某一时刻截图如下**（关于CPU和memory后面分析）：**

在监视标签内，我们可以看到实时的应用程序内存堆以及Metaspace的使用情况,CPU的使用率和垃圾回收活动。



此时heapdump情况如下：堆转储的类视图

堆转储的摘要包括转储的文件大小、路径等基本信息，运行的系统环境信息，也可以显示所有的线程信息。如图中可以看出char[]和java.lang.String占用了较大内存

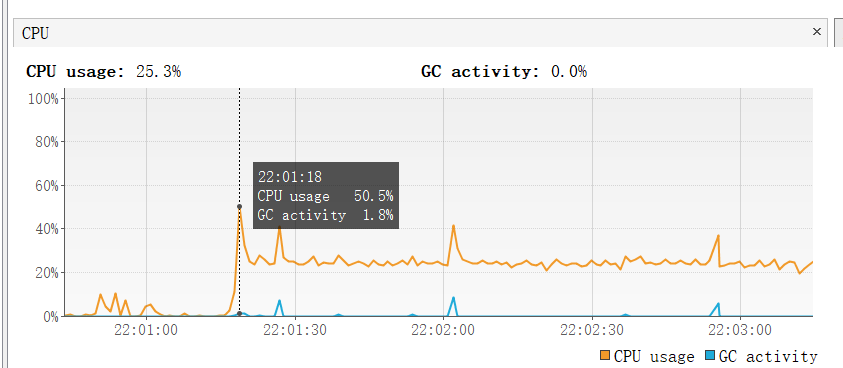


### 使用JMC或VisualVM进行CPU Profiling

我们可以通过 VisualVM 的监视标签和 Profiler 标签对应用程序进行 CPU 性能分析。

**一. 先看监视标签**

在监视标签内，我们可以查看 CPU 的使用率以及垃圾回收活动对性能的影响。在程序刚启动时，CPU usage达到了最高的50.5%。在其他时候，过高的 CPU 使用率可能是由于我们的轨道系统构建中中存在低效的代码，整体上看，垃圾回收活动并不频繁，没有占用了较高的 CPU 资源。



**二. 再看CPU profiler标签**

在CPU 性能分析profiler标签 ,VisualVM 会检测应用程序所有的被调用的方法。当进入一个方法时，线程会发出一个“method entry”的事件，当退出方法时同样会发出一个“method exit”的事件，这些事件都包含了时间戳。然后 VisualVM 会把每个被调用方法的总的执行时间和调用的次数按照运行时长展示出来。

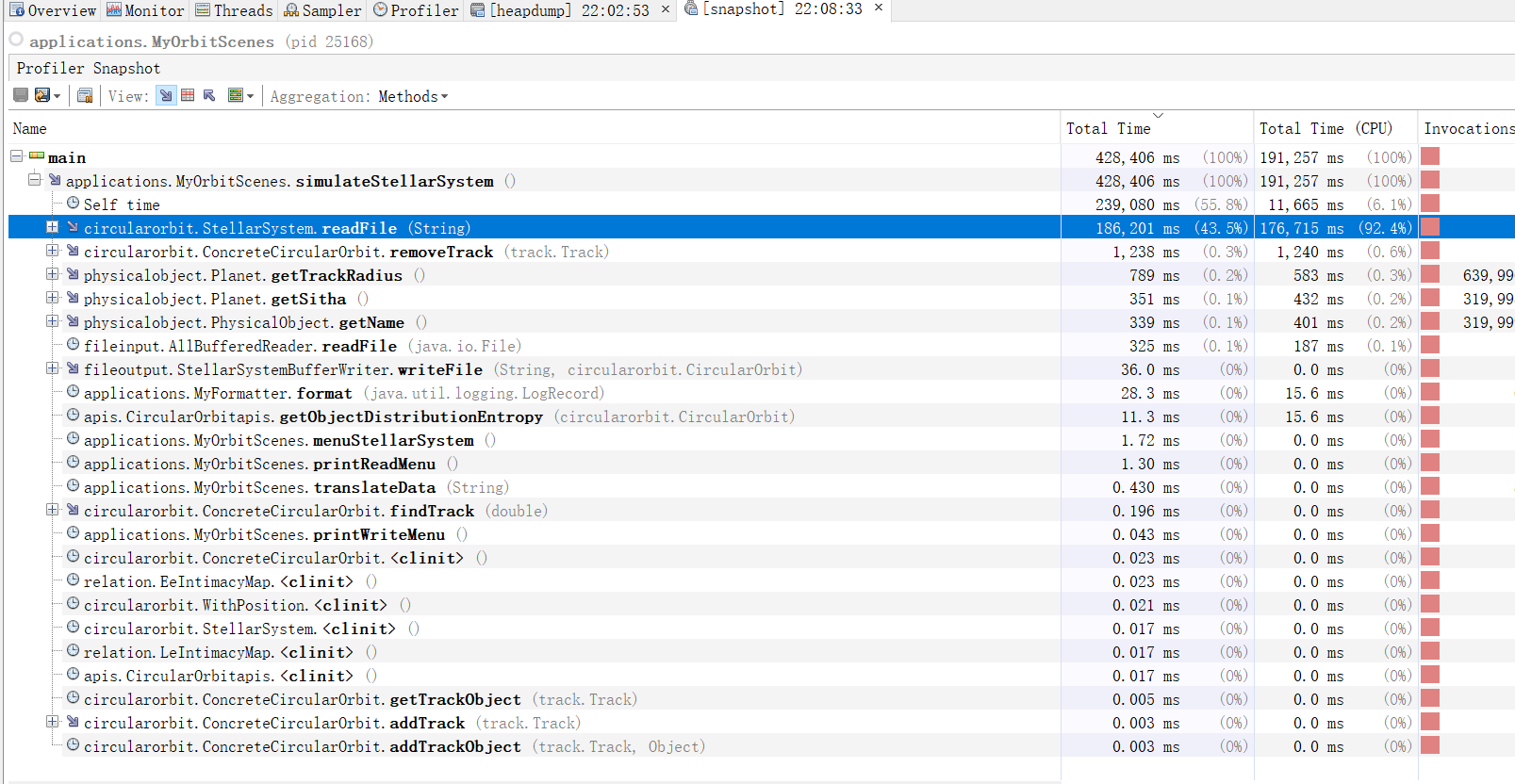
Self time：我们按照Total Time进行排序之后，发现排在第一的是Self time，这个其实对于我们不重要，解释起来，Self time是在方法本身花费的挂钟时间（包括等待/休眠的时间），它是时间处理器时间，因此它不包括等待，睡眠等所花费的时间。

circularorbit.StellarSystem.readFile (String)：对于方法执行的时间，不出所料，耗时最长的是readFile，它主要运行的时间不是在读入一篇txt文件，这个只需要几百ms，而是在将物体添加到轨道上并且根据半径进行排序。简单地说就是通过读入的文件搭建轨道是个漫长的过程。

circularorbit.ConcreteCircularOrbit.removeTrack (track.Track)：移除一条轨道也需要花费不少的时间，因为我们需要先移除轨道上的物体，然后再删除轨道

fileoutput.StellarSystemBufferWriter.writeFile (String, circularorbit.CircularOrbit)：写文件运行所需要的时间也在考虑之中，毕竟有几十万行的数据。

其他方法：主要花费的时间在于没有优化彻底的查找方案，比如获得一个物体的某个属性，需要去set中遍历，造成了不少的时间消耗；另外在控制台的打印，写入日志文件也是需要一定时间的。



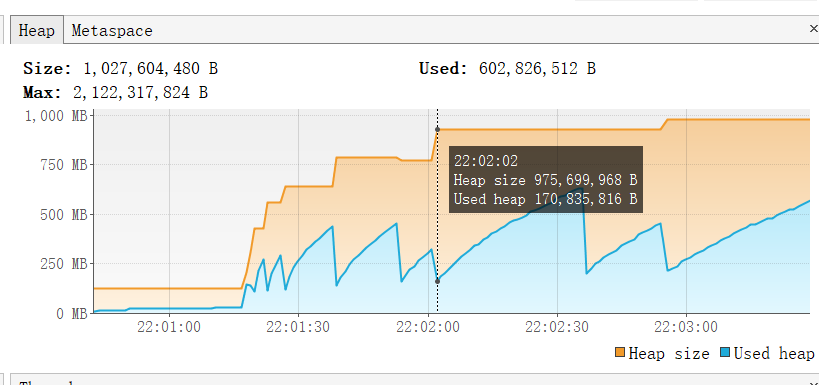


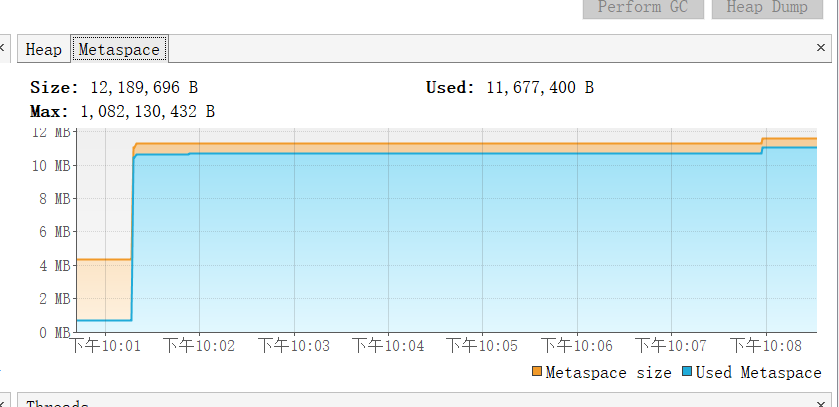
### 使用VisualVM进行Memory profiling

**一. 监视标签**

在监视标签内，我们可以看到实时的应用程序内存堆以及Metaspace区域的使用情况。

我们可以看出，由于构建的对象较多，内存的heap的使用率也是比较高的



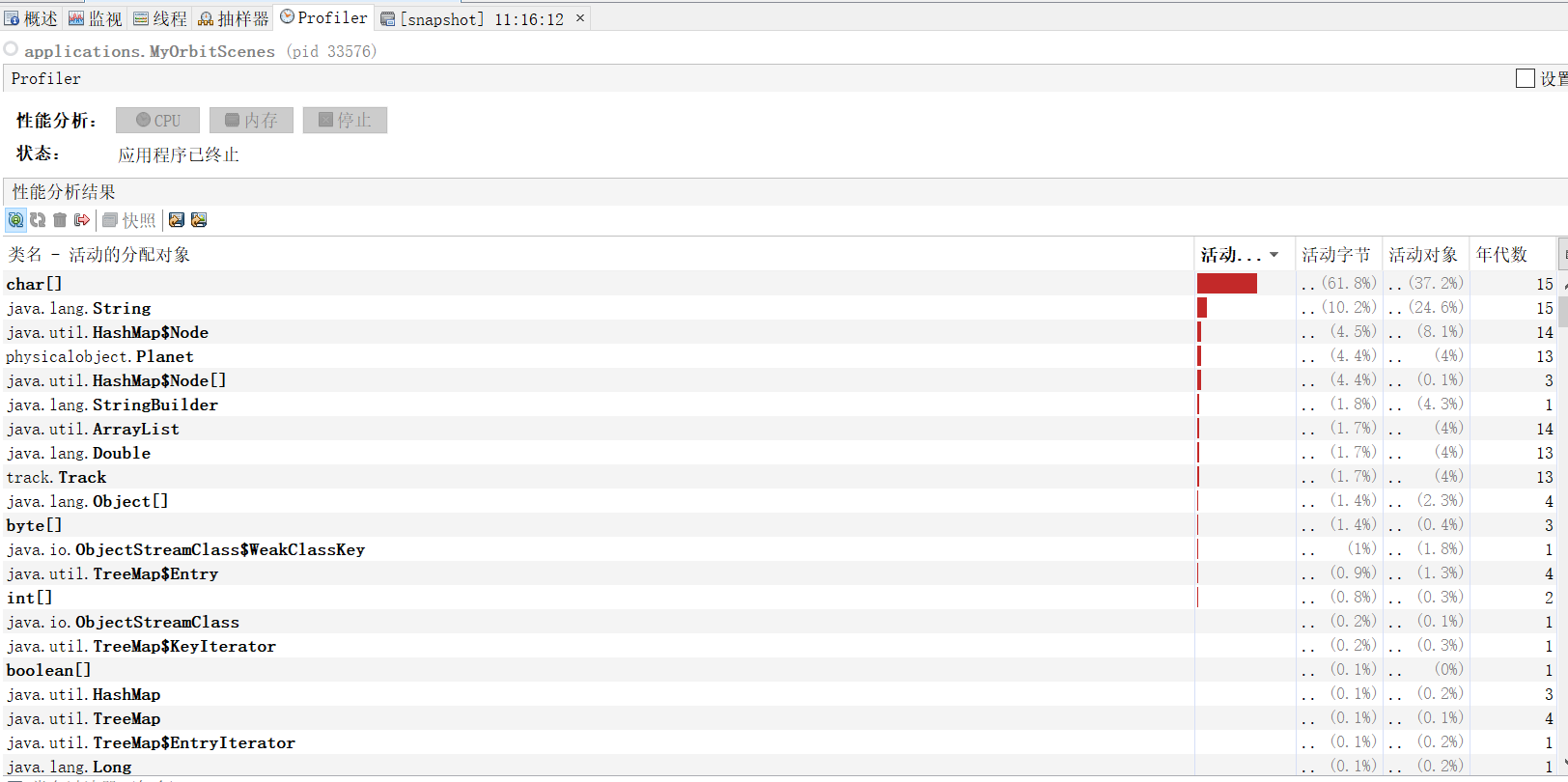


**二. memory profiler**

本以为memory profiler运行起来和CPU profiler一样，可是同样的操作却一直报错，或者就是已经在运行，却一直不显示活动的分配对象。

后来的处理方式是以管理员身份运行cmd，此时打开java visualvm，问题才得以解决。





分析：

通常考虑，字符串对象中就有一个 char[]数组，其他对象中也可能封装char[]数组，先检查一下String对象有多少个，算上其他可能封装char[]数组的对象，也差不多这个数值了。

同理int[]和byte[]也是这个原因占用了很多内存。

但是事实上我们更应关注自己所创建的实例，因为我们解析的时候不断创建字符串，因此String对象毫无疑问有很多。从源码上来看,HashSet实际上为(key,null)类型的HashMap,因此我们使用hashset后会存在很多java.util.HashMap$Node[]。

其余的内存占用居前的类型就很好理解了，因为Planet、track等正是我们构建轨道一直在使用的类型。

总体分析，这些耗费内存最多类型的执行正常。

## Memory Dump Analysis and Performance Optimization

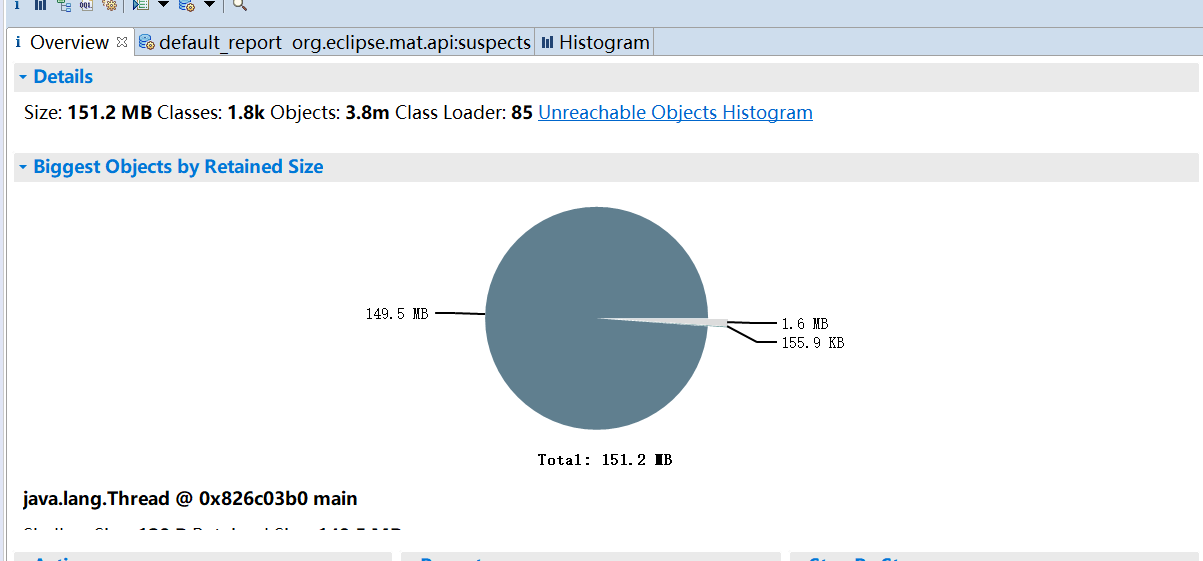
### 内存导出

使用VisualVm进行堆转储后，导出.hprof文件，并将其另存为到本地。

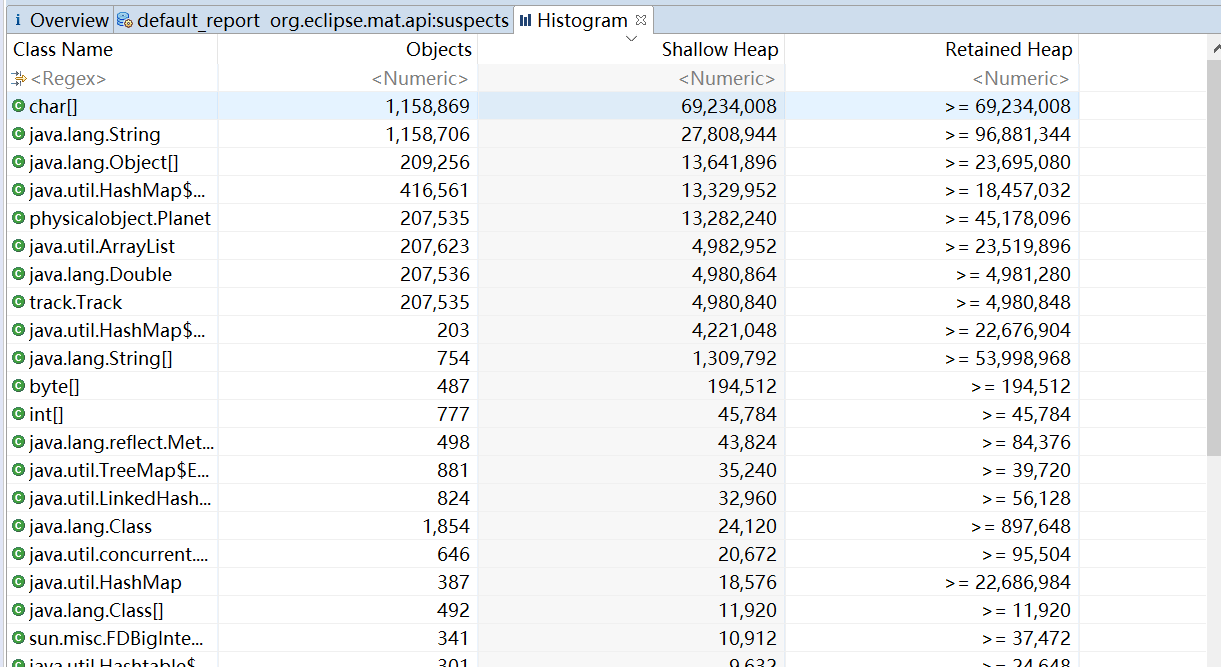
### 使用MAT分析内存导出文件

**Overview：**

内存在“<system class loader>”加载的“java.lang.Thread”的一个实例中累积。

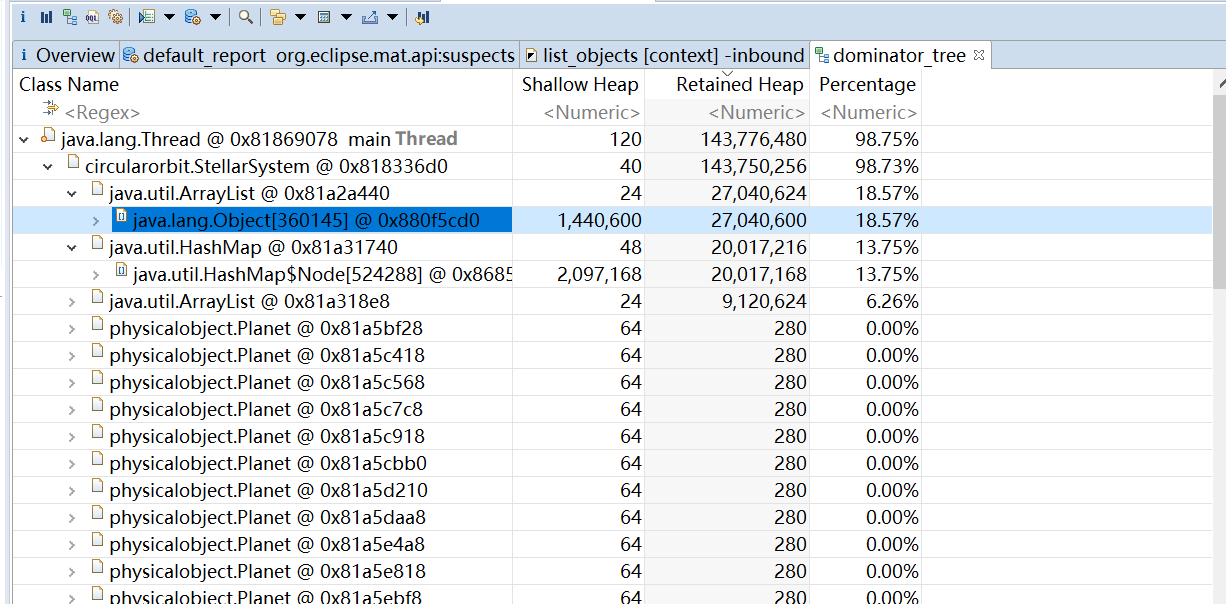


**histogram视图：**展示了此刻内存中存储的各类型的实例数量以及所占用内存的情况。

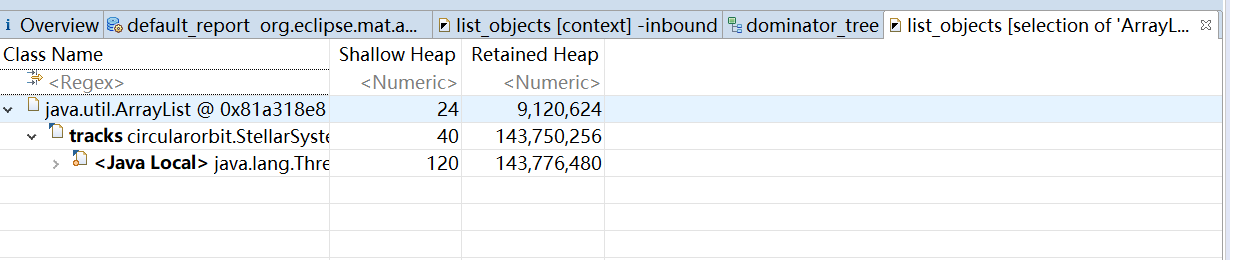


**dominator tree视图：**

各实例之间的引用关系:



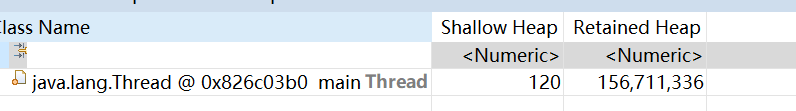
右键选择List Objects -> with incoming reference，这可以列出ArrayList中对象的引用路径：



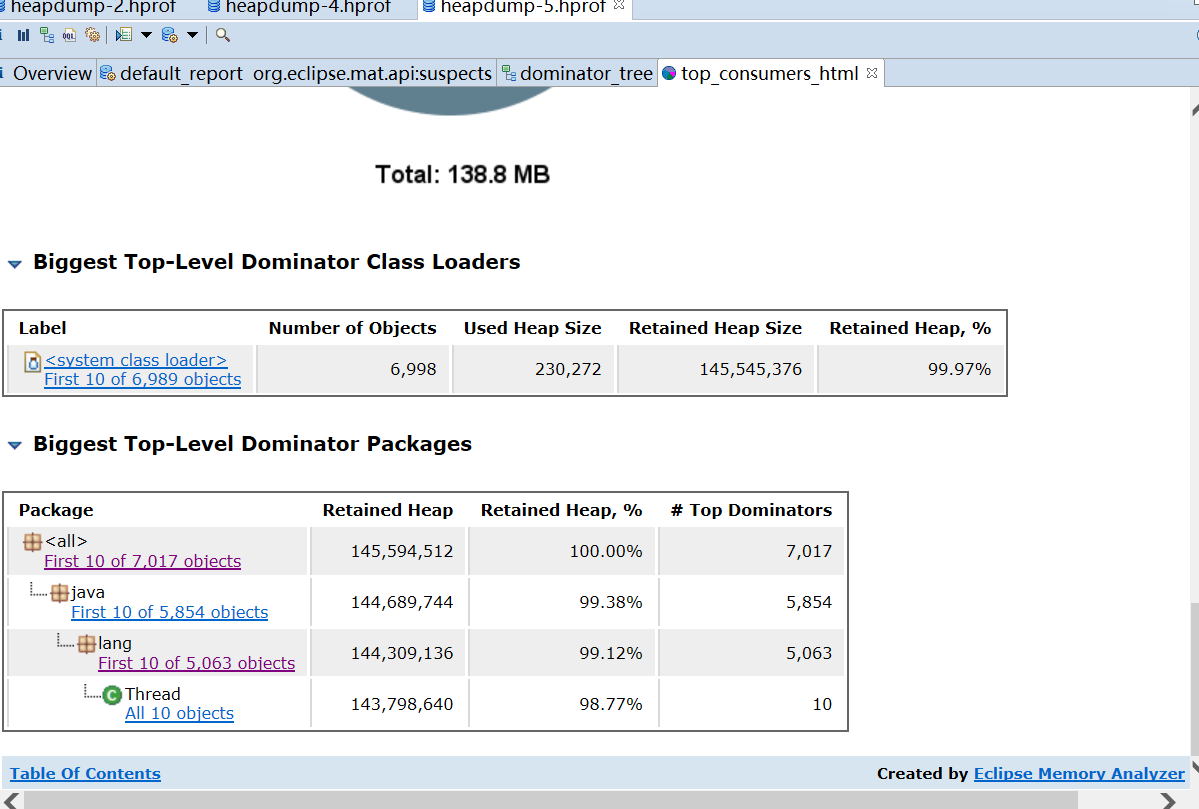
查看一个对象到RC Roots的引用链

通常在排查内存泄漏的时候，我们会选择exclude all phantom/weak/soft etc.references，意思是查看排除虚引用/弱引用/软引用等的引用链，因为被虚引用/弱引用/软引用的对象可以直接被GC给回收，我们要看的就是某个对象否还存在Strong 引用链（在导出HeapDump之前要手动出发GC来保证），如果有，则说明存在内存泄漏，然后再去排查具体引用。

据观察，几乎所有的GC root都是Thread或者System Class。



**top consumer视图：**

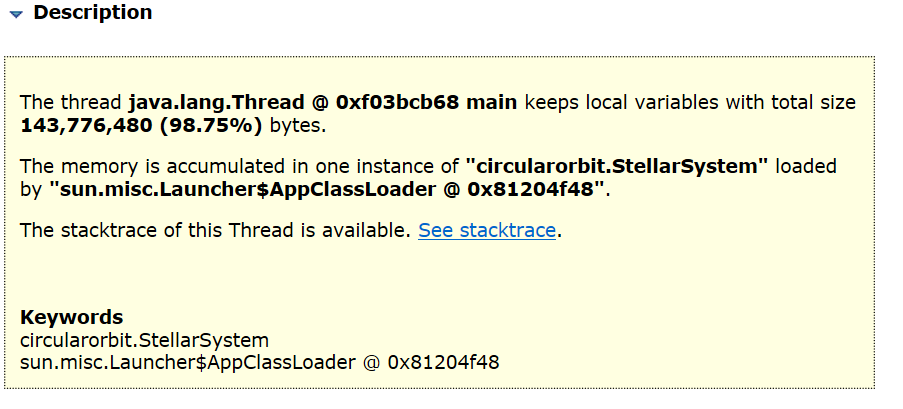


**查看 leak suspects report：**

刚开始我java.lang.Thread @ 0xf03bcb68 main占用堆的比例特别高，于是系统地对leak suspects report进行的分析，最后得出的结论是没有发生内存泄漏，主要的原因是我程序中对象的存活时间比较长，list和hashmap常伴随着对象之间的映射关系，因此在它们gc的回收就会比较延后。

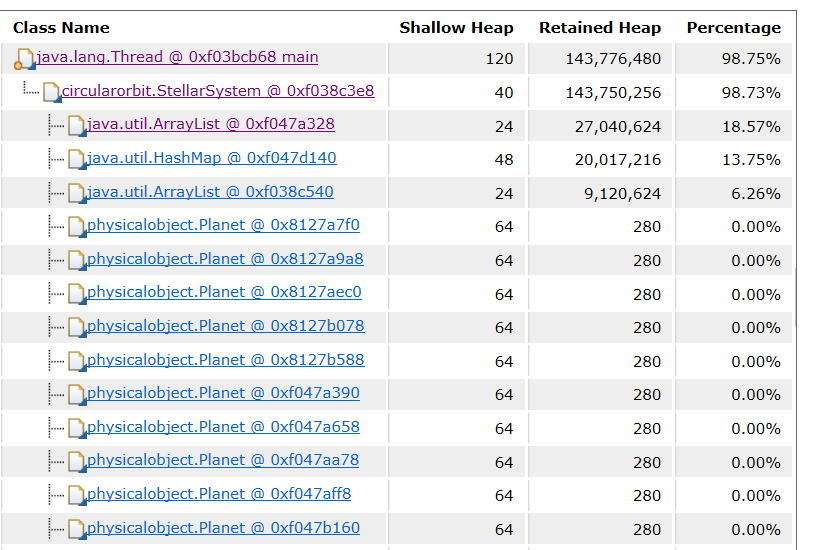
具体的分析过程如下：

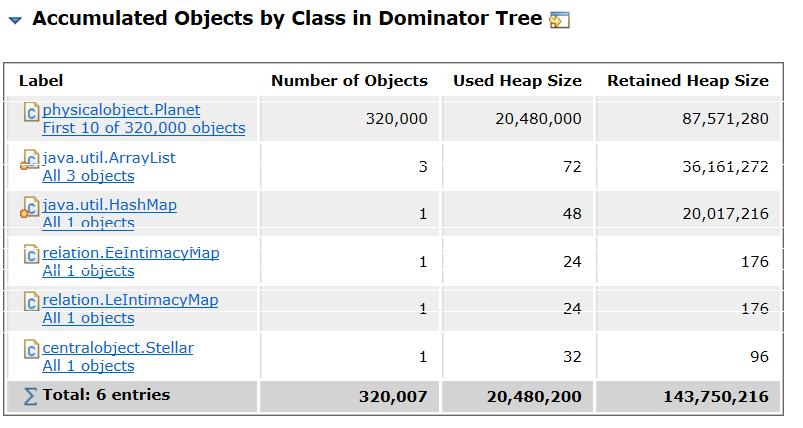
1. 内存疑似泄漏的描述如下。



2. 主要看以下的两张图，首先就可以看到Circularorbit.StellarSystem的shallow heap和retained heap是非常不成比例的，这说明这个在其中持有了大量对象的引用，。于是我们根据retained heap的大小逐步对其支配树进行展开，可以发现heap占用较高的主要是Arraylist、HashMap和Planet。当我们选择List objects ->with incoming references，可以对它们的归属了解得更详细。

两个Arraylist主要是track的列表，和每个轨道上轨道物体嵌套列表；HashMap是在WithPositon类中设计的每个物体映射一个角度产生的，而由于文件较大，产生较多的Planet的也是情理之中。

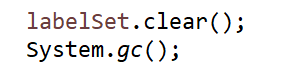




### 发现热点/瓶颈并改进、改进前后的性能对比分析

事实上在第二部分文件读写的时候，代码已经做出了很多的优化，这里只是针对执行时间长、内存分配大进行了程序的优化。

1. 在lab4异常处理的时候，我将所有出现过的行星名称存入一个HashSet，以保证不会出现重复的行星名。MAT同时打开两个堆转储文件，分别打开Histogram时，我发现这个HashSet中的对象在一直增多，回去看代码发现，我一直在往这个HashSet中添加对象，却从没有释放过，于是我在读完文件后：

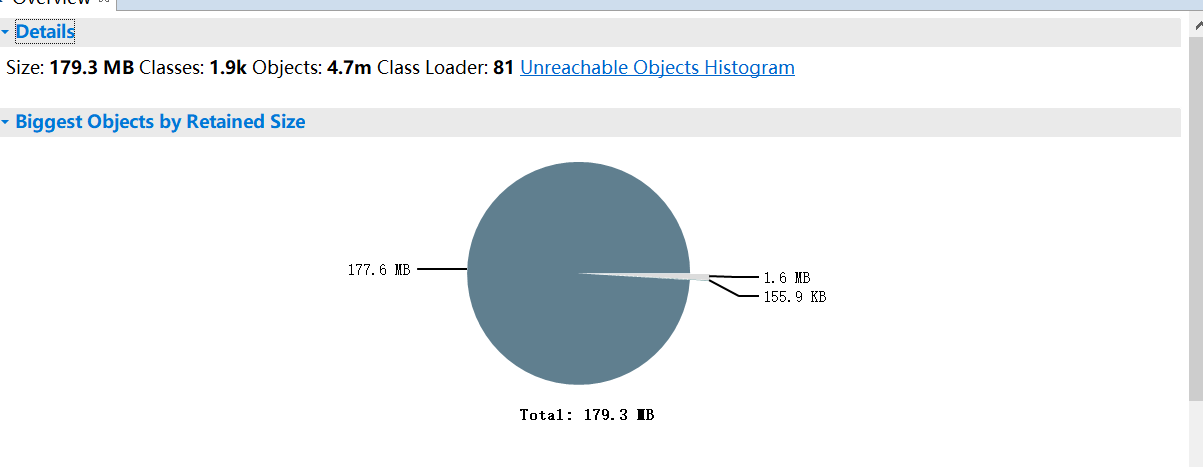


2. 对于labelSet这个集合的对象，是独立于读文件方法外的实例变量，只有在SteallarSystem对象被回收，它才会变成不可达的，才会执行gc。于是我们将其设置为局部变量，在读文件的最后，对其进行垃圾回收。

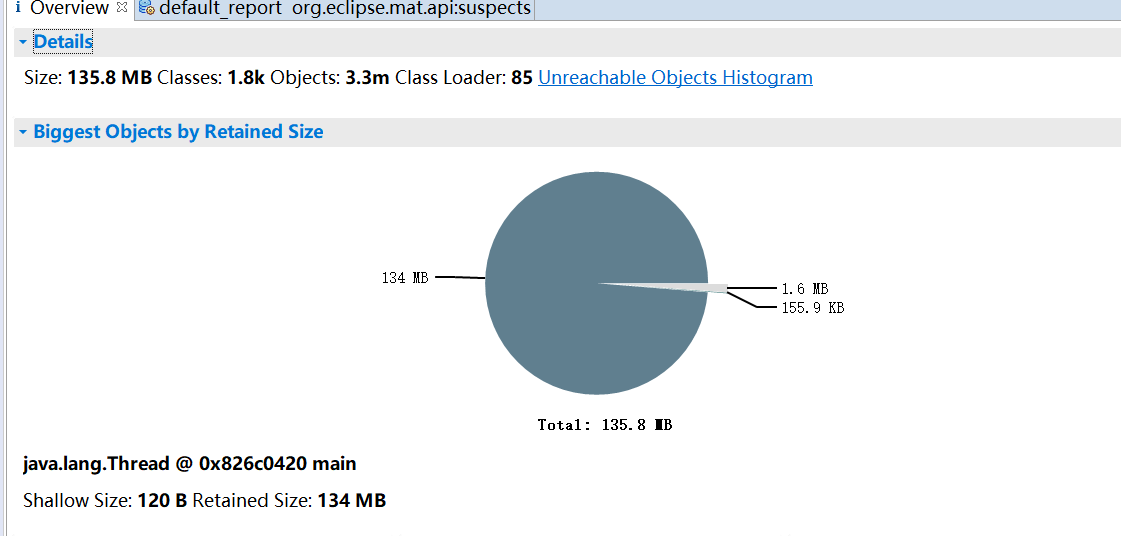
3. 在我们匹配正则表达式，分割字符串的时候，产生了很多字符数组，它们在方法体内是不会回收的，造成了局部的内存呢占用，于是我们可以在某个字符数组使用完后，将其变为null，这样它就可以被回收了。

4. 对于SocialNetworkCircle，其优化的方式和以上完全类似，都是尽量使一个或者是一个集合中的不再使用的对象尽快变成不可达的，从而进行gc。

改进前的Retained Size为179.3MB

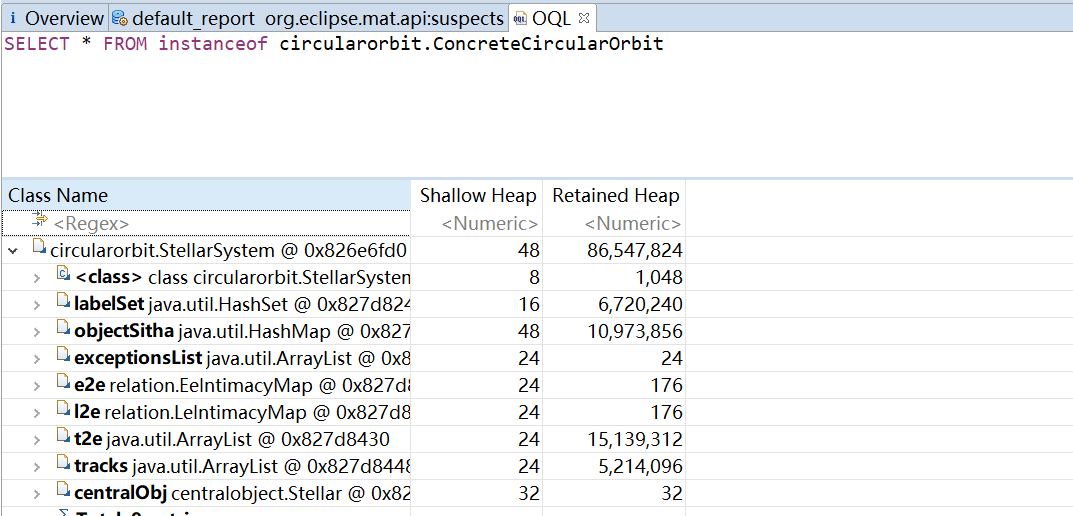


改进后的Retained Size为135.8MB，比优化之前减少了44MB，已经是个很不错的优化了。

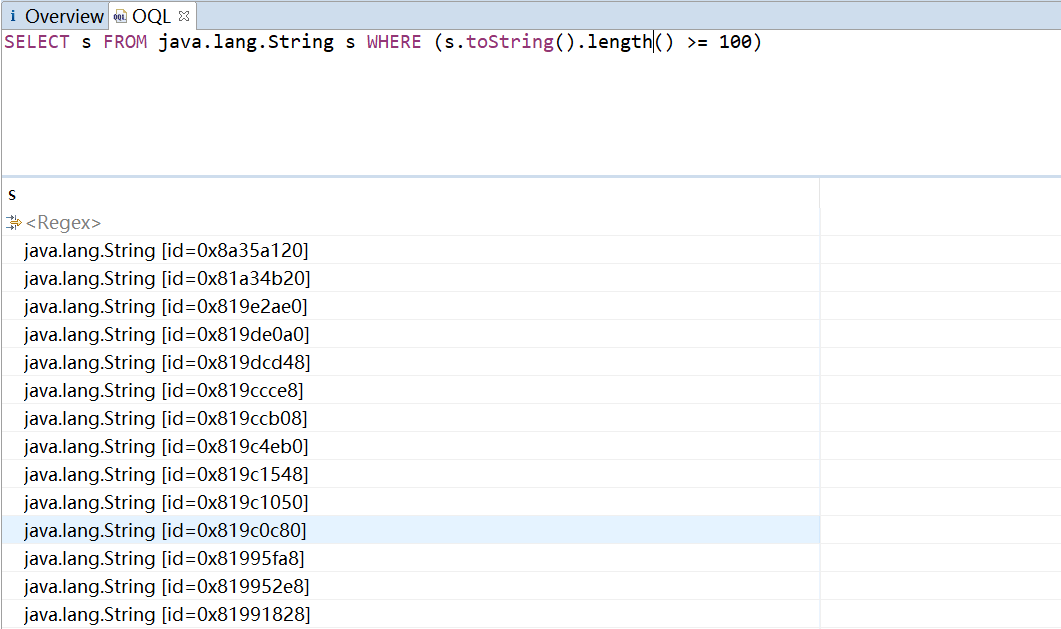


### 在MAT内使用OQL查询内存导出

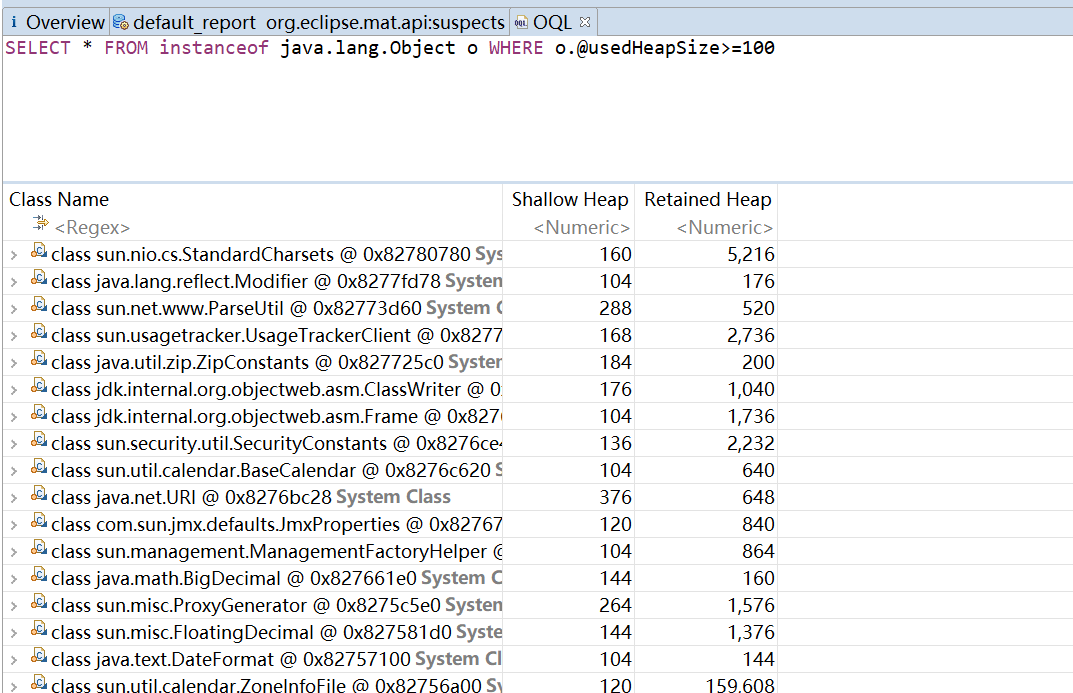
CircularOrbit 的所有对象实例；



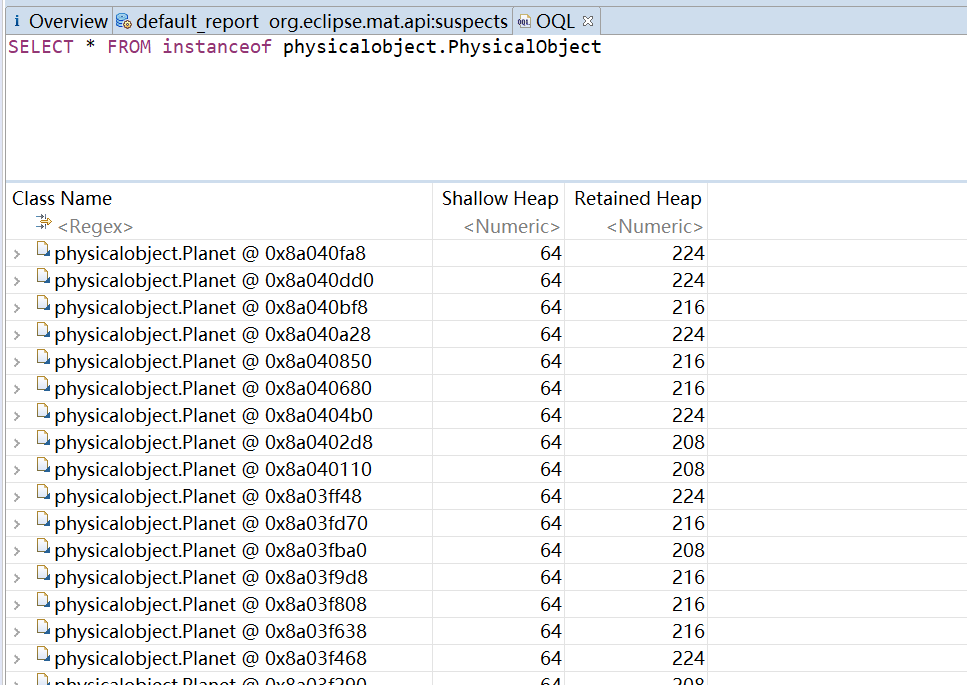
大于长度100的String对象：



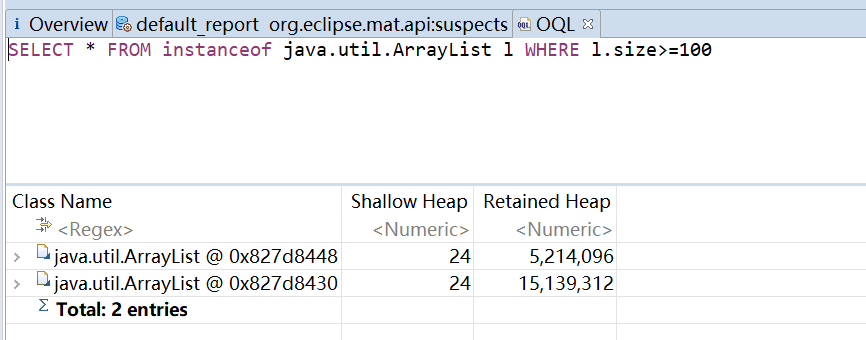
大于特定大小的任意类型对象实例



PhysicalObject（及其子类）的对象实例的数量和总占用内存大小

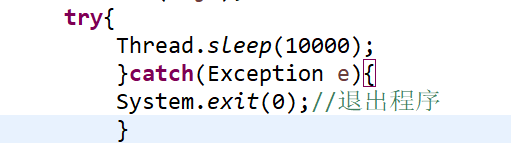


所有包含元素数量大于 100 的 Collections 实例

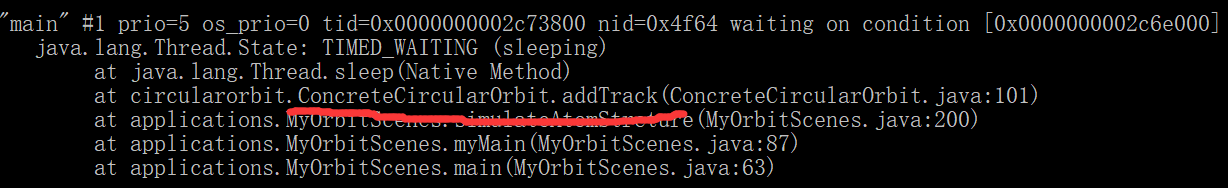


### 观察jstack/jcmd导出程序运行时的调用栈

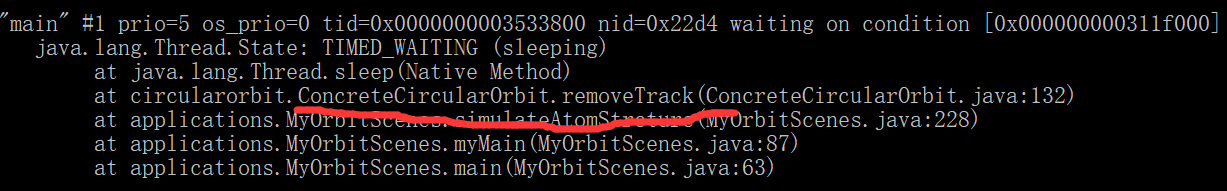
jstack用于生成java虚拟机当前时刻的线程快照。线程快照是当前java虚拟机内每一条线程正在执行的方法堆栈的集合，但是由于调用每个方法的时候速度过快，我们很难在调用和的同时打印出调用栈，因此考虑在调用该方法时，使用sleep()使程序暂停一段时间继续执行，方便我们导出程序运行时的调用栈。



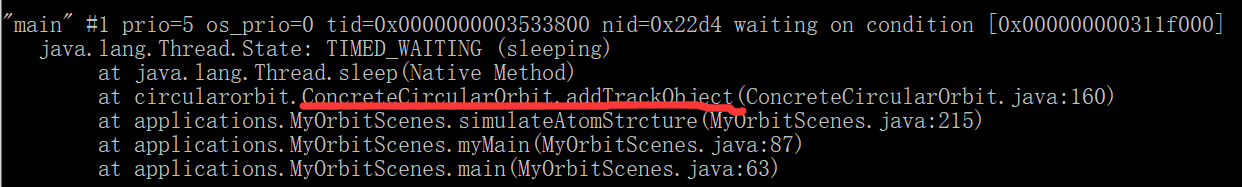
增加轨道：



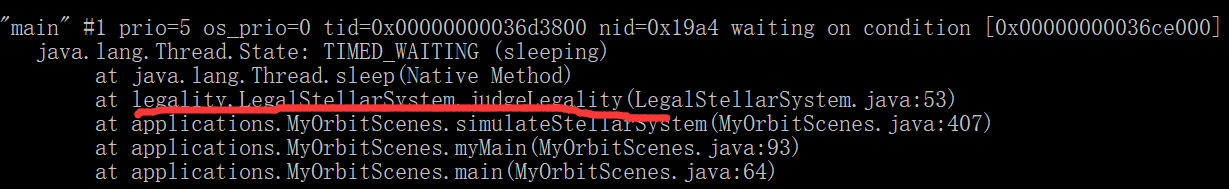
删除轨道：



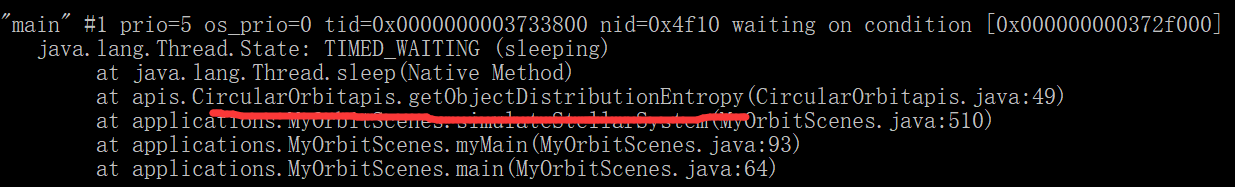
向特定轨道上增加物体：



判断轨道的合法性：



计算轨道系统的熵值：



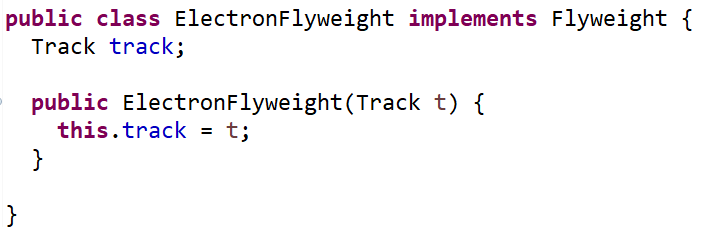
### 使用设计模式进行代码性能优化

**1. FlyWeight设计模式：**

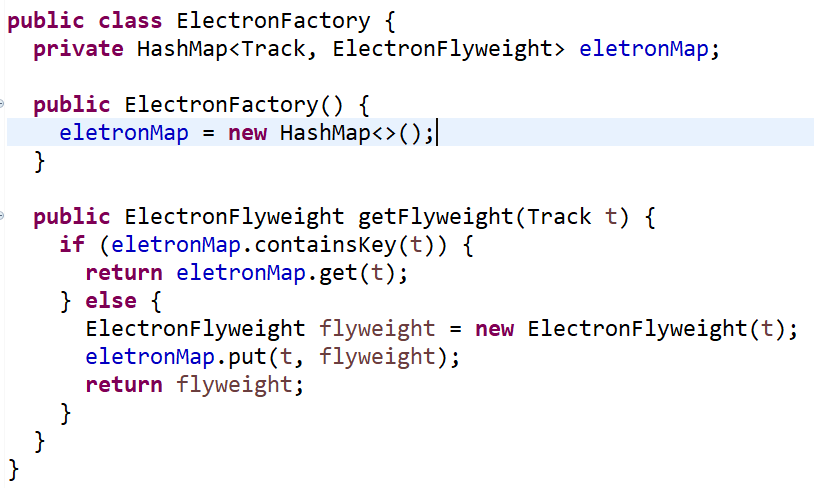
因为AtomicStructure中Electron的对象过多，并且其实它们除了所处的track，是等同的。因此我们使用FlyWeight设计模式设计Electron工厂，基本思想就是对于每一个轨道只有一个Electron实例。

但是有一处需要注意：因为以前的程序是依据引用（内存地址）判断相同的，在添加物体时的普遍规则是如果轨道系统中有相同的对象则报错，此处需要在子类中覆盖该方法，不进行对象相同检查。

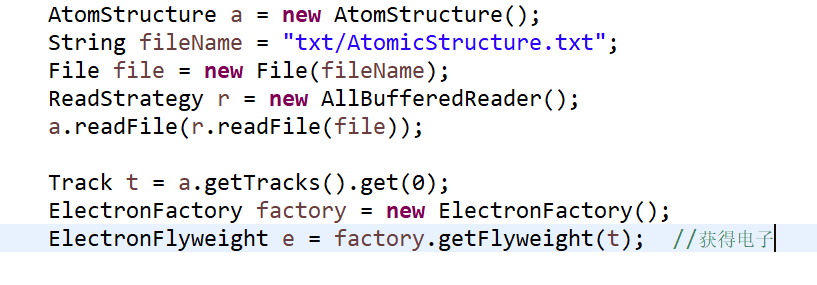
实现Flyweight接口的具体享元电子：



享元电子的工厂：



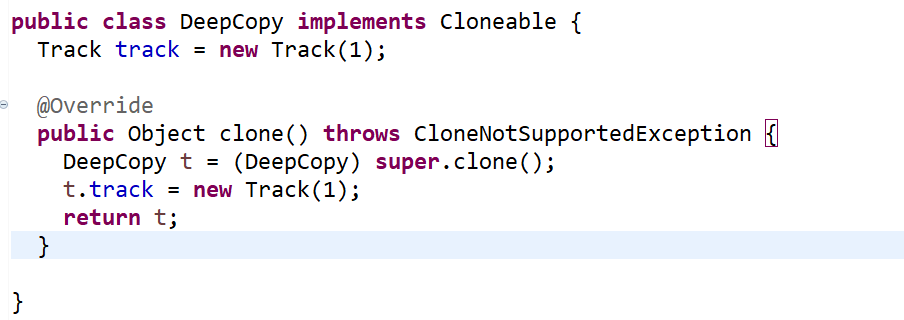
使用示例：



**2. Prototype设计模式：**

对于该轨道系统，事实上我认为最适合Prototype设计模式的是享元电子的部分，我们可以以一个享元电子作为原型，使用clone()方法来创建新的实例。

由于电子的Track是不同的，被复制的享元电子中引用了Track时，需要创建新的Track(而不是共享)。因此我们需要使用深拷贝。



**3. Singleton设计模式**

类似于lab2中构建一个图之后，getInstance返回一个图的实例。我们实验中描述轨道物体之间关系、描述中心物体和轨道物体之间关系的图也可以这样返回图的一个单例，通过封装确保对象的复用，不用让客户端考虑。

**private** **static** EeIntimacyMap *instance* = **new** EeIntimacyMap();

**public** **static** EeIntimacyMap getInstance() {

**return** *instance*;

}

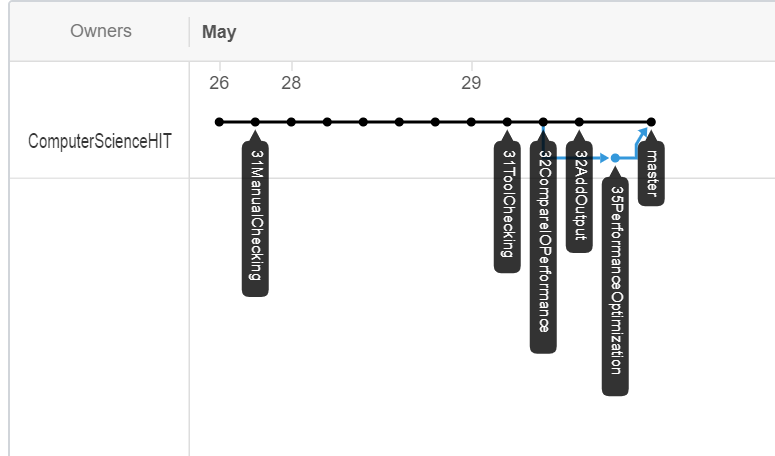
EeIntimacyMap s = EeIntimacyMap.getInstance();

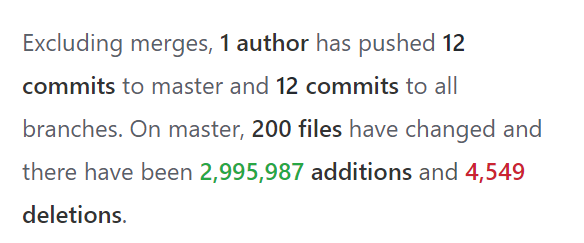
**4. 对于Object Pool 设计模式**，我的程序中并没有很合适的使用之处，但是我们可以产生一个不大的整形数的时候，使用Integer num = 1; 因为Integer类中，缓存了从-128到127之间的所有的整数对象，这样也是内存友好的一种方式。

**5. 其他**的性能改进：声明字符串变量的时候，使用 ”” 而不是用new String方法。

## Git仓库结构

请在完成全部实验要求之后，利用Git log指令或Git图形化客户端或GitHub上项目仓库的Insight页面，给出你的仓库到目前为止的Object Graph，尤其是区分清楚本实验中要求的多个分支和master分支所指向的位置。





# 实验进度记录

请使用表格方式记录你的进度情况，以超过半小时的连续编程时间为一行。

每次结束编程时，请向该表格中增加一行。不要事后胡乱填写。

不要嫌烦，该表格可帮助你汇总你在每个任务上付出的时间和精力，发现自己不擅长的任务，后续有意识的弥补。

|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 时间段 | 计划任务 | 实际完成情况 |
| 2019.5.20 | 14:00-18:00 | 阅读实验 | 按计划完成 |
| 2019.5.21 | 18:30-23:00 | 通读实验要求，配置环境 | 按计划完成 |
| 2019.5.22 | 15:30-23:00 | 手动修改代码 | 比预计晚2小时 |
| 2019.5.23 | 13:00-18:00 | 完成checkstyle | 比预计晚1小时 |
| 2019.5.23 | 19:00-23:00 | I/O读文件比较 | 未完成 |
| 2019.5.23 | 14:00-17:30 | 继续优化input | 按计划完成 |
| 2019.5.25 | 18:30-22:30 | 实现3种I/O策略 | 按计划完成 |
| 2019.5.26 | 14:00-17:00 | 分析垃圾回收，命令行工具使用 | 提前半小时完成 |
| 2019.5.26 | 17:30-23:00 | 设定JVM参数 | 按计划完成 |
| 2019.5.27 | 13:30-15:30 | 动态内存分析VisualVM | 未完成 |
| 2019.5.27 | 18:30-23:00 | 完成前一日内容，使用MAT | 按计划完成 |
| 2019.5.28 | 10:00-17:30 | 对MAT结果分析 | 比预计晚2小时 |
| 2019.5.28 | 18:30-23:00 | 使用设计模式优化代码 | 比预计晚1小时 |
| 2019.5.29 | 18:30-22:30 | 继续优化代码 | 按计划完成 |
| 2019.5.29 | 14:00-17:00 | 完成报告未写完的部分并提交 | 提前半小时完成 |

# 实验过程中遇到的困难与解决途径

|  |  |
| --- | --- |
| 遇到的难点 | 解决途径 |
| 手动代码走查时不清楚代码规范 | 翻阅Google 的代码规范 |
| checkstyle时需要修改的行数过多 | 修改preference即可实现大量代码，尤其是tab符的修改 |
| 读文件的速度可以统计，但由于建轨道过慢，导致写文件时间难以统计 | 持续优化搭建轨道的过程 |
| 使用-verbose gc时总出现Allocation failure | 在网上查资料确定时年轻代空间不足，修改参数 |
| 命令行工具使用出现部分error | 以管理员身份打开cmd |
| 对VisualVm和MAT的使用不熟练，用法不了解 | 在网上查阅相关工具的使用方式，不断学习 |
| 设计模式掌握不熟练 | 重新翻译上课使用的课件，结合着java设计模式的相关书籍，加深印象 |

# 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

## 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？

答：联系和冲突都有的。很多代码看起来不美，但是可能运行起来很高效，但是我们依然最好遵守代码规范，这样方便我们和他人的阅读。个人认为都很重要，“运行起来很美”更重要一些。若编程时间有限，倾向于放在后者。

1. 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？

答：会。很多隐藏的小错误可能会导致很大的麻烦。

1. 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？

答：能搞清楚。有很多I/O策略可能在我们少量代码的测试中体现不出优势，但是它在其他时候会有更高效的用途。

1. 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？

答：JVM的内存管理机制，和之前学的内存管理机制相比，特殊之处在于堆的管理。在C语言中，堆这部分空间是唯一一个程序员可以管理的内存区域。程序员可以通过malloc函数和free函数在堆上申请和释放空间。只不过和C语言中的不同，在Java中，程序员基本不用去关心空间释放的问题，Java的垃圾回收机制会自动进行处理。因此这部分空间也是Java垃圾收集器管理的主要区域。另外，堆是被所有线程共享的，在JVM中只有一个堆。这也是JVM内存管理机制的一个新意。我认为这很好，但是仍有需要改进的地方。

1. 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？

答：在C++中，堆这部分空间是唯一一个程序员可以管理的内存区域。程序员可以通过malloc函数和free函数在堆上申请和释放空间。只不过和C语言中的不同。

而在Java中，程序员基本不用去关心空间释放的问题，Java的垃圾回收机制会自动进行处理。因此这部分空间也是Java垃圾收集器管理的主要区域。另外，堆是被所有线程共享的，在JVM中只有一个堆。还有改进的空间。

1. 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？

答：有很大的回报。内存分配更加合理，内存溢出的概率变小了。

1. 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？

答：体会到了。尤其是在我看到gc日志里的异常情况无法下手时，这些工具给了我很大的帮助。

1. 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？

答：必须得有耐心，这是普通的码农到高级程序员必经之路。

1. 关于本实验的工作量、难度、deadline。

答：工作量看似不大，做起来也不轻松，难度不大，deadline较合适。

1. 到目前为止，你对《软件构造》课程的意见与建议。

答：这是我非常喜欢的一门课，课程教会了我很多的知识，带我走进了软件构造的新世界。但是由于课程进度快，且ppt都是英文，因此课程难度较大，整个课程的设计应该还有可以完善的地方。因为课程ppt内容过多，建议多考核重点部分，不过多在意细枝末节。这个时候课程已经结束了，希望这门课越来越好！！