

**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 | 沈子鸣 |
| 学号 | 1170301007 |
| 班号 | 1703010 |
| 电子邮件 | [2508754153@qq.com](mailto:2508754153@qq.com) |
| 手机号码 | 18800421860 |

**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 1](#_Toc3923019)

[3.1 Static Program Analysis 1](#_Toc3923020)

[3.1.1 人工代码走查（walkthrough） 1](#_Toc3923021)

[3.1.2 使用CheckStyle和SpotBugs进行静态代码分析 1](#_Toc3923022)

[3.2 Java I/O Optimization 1](#_Toc3923023)

[3.2.1 多种I/O实现方式 1](#_Toc3923024)

[3.2.2 多种I/O实现方式的效率对比分析 2](#_Toc3923025)

[3.3 Java Memory Management and Garbage Collection (GC) 3](#_Toc3923026)

[3.3.1 使用-verbose:gc参数 3](#_Toc3923027)

[3.3.2 用jstat命令行工具的-gc和-gcutil参数 3](#_Toc3923028)

[3.3.3 使用jmap -heap命令行工具 3](#_Toc3923029)

[3.3.4 使用jmap -clstats命令行工具 3](#_Toc3923030)

[3.3.5 使用jmap -permstat命令行工具 3](#_Toc3923031)

[3.3.6 使用JMC/JFR、jconsole或VisualVM工具 3](#_Toc3923032)

[3.3.7 分析垃圾回收过程 3](#_Toc3923033)

[3.3.8 配置JVM参数并发现优化的参数配置 3](#_Toc3923034)

[3.4 Dynamic Program Profiling 3](#_Toc3923035)

[3.4.1 使用JMC或VisualVM进行CPU Profiling 3](#_Toc3923036)

[3.4.2 使用VisualVM进行Memory profiling 3](#_Toc3923037)

[3.5 Memory Dump Analysis and Performance Optimization 3](#_Toc3923038)

[3.5.1 内存导出 3](#_Toc3923039)

[3.5.2 使用MAT分析内存导出文件 3](#_Toc3923040)

[3.5.3 发现热点/瓶颈并改进、改进前后的性能对比分析 3](#_Toc3923041)

[3.5.4 在MAT内使用OQL查询内存导出 4](#_Toc3923042)

[3.5.5 观察jstack/jcmd导出程序运行时的调用栈 4](#_Toc3923043)

[3.5.6 使用设计模式进行代码性能优化 4](#_Toc3923044)

[4 实验进度记录 4](#_Toc3923045)

[5 实验过程中遇到的困难与解决途径 4](#_Toc3923046)

[6 实验过程中收获的经验、教训、感想 5](#_Toc3923047)

[6.1 实验过程中收获的经验和教训 5](#_Toc3923048)

[6.2 针对以下方面的感受 5](#_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），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

本次实验环境配置与之前相同。

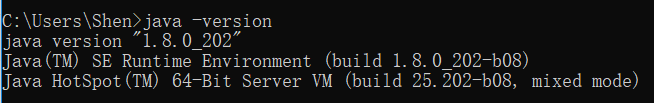
硬件环境：Intel Core i5-7200U，64位; 4G;

软件环境：Windows 10家庭中文版

IDE：Eclipse Java 2018-12

关于Java：JDK 8, JRE 8

关于Git：Git 2.20.1





GitHub仓库的URL地址：

<https://github.com/ComputerScienceHIT/Lab5-1170301007>

# 实验过程

## Static Program Analysis

### 人工代码走查（walkthrough）

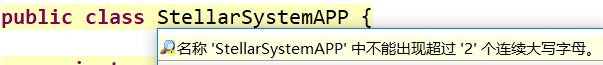
人工代码走查可以发现一些明显的格式错误。比如说包名规范问题，首字母建议小写，类名首字母建议大写，但其实这些问题都是在写代码是注意到的，人工代码走查也不会发现什么问题。

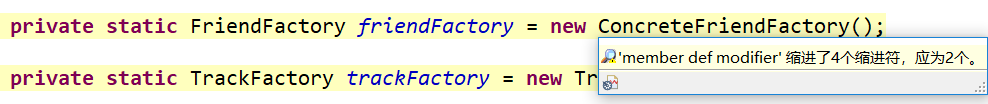
### 使用CheckStyle和SpotBugs进行静态代码分析

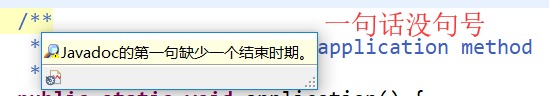
关于checkstyle的代码风格审查，我采用Google规范做代码审查。其它风格的审查也试过了，像阿里的代码风格审查，对我的要求比较不友好，最不友好的就是所有的常数都报出了non magic number的警告，还有system.out似乎也是不应该出现的，看起来阿里的发布软件规范中，不应该存在标准输出吧。

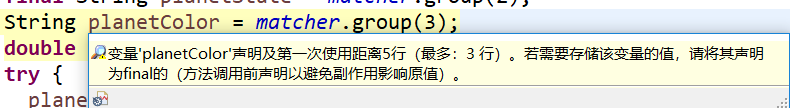
关于代码审查的修改，采用Eclipse中自带的formatter功能，在网上下载相关规范的formatter xml配置文件，然后在Eclipse中的formatter中import该xml文件，用自动格式化代码的功能，可以修改一些格式上的错误，比如缩进，注释等。但相关的无法自动修正的错误，如命名不规范，还是要人工解决的。

#### Checkstyle

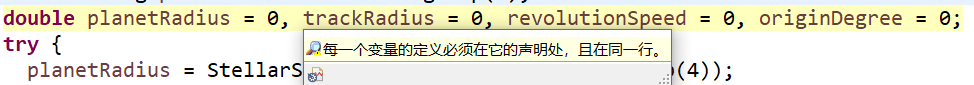


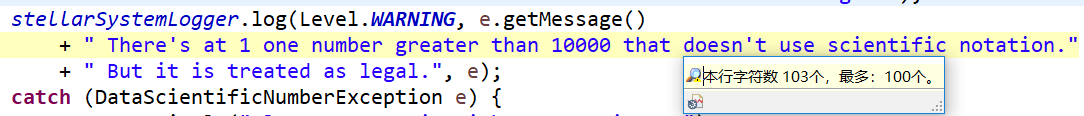


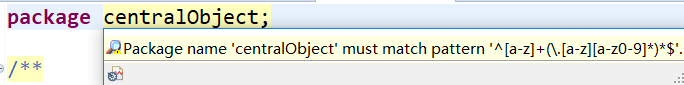




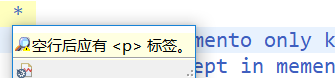
上图所示的不规范，如果这个变量没有后面修改的必要，就声明为final的就好。不然的话，这就被视作是一个局部变量，而且要被经常修改，那么它距离第一次使用的地方越近越好，这样可读性更好。



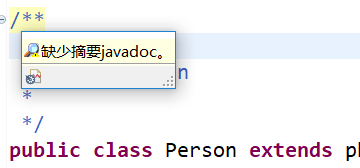


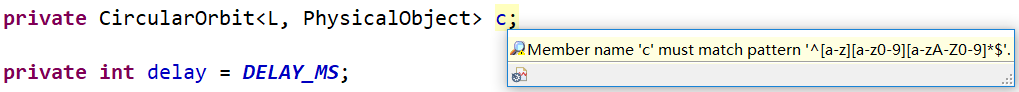


上图所示的错误解决起来也比较麻烦，倒也不是麻烦，是不符合之前的习惯而已，Google规范规定，包名必须全部用小写字母，可以用句点分割，但用句点分割语义单词好像不是那么常见。



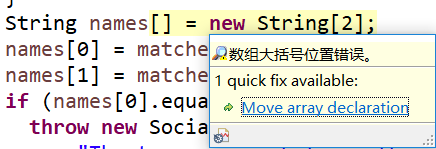
上图所示的更棘手。当我的javadoc有两段注释内容时，空行后应该有<p> html标签，但是加了标签，再用formatter就会把这个标签单独放到一行中。为了解决这个问题，需要修改formatter的权限，把javadoc中的html标签自动格式化禁用掉。



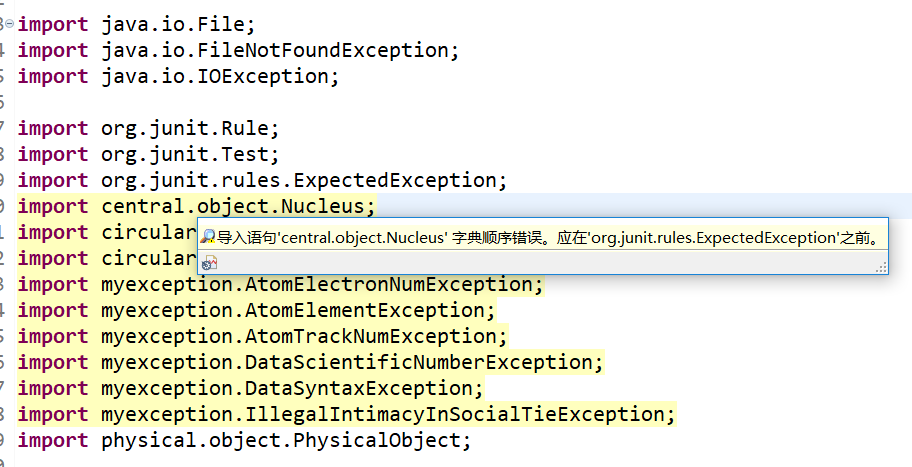




上图是一个抽象类，抽象类要求首字母规范大写。



这个规范是头一次见，其实之前在定义数组时，大括号位置是放在类型后，还是变量后，也是模糊不清，这次记住了，放在类型后。



这是什么奇葩规范(⊙ˍ⊙)？好多地方报出了这种“import顺序”的不规范，其实import顺序倒也没什么影响，可能对开发人员来说，过多的import还是需要按规范顺序写才行。

这个import不规范真的是挺烦人的。写代码的时候用到什么了，就直接只能import，但可能执行的顺序不是严格的逻辑顺序，就是按调用顺序import的，后来报出这么多不规范，就得都挨个改。

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。

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。

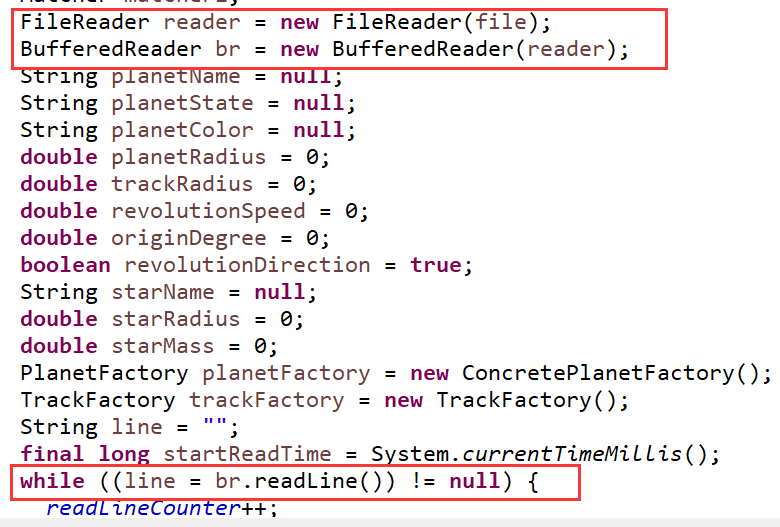
## Java I/O Optimization

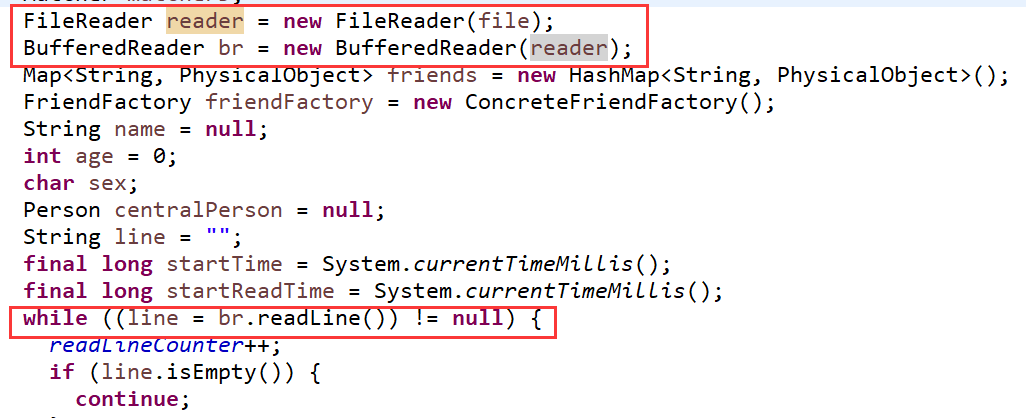
### 多种I/O实现方式

#### FileReader/FileWriter

java.io.FileReader/FileWriter是用来读写文本文件的，因为文本文件本质是二进制数据流，这个类先将二进制流转成字符流，然后读取进来。

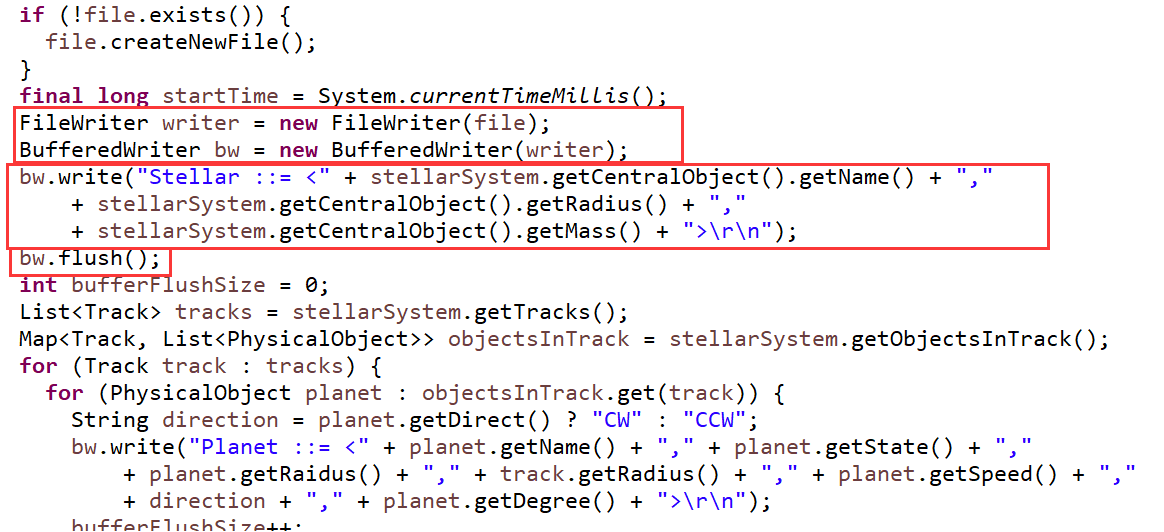
为了实现按行读取以及读取效率的提高，用java.io.BufferedReader/BufferedWriter来辅助读写。读文件方式如下：

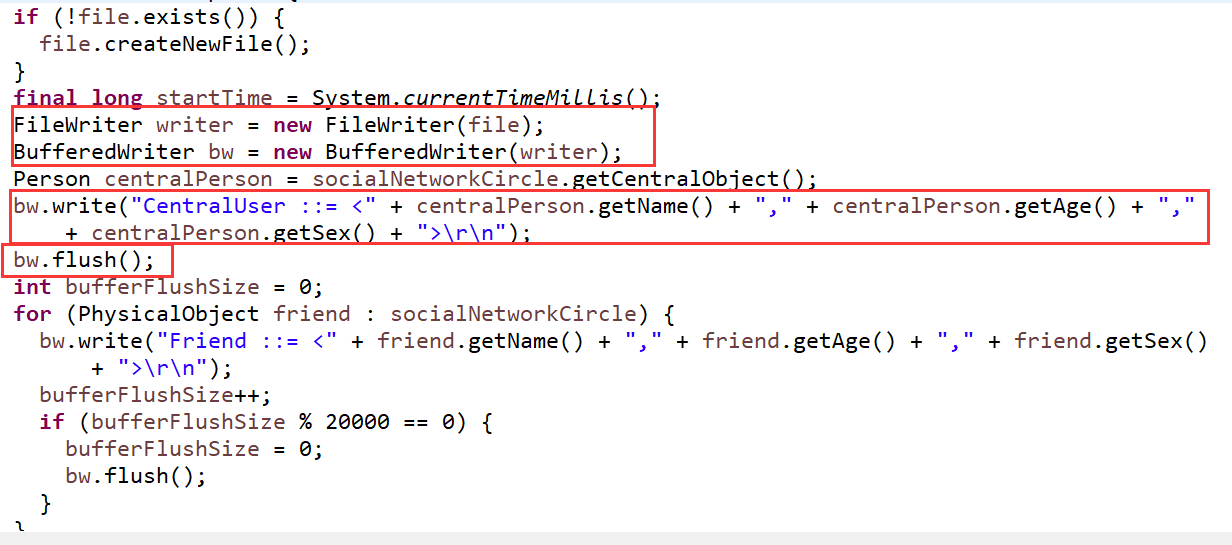




变量line即为读取文本的每一行文本信息。

写文件方式如下：





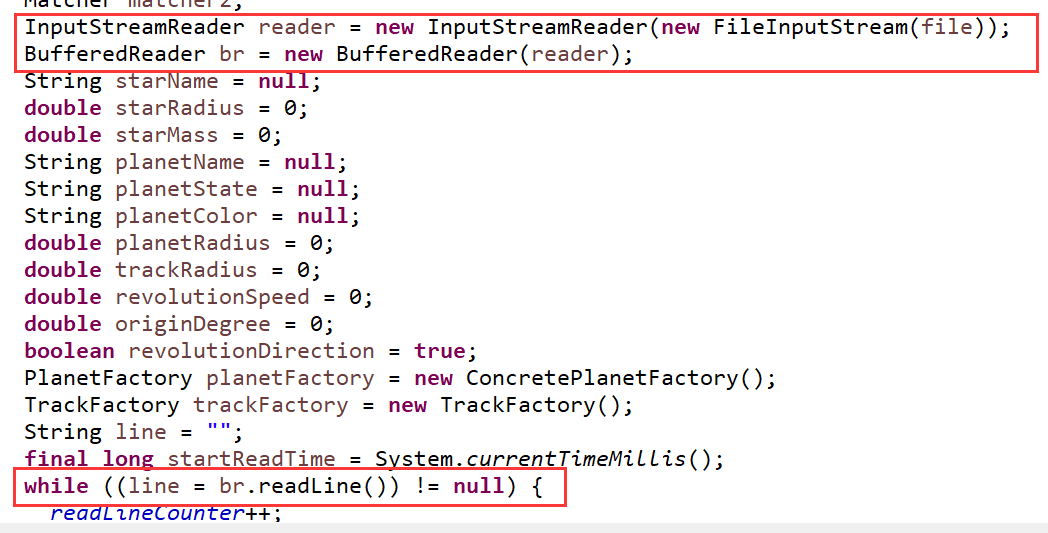
用BufferedWriter对象执行写操作，但这只是写入到缓存区，真正写入到文件中的语句是flush()方法。

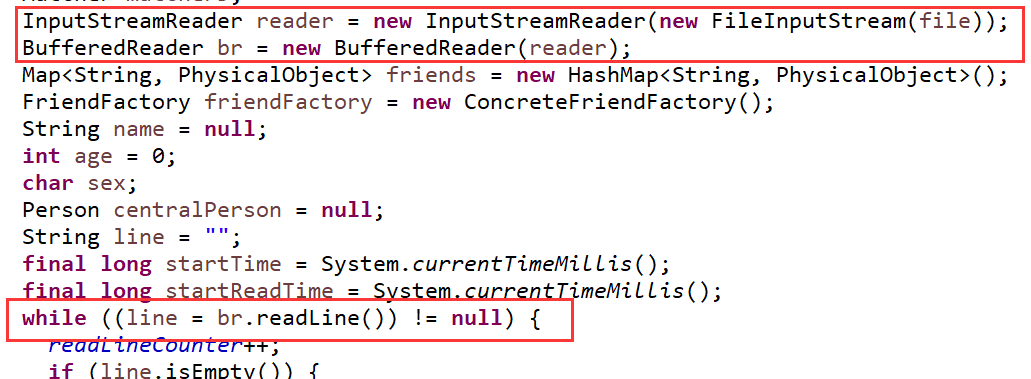
另外不要忘了在每一个可能导致方法返回的语句之前加上方法close(),防止BufferedReader/BufferedWriter资源泄露。

#### FileInputStream/FileOutputStream

java.io.FileInputStream/FileOutputStream和FileReader/FileWriter差不多，但它是用来读写二进制数据文件的，与FileReader/FileWriter不同的是，它是直接读写二进制字节，而不是先转成字符再读写。

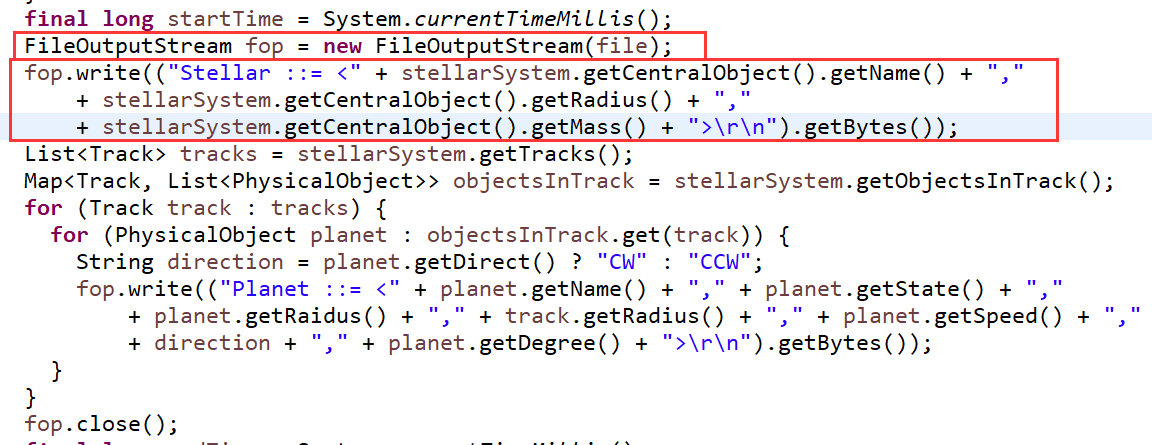
为了实现按行读写以及效率提高，辅以BufferedReader来读取文件。读取方式如下：

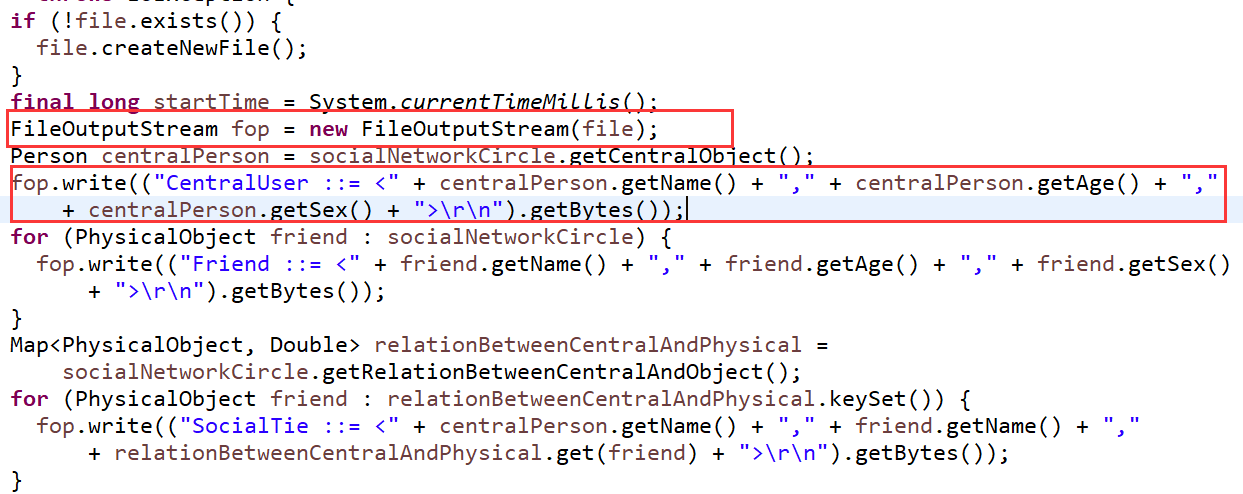




变量line即为读取文本的每一行文本信息。

写文件如下：





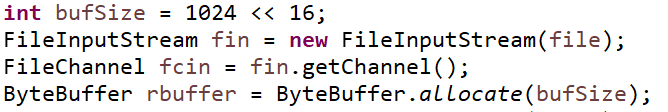
其实如果使用Buffered Stream相关的类的话，算上一个缓冲区用来写文件，会更加高效，注意这样的话需要调用flush方法写文件。FileOutputStream中也有flush方法，但其实没起什么作用。

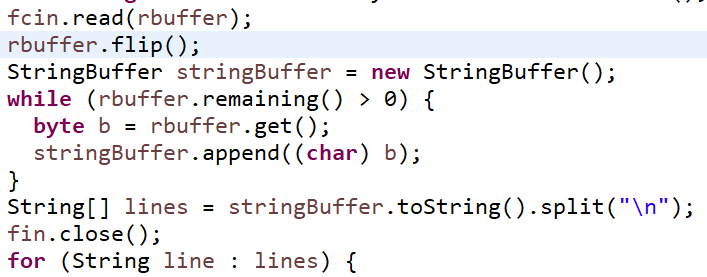
另外不要忘了在每一个可能导致方法返回的语句之前加上方法close(),防止BufferedReader资源泄露。

#### NIO

既然要求使用至少三种IO方式，第三种就学习一下nio相关读写吧。之前从来没有接触nio，在博客上学习一下。

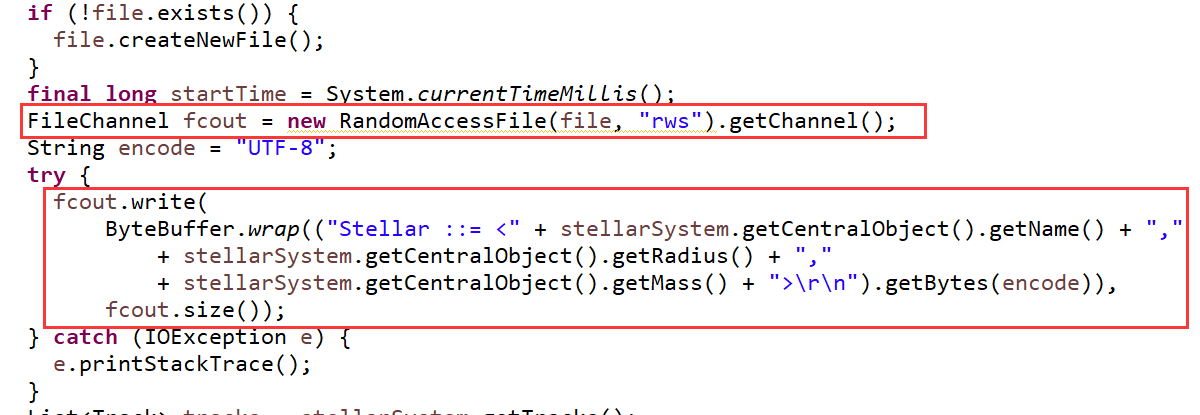
读文件具体实现方法如下：





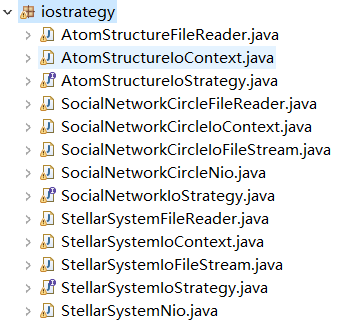
变量stringBuffer是读取到的所有文本内容。将其用split方法解析，得到的数组lines中的变量line是一行完整的字串信息。接下来用之前用到过的代码段解析字串即可。

写文件的具体代码如下：



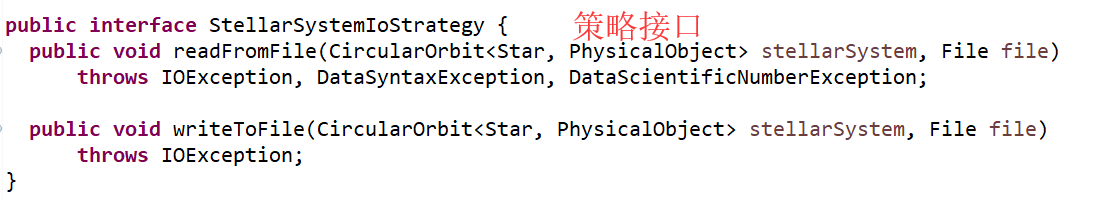
#### Strategy设计模式

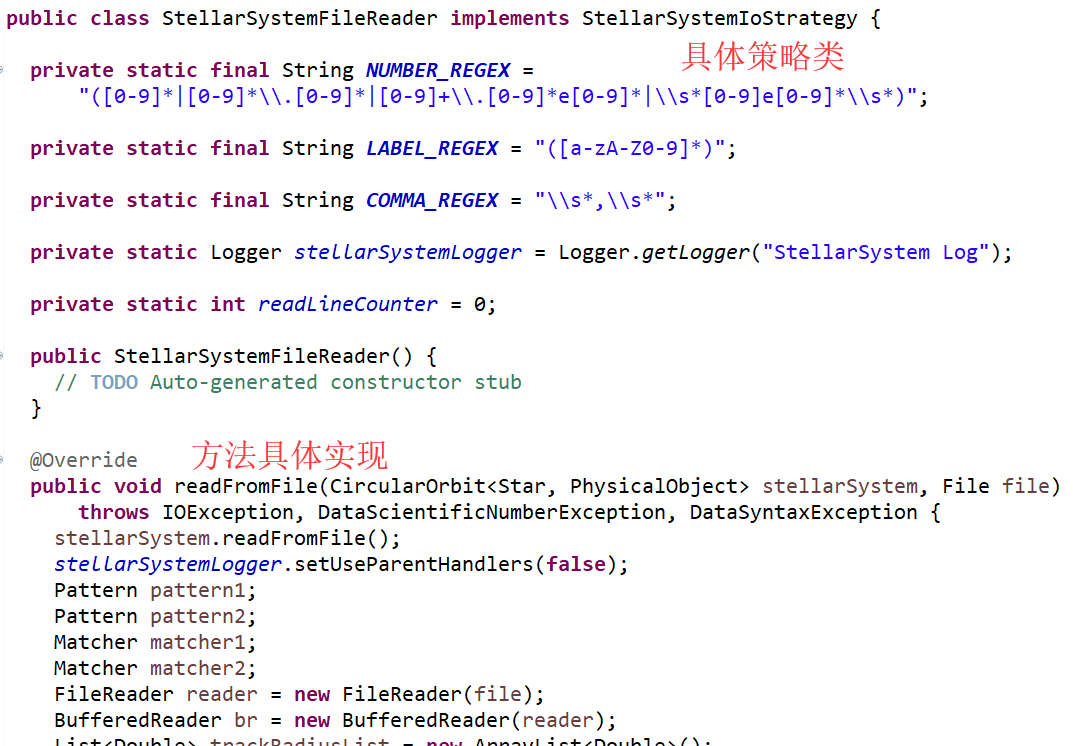
虽然本实验中没有对原子结构模型的大数据文件读取要求，但也加入到strategy设计模式中了，类结构如下：



Strategy设计模式可以自由的在多种代码策略中切换。主要组成部分有，策略接口，具体策略类，策略上下文类。实现方法为，策略接口中定义方法，具体策略类引用策略接口，具体实现策略接口中定义的方法，具体实现取决于具体策略，具体策略类通常有多个。最后再实现一个策略上下文类，该类中有一个策略接口类型的具体策略类对象表示，并在策略上下文类构造器中传入参数，构造指派关系，在策略上下文类中再定义需要调换策略的各个方法，方法中调用指派表示的方法，表示执行策略。

具体代码如下：







### 多种I/O实现方式的效率对比分析

#### 时间计算

用经典的计算代码执行时间的套路，用System.currentTimeMillis()获取当前时间，单位为ms，变量类型为long，在读文件前后调用两次，取差值即可。

#### 表格方式对比不同I/O的性能

注：每个数据都做了若干次实验，取其中最相近的三次结果的平均值记为最后结果。

表一是行星轨道系统的读取以及构造的所耗时间。

表二是社交关系网络的读取所耗时间。

表三是社交关系网络的读取及构造和写入所耗时间。

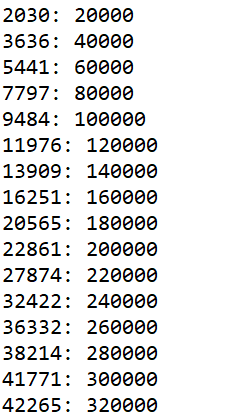
|  |  |  |  |
| --- | --- | --- | --- |
| StellarSystem | FileReader(BufferedReader) | FileInputStream(BufferedReader) | NIO |
| 1 | 13.097s | 12.296s | 13.147s |
| 2 | 13.125s | 11.837s | 12.838s |
| 3 | 13.562s | 12.601s | 12.594s |
| Average | 13.261s | 12.245s | 12.860s |

|  |  |  |  |
| --- | --- | --- | --- |
| SocialNetworkCircle | FileReader(BufferedReader) | FileInputStream(BufferedReader) | NIO |
| 1 | 10.893s | 10.3s | 9.454s |
| 2 | 10.991s | 9.696s | 9.569s |
| 3 | 10.938s | 9.876s | 9.509s |
| Average | 10.941s | 9.957s | 9.511s |

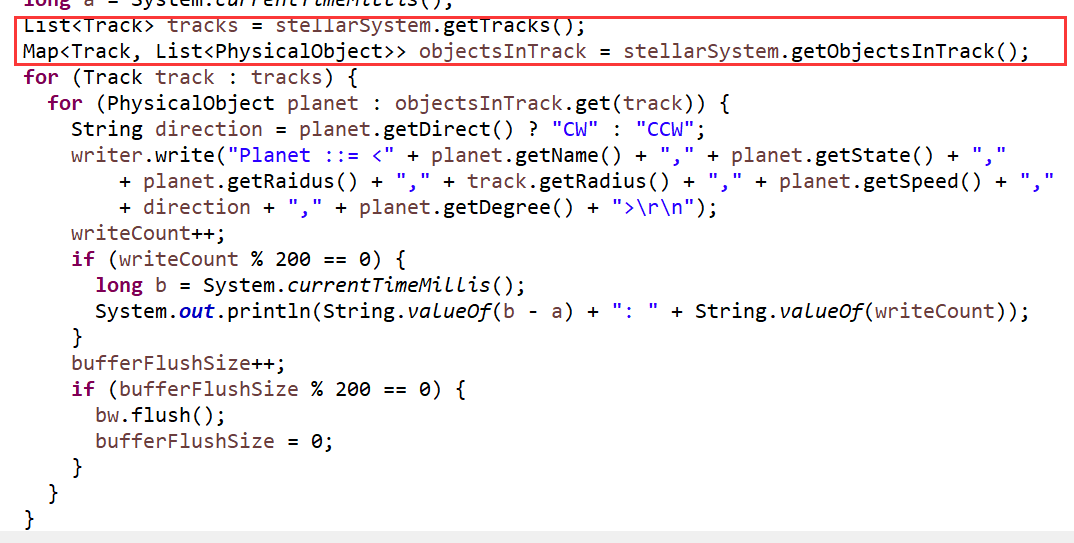
|  |  |  |  |
| --- | --- | --- | --- |
|  |  | StellarSystem.txt | SocialNetworkCircle.txt |
| FileReader/FileWriter  (BufferedReader/BufferedWritter) | 读文件 | 4.979s | 12.444s |
| 写文件 | 5.607s | 0.306s |
| FileInputStream/FileOutputStream | 读文件 | 4.328s | 11.04s |
| 写文件 | 8.047s | 5.504s |
| NIO | 读文件 | 4.609s | 10.602s |
| 写文件 | 7.789s | 5.396s |

#### 相关优化

关于两个轨道系统的构造时间，社交关系网络的构建由于数据文件的特点，而且要用最短路径算法求得每个物体到中心的逻辑最短距离，必须要先读取所有数据后，再做解析构建，但行星轨道系统不同，每一行数据就可以提供一条构建信息，所以轨道的构建是可以在读文件时进行的。但是，经过测试发现，如果将两者分开，用几个中间变量来存储读取到的数据，当读文件结束后再根据中间变量构建，那么总耗时比一起处理要更慢，结果是慢三到四倍。 事实上，正常来说，注释掉大部分耗时的异常检测，仍然需要8到10分钟的性能瓶颈在于，每一次向轨道上添加物体，都要做一次按角度排序，这是为了在之后的遍历中，同一轨道上上的物体可以按角度升序遍历，因为这个功能用的地方不多，故此处将该功能取消，否则，这个时间复杂度就会达到O(n2)，非常耗时。可以看出，每两万行数据处理的速度是线性递减的。



在尝试得到写文件的时间时，发现性能极其低下，分析原因发现在获取轨道系统的相关信息时，将observer方法写入到了for循环中，也就是每依次循环都要调取这observer方法，而observer方法中做了防御式拷贝，这样的话在大数据时，几十万次循环，每次循环都做大量拷贝，性能急剧降低，为此，将observer方法提取到外面如下：

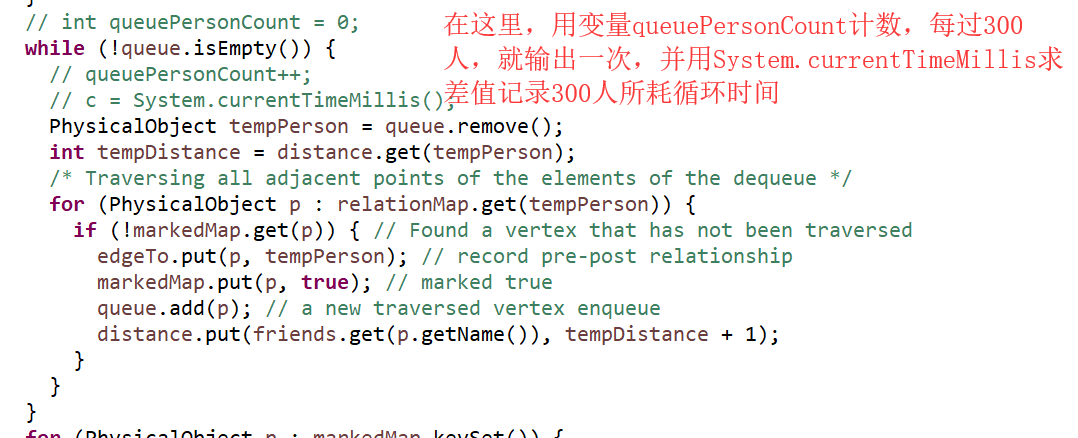


另外，在读取社交网络关系数据和构建ADT时，速度极慢，性能十分低下，第一构建时，总用时长达两个多小时。经过一整晚的不懈努力，终于把时间控制在了一个比较理想的范围内。最后的读取文件以及构建ADT总用时大约不到10秒，而且以我的数据结构，如果考虑某些对象冲突异常，那么处理的时间复杂度将是O(n2)，这对性能影响非常大，所以，尽可能的将所有相关数据结构改为HashMap或HashSet，利用containKey方法来判断冲突，不能用containValue方法，这个性能也非常低。事实上，经过一次比较长的运行，证明所给数据是不存在异常的。

首先，第一次运行了两个多小时，是因为中间变量太大。因为我在之前的实验中，考虑社交网络关系中给定关系数据中没有人的相关信息，而Friend和SocialTie并没有严格规定谁先给出，所以要构建关系的话，可能需要先读取所有人的信息保存下来，再构造关系。而我连SocialTie的信息也保存在字串数组中了，这样的话就会出现长度为一百万的中间变量，再次解析性能极低。为此，取消中间变量，以姓名作为人的唯一标识，在构建关系时不给定人的信息，而只是用人的名字加上虚构的信息构造对象，建立关系。这样，一次读取文件，就可以构造所有人和关系。这样一来，读数据的时间代价是可观的，但是构造起来却还是要半个小时。



不满足半个小时的性能，继续优化。这次优化主要对BFS算法中的数据结构优化。分析性能瓶颈，由于我们需要知道所有人所在的轨道位置，才能将人和轨道匹配，用BFS的话，每次出队，就是一个人，理应有10万次出队（10万个人）。在循环中对出队速度监视：



得到的结果是每300人大概需要5到6秒，这样的话10万人用半个小时也就正常了，说明时间消耗主要在这个出队循环上。对while循环中的语句再做记时分析，发现，其中的for循环（与上图不同，当时做的时候，调用了CircularOrbitApis.targets方法）每做一次循环耗时10-20毫秒，该方法是在关系结构List<Edge<PhysicalObject>>中，搜索Edge对象，找到与tempPerson有关的Edge对象，这个耗时是一次的，但是，每个人都要搜索一次，这样时间就是二次的，虽然一次用时十几毫秒，但10万人用半个小时也就不足为奇了。为此，决定用更省时的数据结构，将targets所表述的信息在循环外先提取出来，在循环内部直接调用即可。利用HashMap的特殊属性，构造一个数据结构Map<PhysicalObejct, HashSet<PhysicalObject>>，意为构造一个映射表，将每个人映射到与他有关系的朋友集合上。这样使用HashMap的get方法，可以很快得到关系集合，也就是targets给出的信息。这段改进代码和上图所示相同。这回，构造图的时间优化到了不到3秒！

#### 图形对比不同I/O的性能

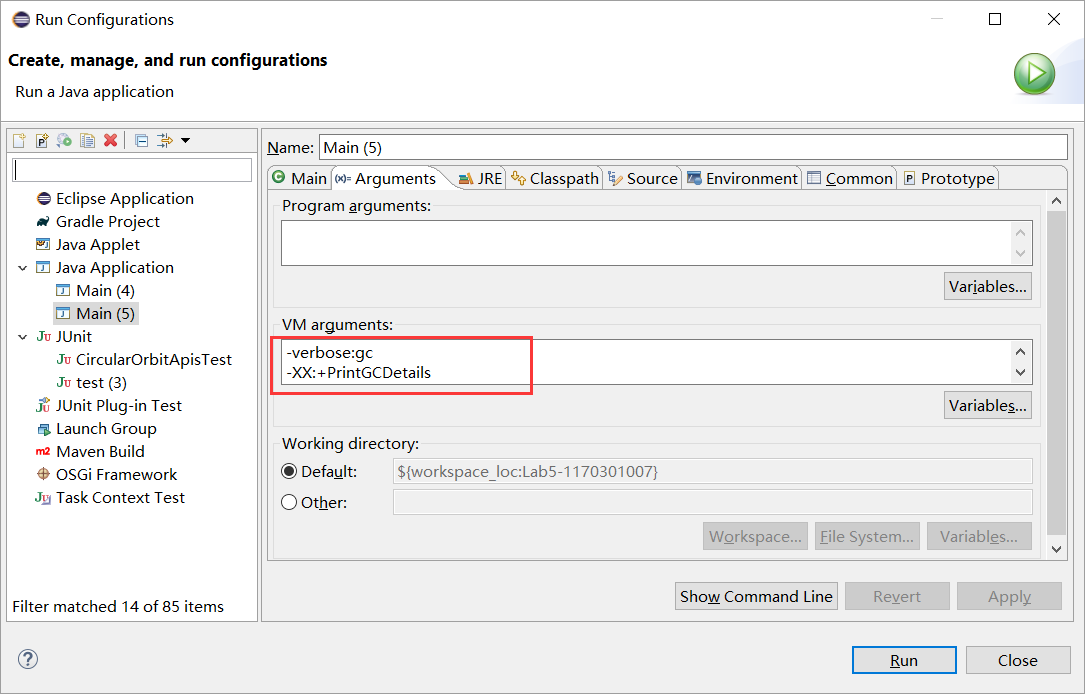
读文件

写文件

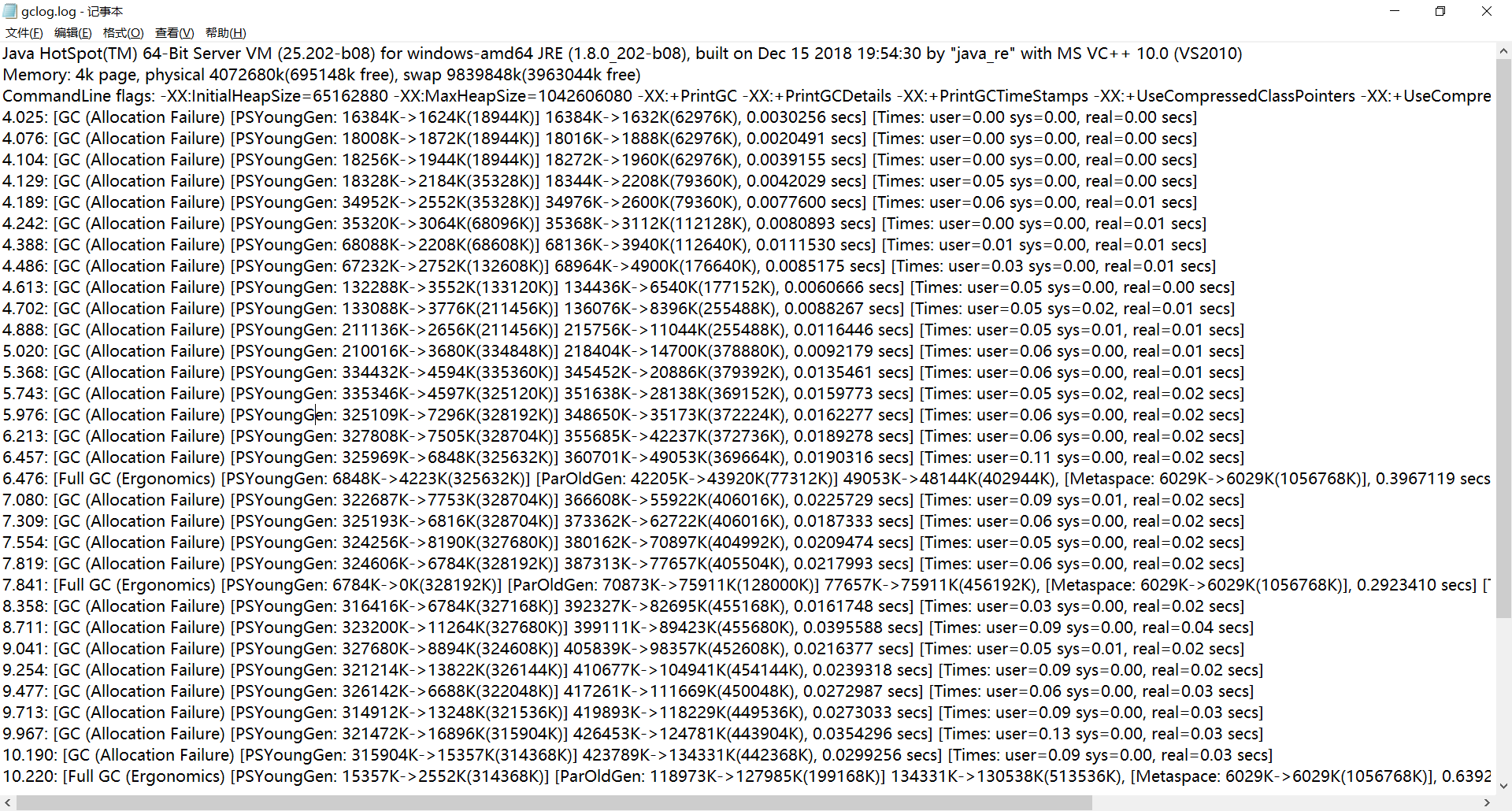
## Java Memory Management and Garbage Collection (GC)

### 使用-verbose:gc参数

使用方法：在代码编辑区右键→Run As→Run Configurations→在Java Application中，找到Arguments选项卡，在VM arguments编辑区中，以行为单位输入配置命令。



在一次读取社交关系网络的运行后，查看GC日志如下：



在该日志中，GC标识缺省表示Minor GC，否则为Full GC，标识后面括号中表明了本次GC发生的原因，如Allocation Failure，主要是因为向young generation(eden)给新对象申请空间，但是young generation(eden)剩余的合适空间不够所需的大小导致的minor gc。显然，我们的程序运行中，绝大时刻发生的都是Minor GC，偶尔会发生一次Full GC，这是由于Ergonomics导致的，Ergonomics是JVM的一种自适应机制，因为开启了UseAdaptiveSizePolicy，JVM自己进行自适应调整引发的Full GC。

两种GC单次花费的时间可以从最后一列的real分量数据得知。从程序启动开始，第一次发生的Minor GC在4.025秒时，在第一次发生Full GC之前，Minor GC单次消耗不足0.01秒，0.01秒和0.02秒不等。第一次发生Full GC使用了400毫秒左右，可以看出Full GC单次时间远多于Minor GC时间。第一次发生Full GC之后，Minor GC的单次时间仍在递增，最长的一次有150毫秒。而后面发生的两次Full GC单次时间分别为640毫秒和1380毫秒。该分量给出了准确时间：

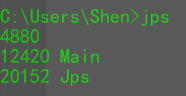




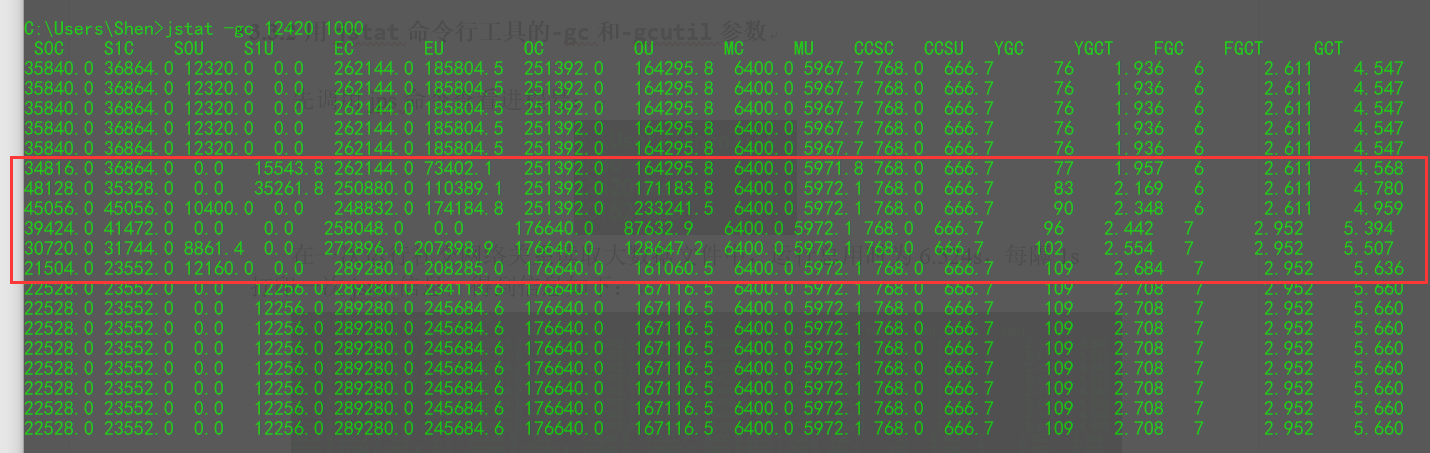
另外，在一次Minor GC中，红框中表示Young Generation中的内存变化，蓝框中表示堆中的内存变化，红框变化值减去蓝框变化值，即为本次GC被移动到Old Generation的内存值。

### 用jstat命令行工具的-gc和-gcutil参数

先调用jps命令查看进程号：



在一次运行社交网络关系读取大数据文件中，运行总用时为6.524s，每隔1s打印一次jstat信息，得到信息如下：



第一行各项解释如下：

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

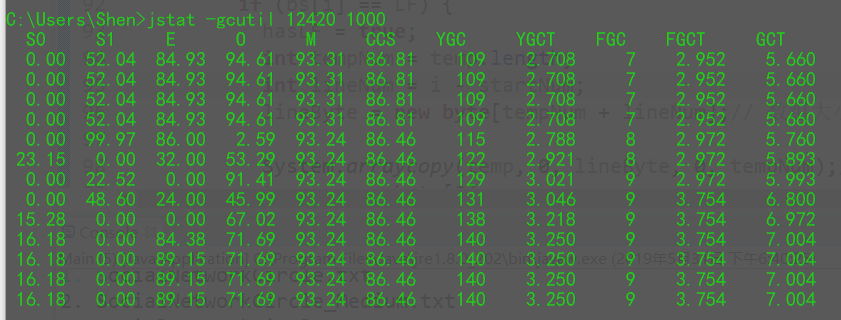
YGCT：年轻代垃圾回收消耗时间

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

-gcutil选项可以查看各个内存区所使用的百分比



其中第一行各项解释如下：

S0：幸存1区当前使用比例

S1：幸存2区当前使用比例

E：伊甸园区使用比例

O：老年代使用比例

M：元数据区使用比例

CCS：压缩使用比例

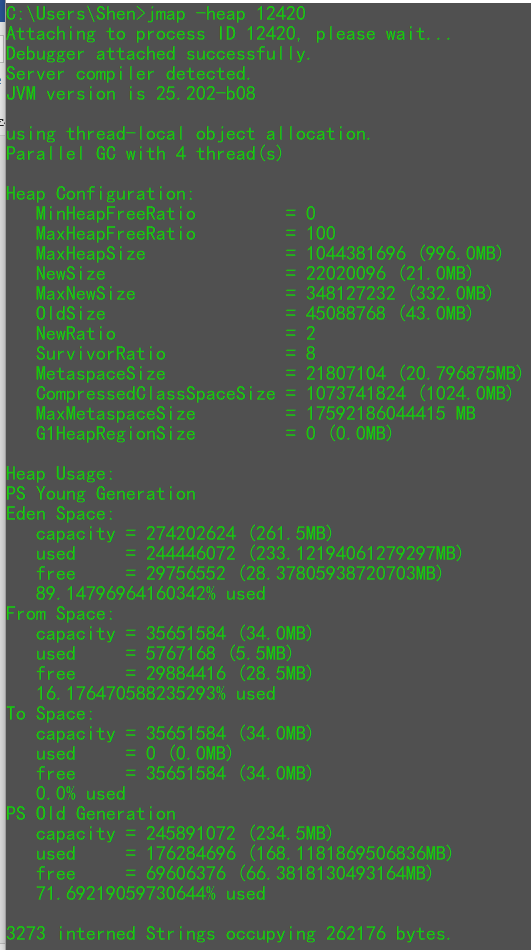
YGC：年轻代垃圾回收次数

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

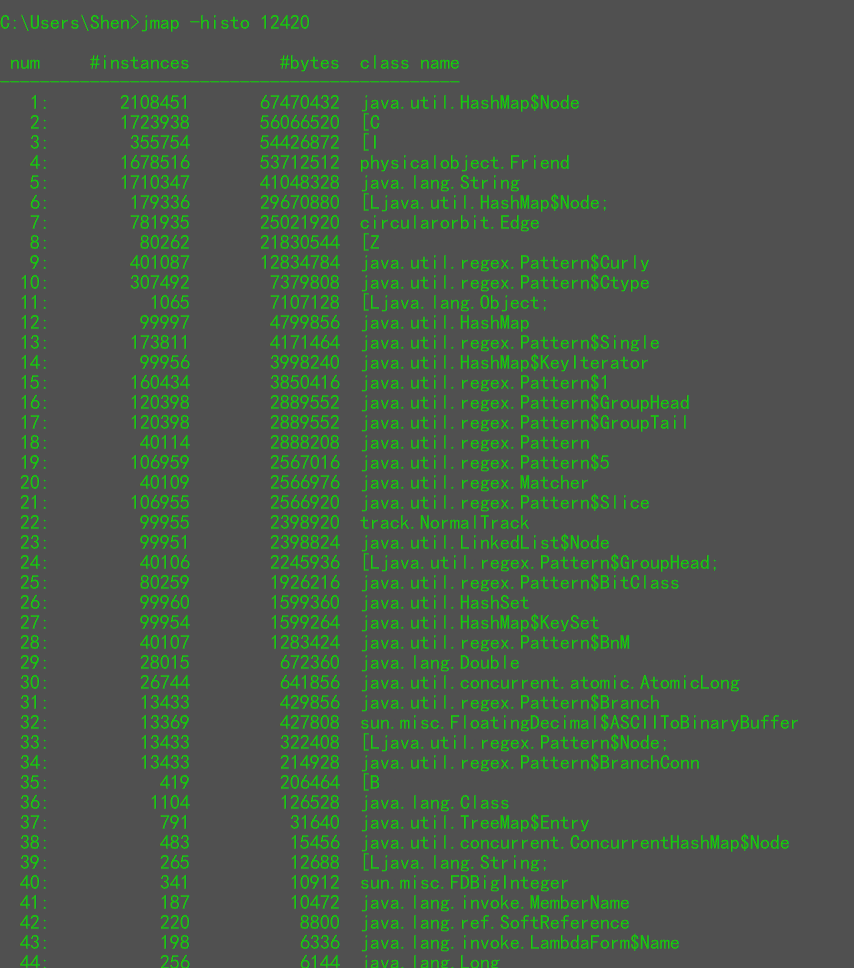
### 使用jmap -heap命令行工具



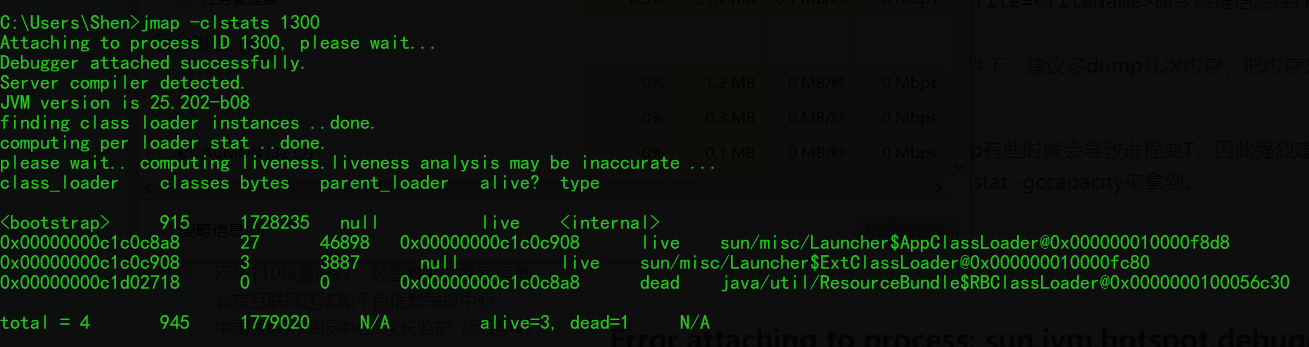
上图中，依次的信息块为Heap Configuration（堆内存初始化配置），下面的各参数都是和堆相关的参数配置。Heap Usage下面的各个信息块是Eden space（伊甸区的使用情况），From Space（其中一个survivor区的使用情况），To Space（另外一个survivor区的使用情况），PS Old Generation(Old区使用情况)。

### 使用jmap -clstats命令行工具

jmap -histo 12420(PID)部分所示如下，第一行的各项分别是对象序号、某个对象示例数、当前对象所占内存大小、当前对象的全额定名。



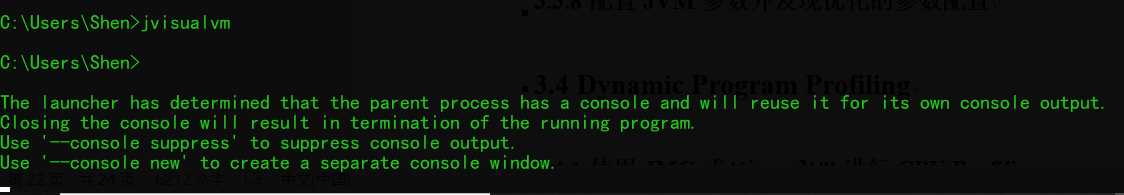
jmap -clstats 1300(PID)所示如下：



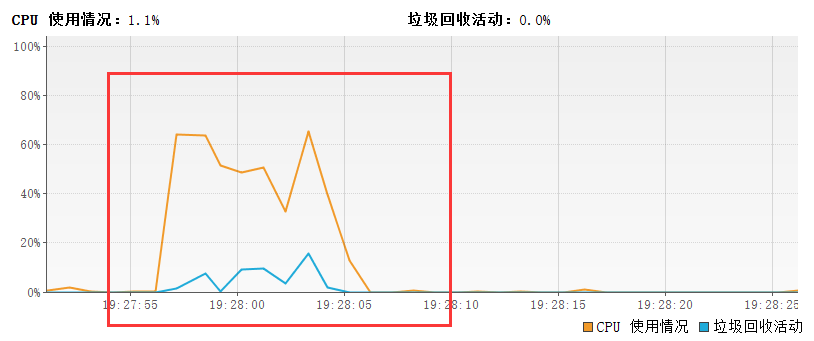
### 使用jmap -permstat命令行工具

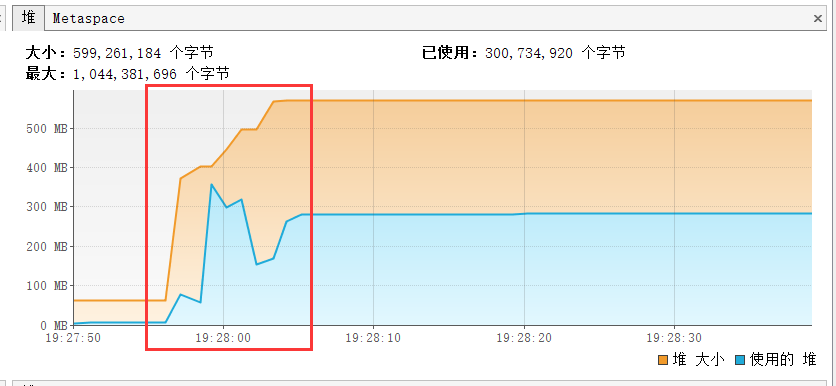
### 使用JMC/JFR、jconsole或VisualVM工具

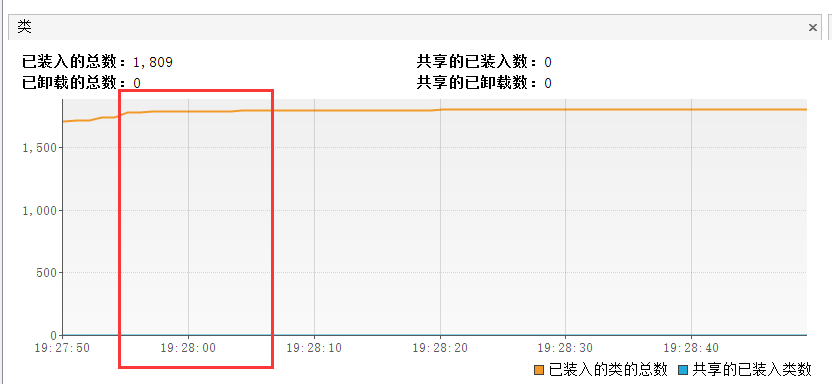
使用VisualVM，在命令行输入命令jvisualvm运行之：

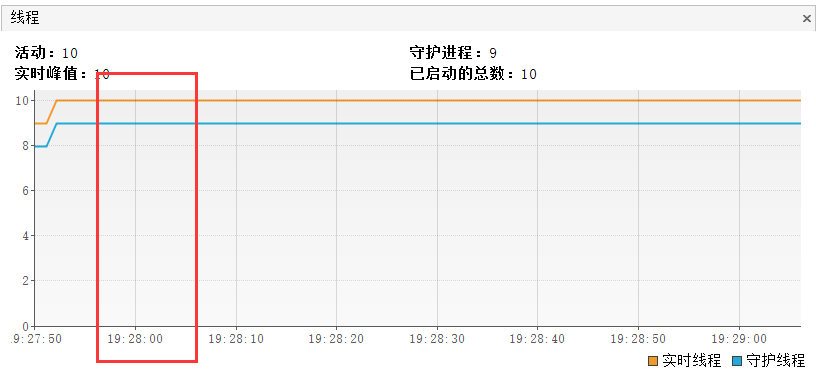


运行读取社交关系网络的大数据文件，监视结果如下：









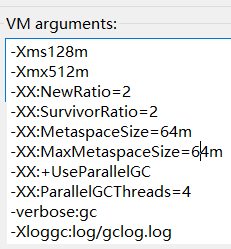
读取文件并构建对象的用时为8.518s。可以看出，在该段时间内，CPU使用峰值达到了约50%，垃圾回收的峰值达到了约20%。堆的使用情况如图二，类和线程的使用没有涉及，如图三和图四。

### 分析垃圾回收过程

CPU利用率和垃圾回收的活动强度，大体上是同步的，当垃圾回收活动发生时，CPU利用率会变高。从堆使用大小和总大小曲线图可以看出，每当堆使用达到一定值时，就会触发一次垃圾回收，堆使用大小减小，然后再次使用，重复该过程。另外，堆的总大小始终是不断增大，而结合垃圾回收过程的的曲线图，可以看出垃圾回收和堆大小的关系是同步的。

### 配置JVM参数并发现优化的参数配置

参数配置如下：



经过实验测试，以读取社交网络并构建rep为例，如果配置-Xms=64m，-Xmx=128m，是需要很长时间的，如果配置-Xms128m，-Xmx256m，读取文件稍慢，而构建过程中会报出内存不足的错误。当保持-Xms128m，增大-Xmx512m，使得内存足够使用，但性能相比-Xms256m没有太大变化。

而对于参数-XX:NewRatio来说，配置参数为2到6之间对读取并构造文件的运行时间都没有太大影响。-XX:SurvivorRatio在2到6之间对运行时间也没有太大影响。-XX:MetaspaceSize和-XX:MaxMetaspaceSize置为64m或128m或256m对运行时间也影响不大。

## Dynamic Program Profiling

### 使用JMC或VisualVM进行CPU Profiling



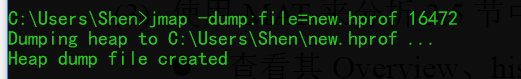
### 使用VisualVM进行Memory profiling



## Memory Dump Analysis and Performance Optimization

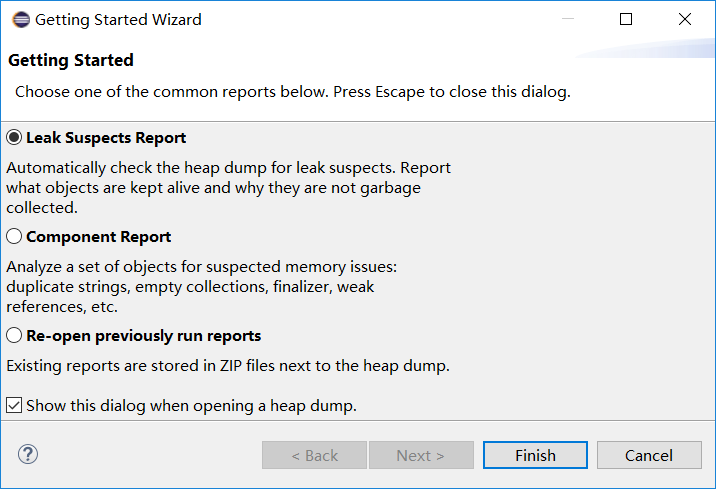
### 内存导出

内存导出用jmap -dump命令就可以，但我们需要用Eclipse的MAT工具来分析，所以这里最好导出.dprof文件，而不是.dump文件。

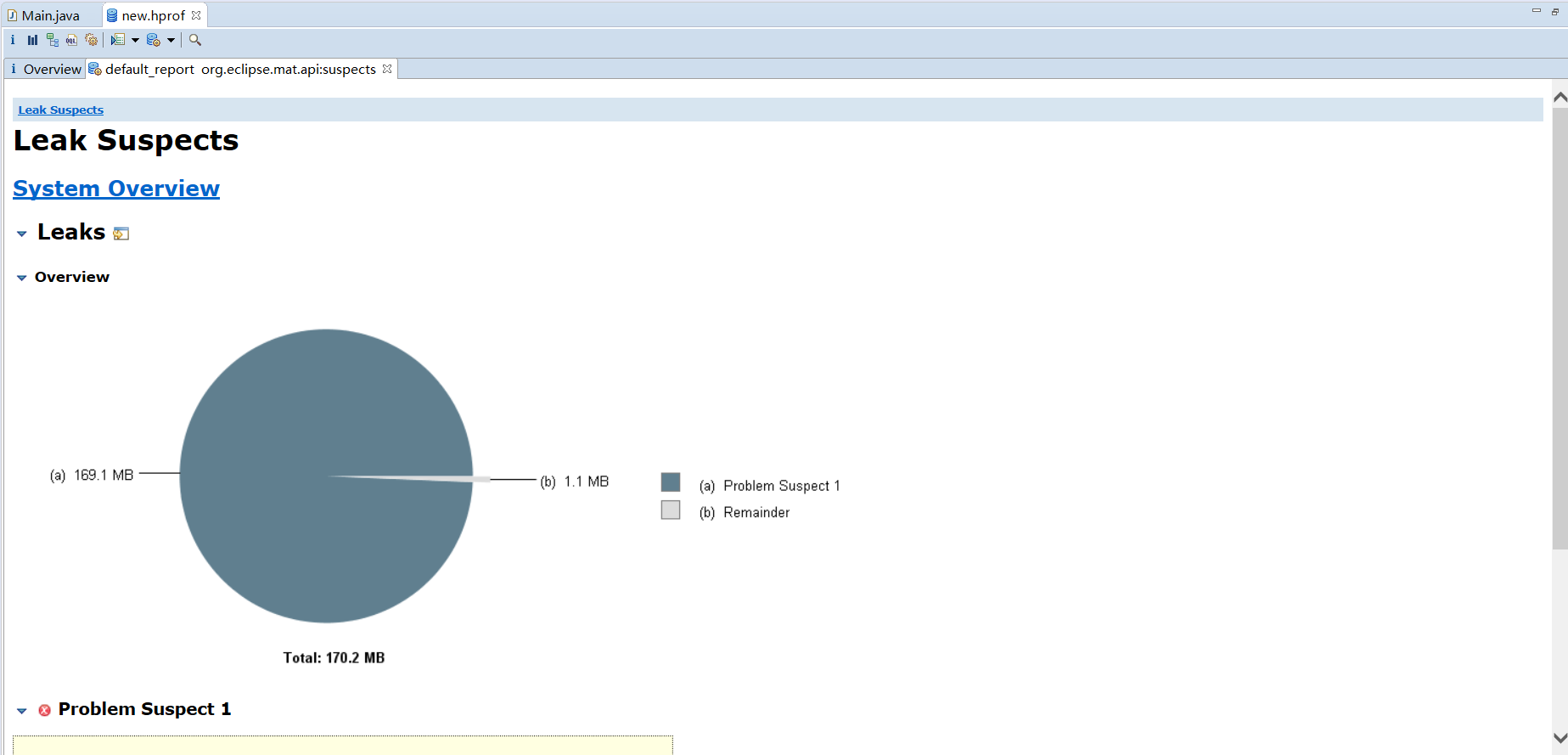


### 使用MAT分析内存导出文件

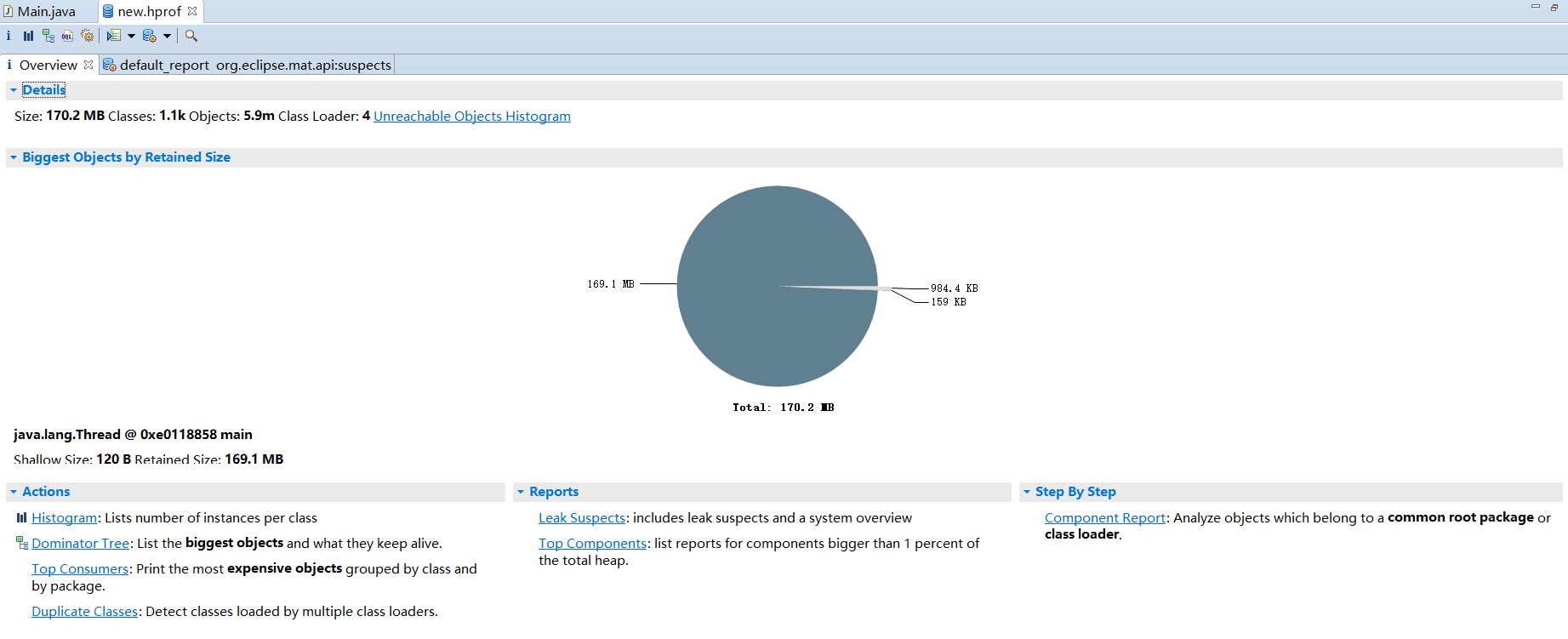
在Eclipse市场中搜索memory analysis tool，安装，然后直接打开.hprof文件即可。该对话框选中第一个即可。



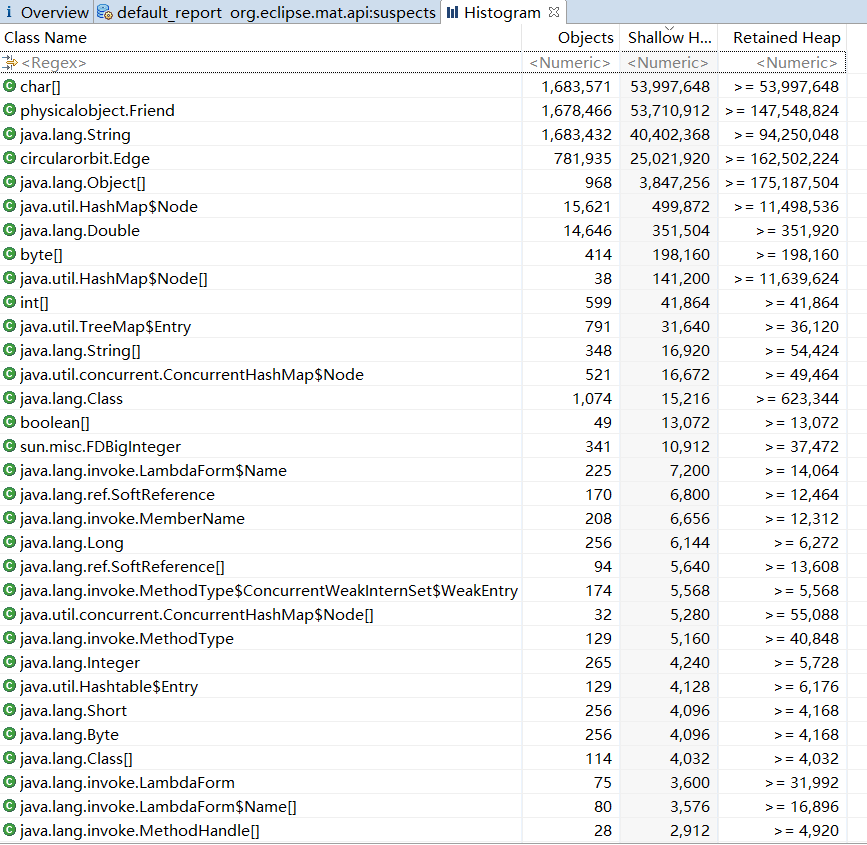
得到如下的分析界面



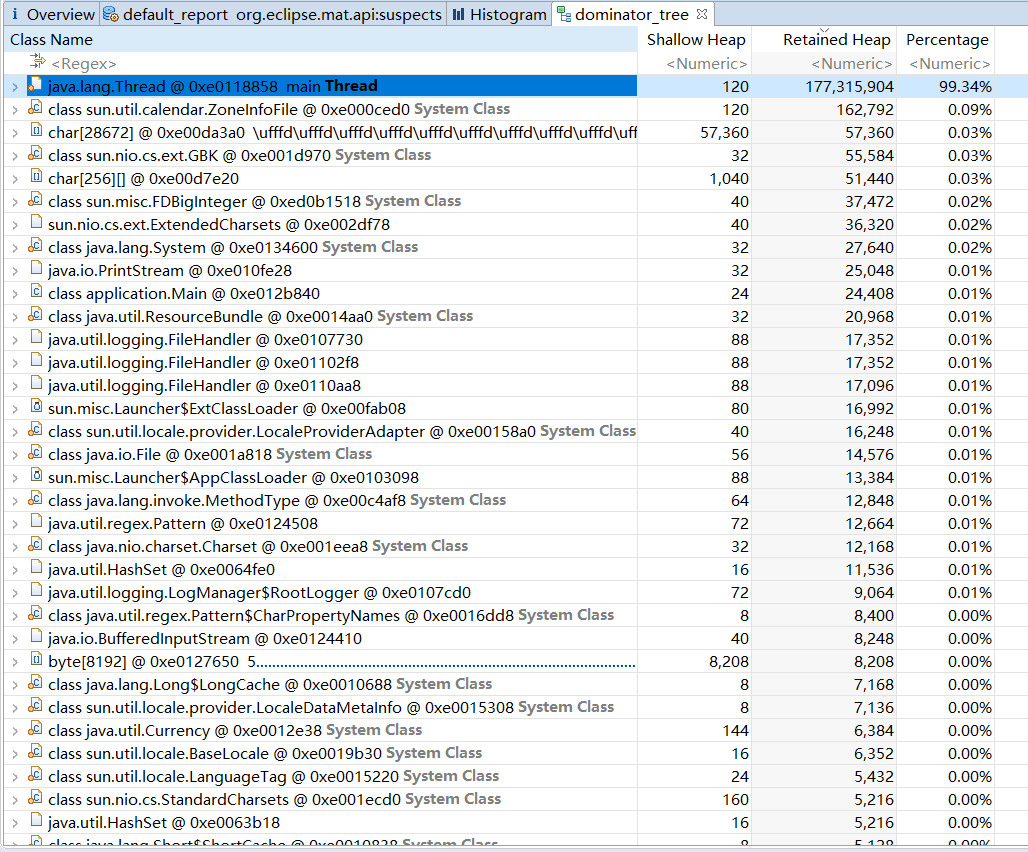
Overview视图



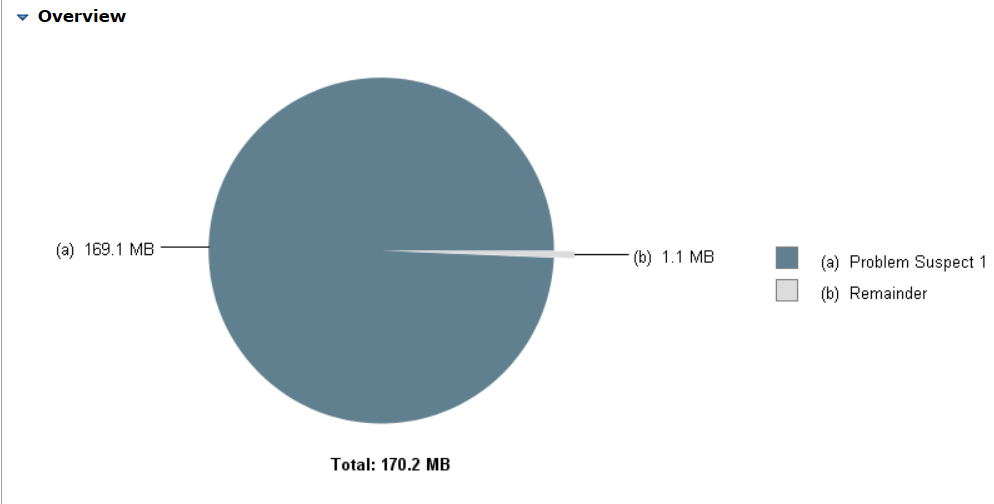
Histogram视图



Dominator tree视图



Top consumer视图



Leak suspect report

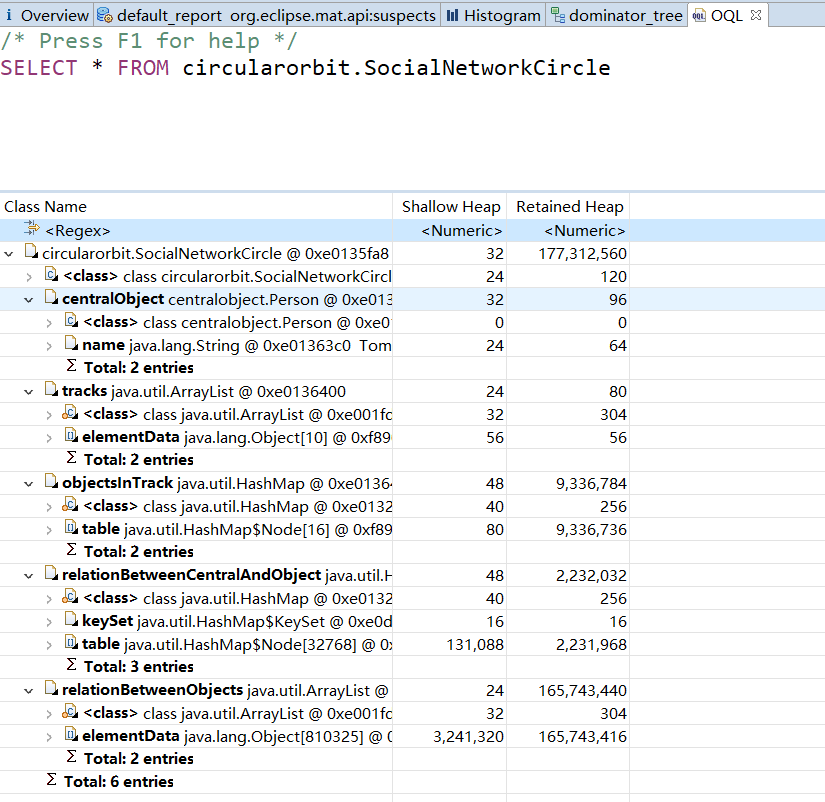


说明主线程中存有的中间变量太大了。

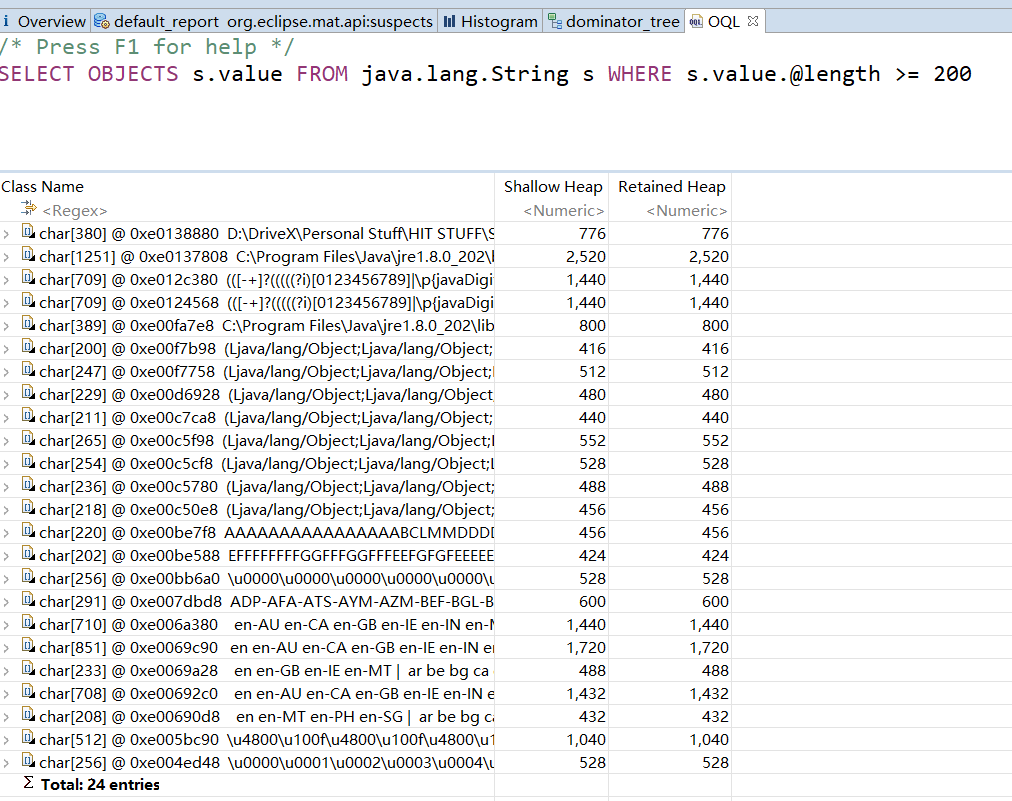
### 发现热点/瓶颈并改进、改进前后的性能对比分析

### 在MAT内使用OQL查询内存导出

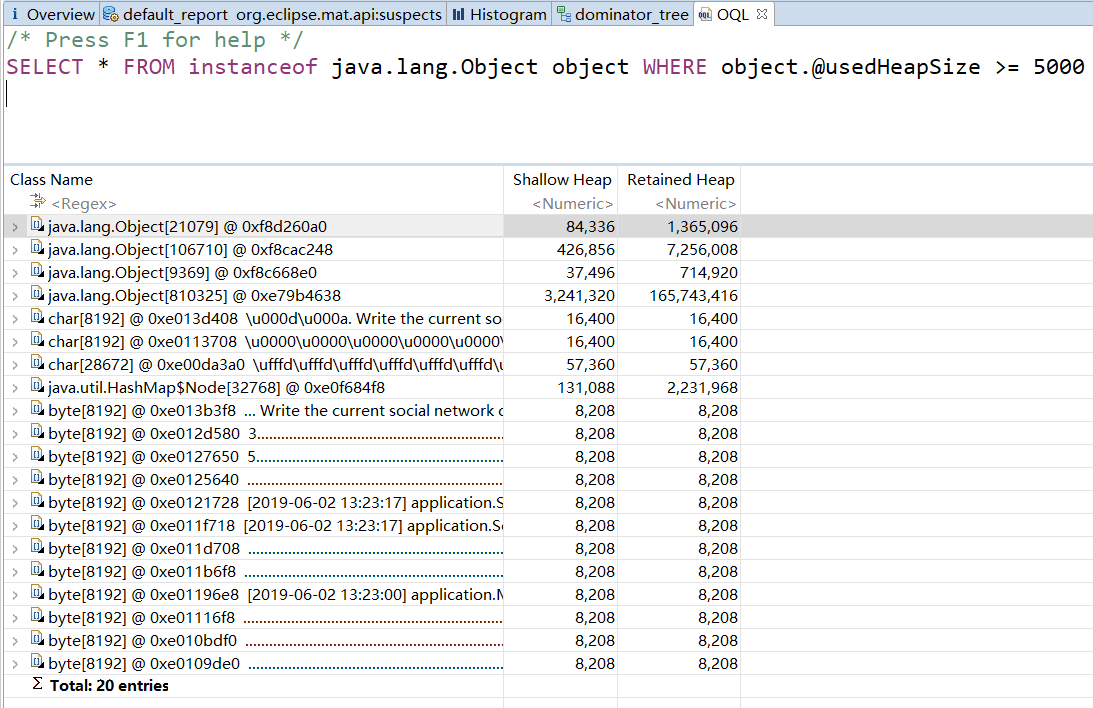
在社交关系网络中，读取文件后，所有轨道系统ADT的对象实例如下：



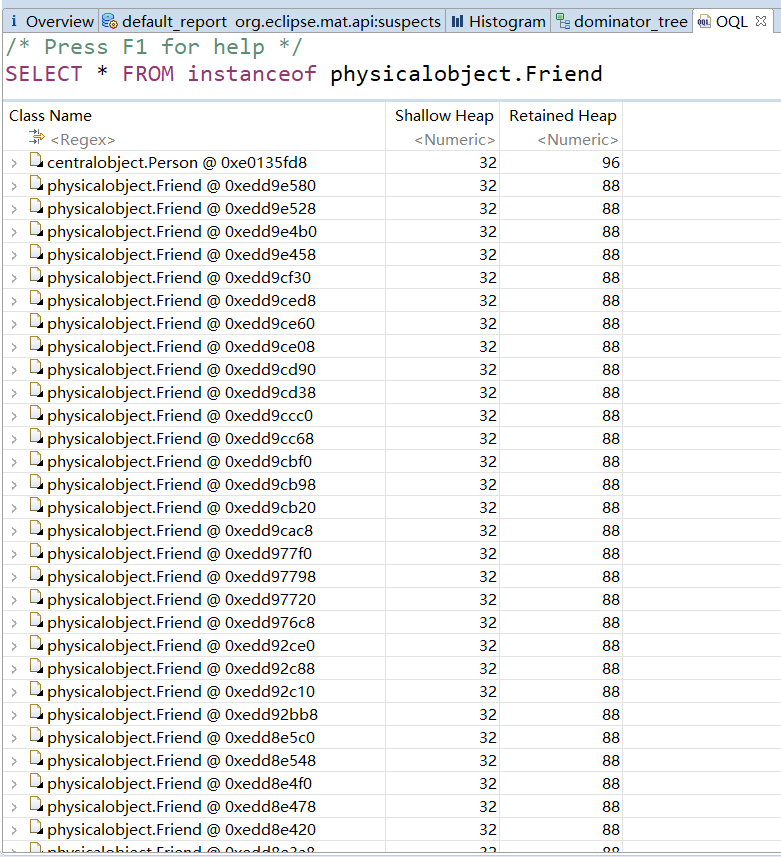
查找所有长度大于200的字串对象：



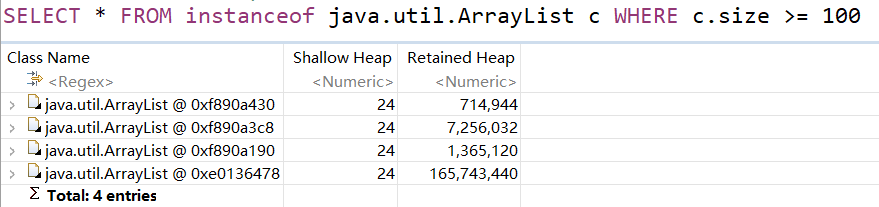
大于制定大小的任意类型对象实例

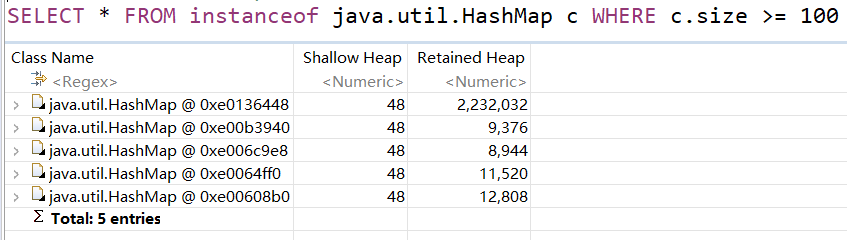


PhysicalObject的对象示例的数量和总占用内存大小

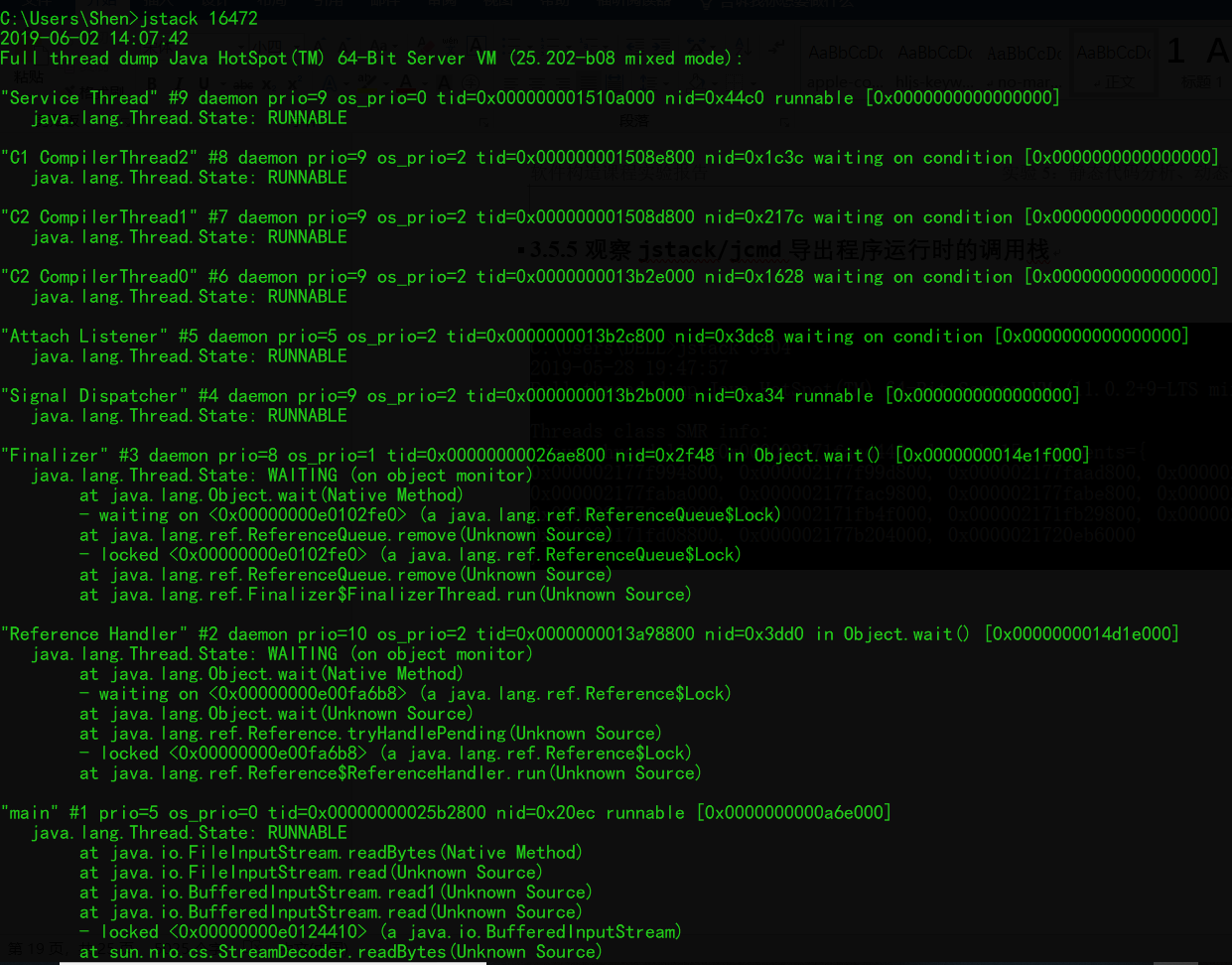


所有元素数量大于100的Collections实例：



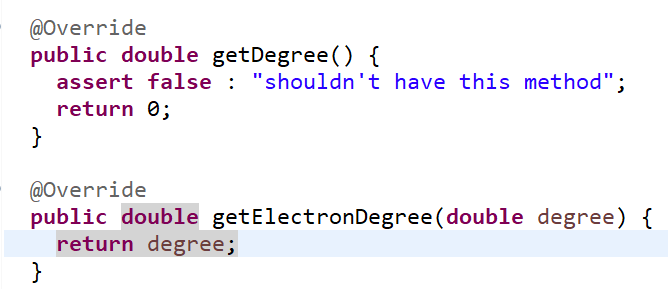


### 观察jstack/jcmd导出程序运行时的调用栈

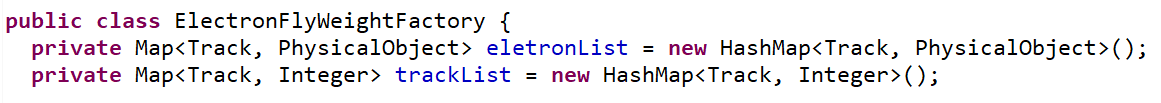


### 使用设计模式进行代码性能优化

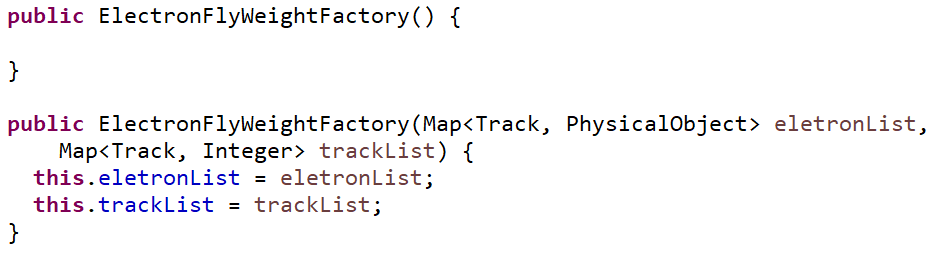
用flyweight设计模式对原子结构中的电子进行改造，改造的地方较多。首先电子对象是应用了PhysicalObject接口的，而所有电子都是相同的，唯独它的角度属性，在绘图时是不同的，是随机生成的。所以，按照flyweight设计模式，设计一个getElectronDegree方法，通过传参来获取电子的角度属性，而之前设计的getDegree方法，在电子中就不用了。

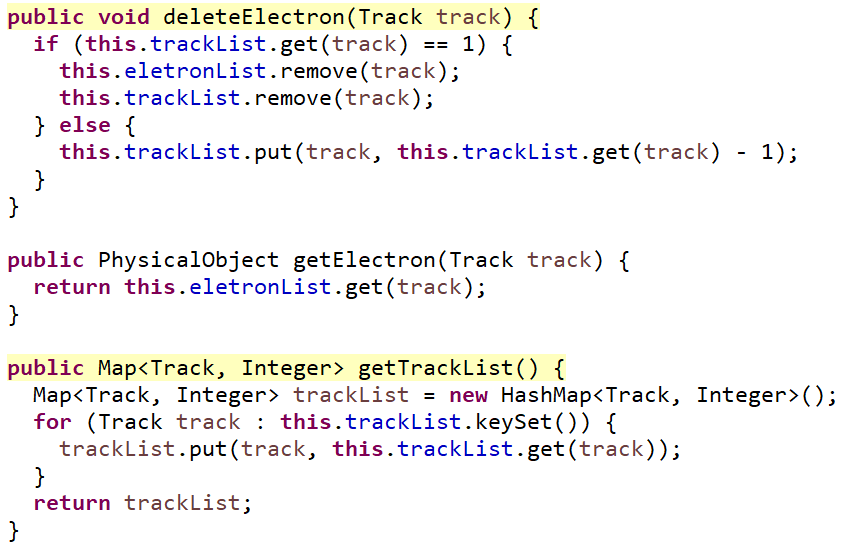


然后要设计一个工厂类，来管理轨道和电子的关系。又由于轨道上的电子虽然相同，但我们还需要得知轨道上的电子数，所以设计工厂类如下：



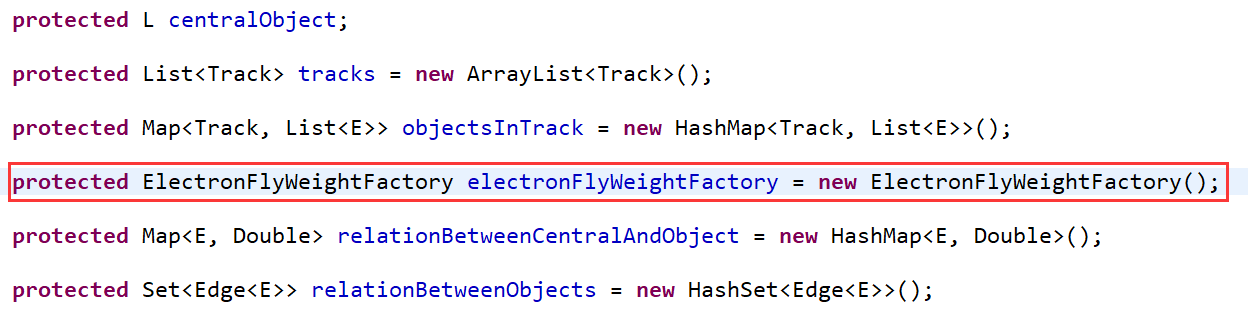
在该工厂类中定义必要的一些方法，



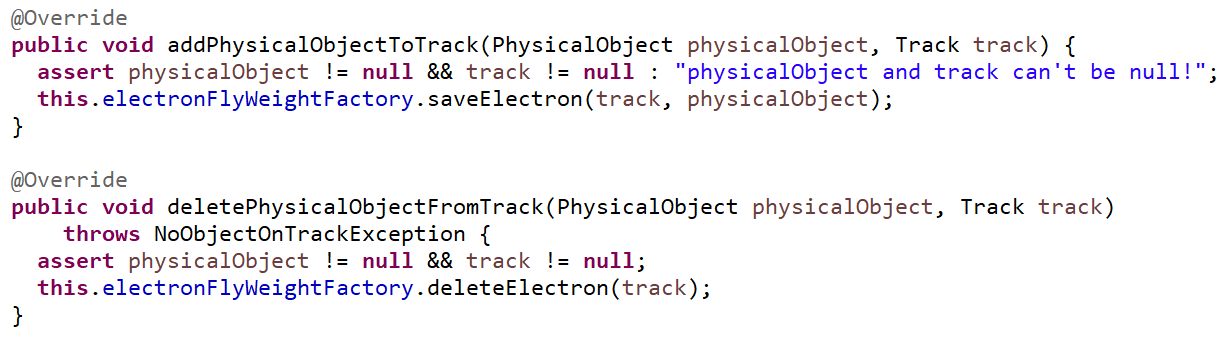


之所以需要两个构造器是因为在做防御式拷贝时，由于该工厂类的两个rep都是集合类对象，所以需要传入这两个rep的深拷贝对象，来构造新的工厂类对象。

在ConcreteCircularOrbit中的rep改造如下：



还有一些关于轨道上物体的增删操作，也需要在具体实现类中重写。



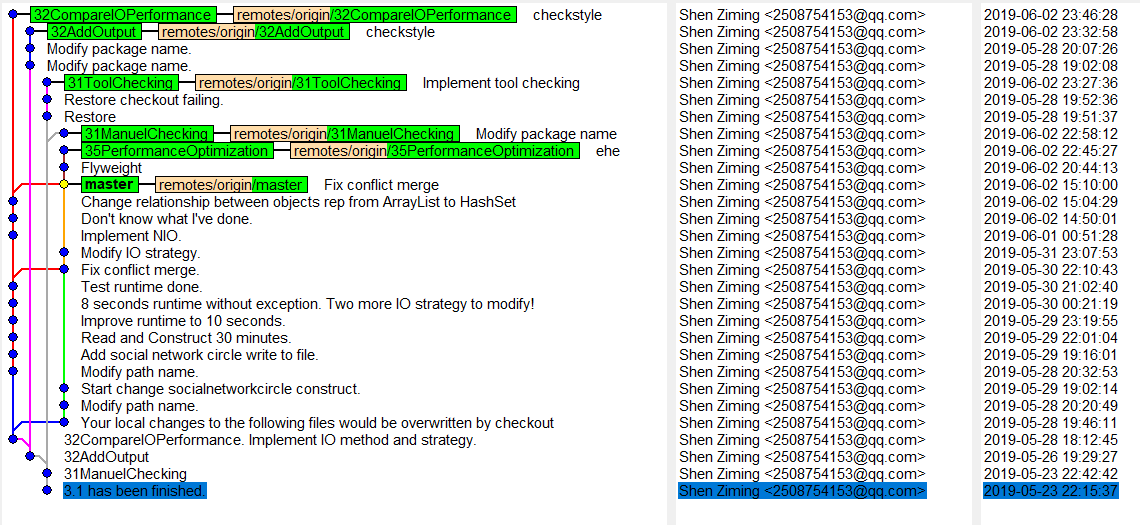
然后，在和这些改变相关的功能处，都要改变代码，比如在测试中：



这样一来，由工厂类中的rep可知，可以大大减少原子结构系统中，产生电子对象的个数，提高了性能。

## Git仓库结构





# 实验进度记录

|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 时间段 | 计划任务 | 实际完成情况 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# 实验过程中遇到的困难与解决途径

|  |  |
| --- | --- |
| 遇到的难点 | 解决途径 |
| 读取文件时，构造社交网络关系耗时太长 | 优化存储的数据结构，尽量多使用hash结构，具体可见上文报告 |
| 对电子用flyweight模式改造时，增删电子操作无法实现 | 在工厂类中加一个rep来管理每条轨道上的电子数 |
|  |  |

# 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

尤其学会了IO性能优化，这部分做了很久。其它的，学会了对JVM参数配置，还有java的自带的一些工具的使用。

## 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？

**肯定是运行起来更美更重要。保证正确性，提高健壮性，至于代码写的美不美，都是次要的，在团队开发中，只是作为提高工作效率的一部分，对发布来说，没有影响。**

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？

**我认为会，但帮助不大。就实验的代码量来说，以个人为单位写代码，代码风格完全无关紧要，而spotbugs就相对重要一点，他给出的bug检查是在运行中可能发生的，但是静态检查以及我们的等价类检查可能都没有发现的，这在发布中是很危险的，建议全部改正。**

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？
2. 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？
3. 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？
4. 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？
5. 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？
6. 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？
7. 关于本实验的工作量、难度、deadline。

本实验工作量非常难受，难度非常难受，deadline合理。本次实验不是教学重点，教学课时不多，需要自学的地方太多太多，而那些参数分析和优化分析在网上没有什么太好的自学资源，总体上感觉比较无助，私以为，这是到目前为止做的效果最差的一次实验，除了IO性能优化印象比较深刻以外，其它的效果都不好，着实感觉没什么实用价值。

1. 到目前为止，你对《软件构造》课程的意见与建议。