

**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 | 林之浩 |
| 学号 | 1170300817 |
| 班号 | 1703008 |
| 电子邮件 | [630073498@qq.com](mailto:630073498@qq.com) |
| 手机号码 | 18065053516 |

**目录**

[1 实验目标概述 1](#_Toc10321725)

[2 实验环境配置 1](#_Toc10321726)

[3 实验过程 2](#_Toc10321727)

[3.1 Static Program Analysis 2](#_Toc10321728)

[3.1.1 人工代码走查（walkthrough） 2](#_Toc10321729)

[3.1.1.1 错误1：命名不规范 2](#_Toc10321730)

[3.1.1.2 错误2：注释不规范 2](#_Toc10321731)

[3.1.1.3 错误3：大括号 3](#_Toc10321732)

[3.1.2 使用CheckStyle和SpotBugs进行静态代码分析 3](#_Toc10321733)

[3.1.2.1 注释 3](#_Toc10321734)

[3.1.2.2 导入 3](#_Toc10321735)

[3.1.2.3 单行字符数 3](#_Toc10321736)

[3.1.2.4 Javadoc 4](#_Toc10321737)

[3.1.2.5 空格/Tab错误 4](#_Toc10321738)

[3.2 Java I/O Optimization 4](#_Toc10321739)

[3.2.1 多种I/O实现方式 4](#_Toc10321740)

[3.2.2 多种I/O实现方式的效率对比分析 7](#_Toc10321741)

[3.3 Java Memory Management and Garbage Collection (GC) 9](#_Toc10321742)

[3.3.1 使用-verbose:gc参数 9](#_Toc10321743)

[3.3.2 用jstat命令行工具的-gc和-gcutil参数 10](#_Toc10321744)

[3.3.3 使用jmap -heap命令行工具 12](#_Toc10321745)

[3.3.4 使用jmap -clstats命令行工具 13](#_Toc10321746)

[3.3.5 使用jmap -permstat命令行工具 13](#_Toc10321747)

[3.3.6 使用JMC/JFR、jconsole或VisualVM工具 13](#_Toc10321748)

[3.3.7 分析垃圾回收过程 14](#_Toc10321749)

[3.3.8 配置JVM参数并发现优化的参数配置 15](#_Toc10321750)

[3.4 Dynamic Program Profiling 16](#_Toc10321751)

[3.4.1 使用JMC或VisualVM进行CPU Profiling 16](#_Toc10321752)

[3.4.2 使用VisualVM进行Memory profiling 17](#_Toc10321753)

[3.5 Memory Dump Analysis and Performance Optimization 17](#_Toc10321754)

[3.5.1 内存导出 17](#_Toc10321755)

[3.5.2 使用MAT分析内存导出文件 18](#_Toc10321756)

[3.5.3 发现热点/瓶颈并改进、改进前后的性能对比分析 23](#_Toc10321757)

[3.5.4 在MAT内使用OQL查询内存导出 24](#_Toc10321758)

[3.5.5 观察jstack/jcmd导出程序运行时的调用栈 29](#_Toc10321759)

[3.5.6 使用设计模式进行代码性能优化 30](#_Toc10321760)

[3.6 Git仓库结构 32](#_Toc10321761)

[4 实验进度记录 33](#_Toc10321762)

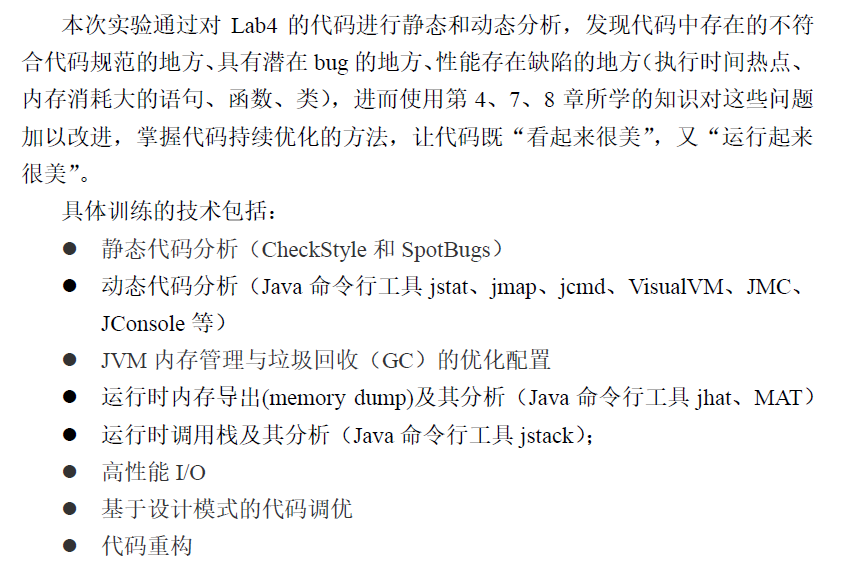
[5 实验过程中遇到的困难与解决途径 33](#_Toc10321763)

[6 实验过程中收获的经验、教训、感想 33](#_Toc10321764)

[6.1 实验过程中收获的经验和教训 33](#_Toc10321765)

[6.2 针对以下方面的感受 33](#_Toc10321766)

# 实验目标概述



# 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。

从指导书上的网址下载的MAT，visualvm用的是自带的（从cmd进入输入jvisualvm即可）

在这里给出你的GitHub Lab5仓库的URL地址（Lab5-学号）。

<https://github.com/ComputerScienceHIT/Lab5-1170300817>

# 实验过程

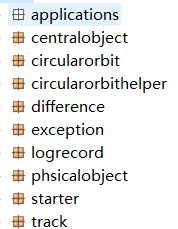
请仔细对照实验手册，针对每一项任务，在下面各节中记录你的实验过程、阐述你的设计思路和问题求解思路，可辅之以示意图或关键源代码加以说明（但千万不要把你的源代码全部粘贴过来！）。

## Static Program Analysis

### 人工代码走查（walkthrough）

#### 错误1：命名不规范

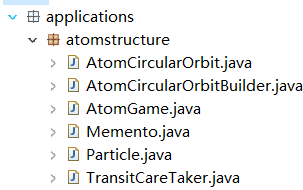
包名应该全部是小写字符



局部变量命名应该符合：'^[a-z]([a-z0-9][a-zA-Z0-9]\*)?$'

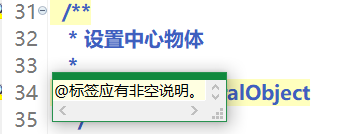
方法名需要符合： '^[a-z][a-z0-9][a-zA-Z0-9\_]\*$'.

类的命名应该符合'^[A-Z][a-zA-Z0-9]\*$'



#### 错误2：注释不规范

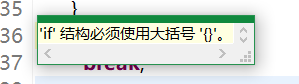
所有@参数后面都必须有说明，不能为空。



尽可能地使用@Override标记重写。

#### 错误3：大括号

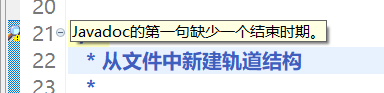
if，else，for，do和while语句后面必须跟有大括号。



### 使用CheckStyle和SpotBugs进行静态代码分析

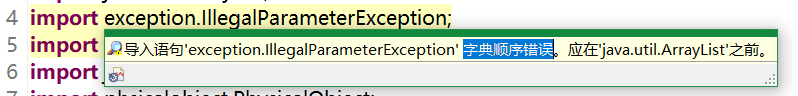
#### 注释

Javadoc第一句描述必须要有一个句点



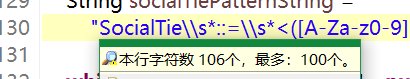
#### 导入

导入语句要遵守字典顺序。



#### 单行字符数

每行字符不超100个。用+连接长字符串。

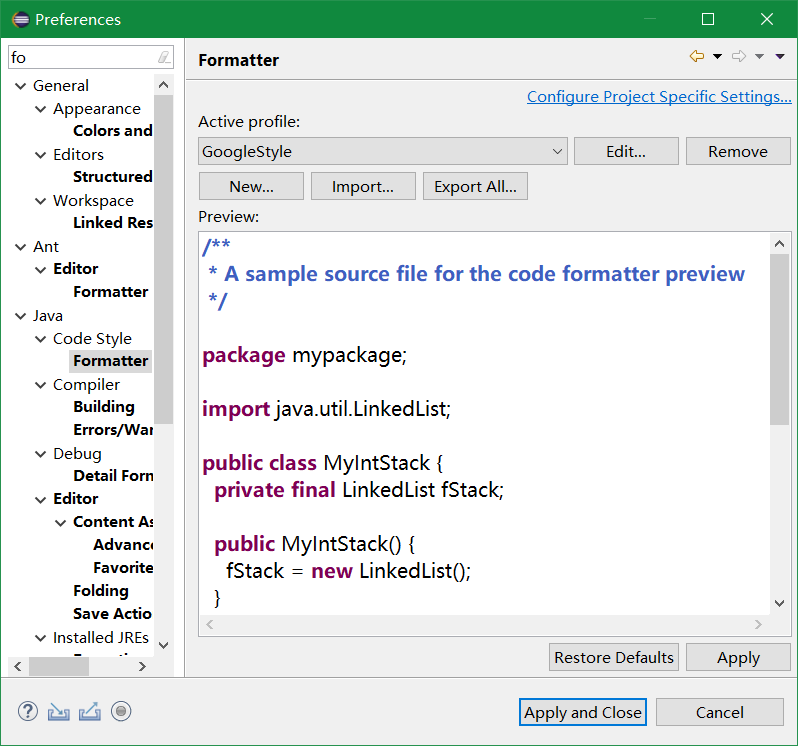


#### Javadoc

每个方法都要有文档，除了简单的getter和setter方法。

#### 空格/Tab错误

从网上下载了一个eclipse用的谷歌风格的formatter配置文件导入后一键format就行了。



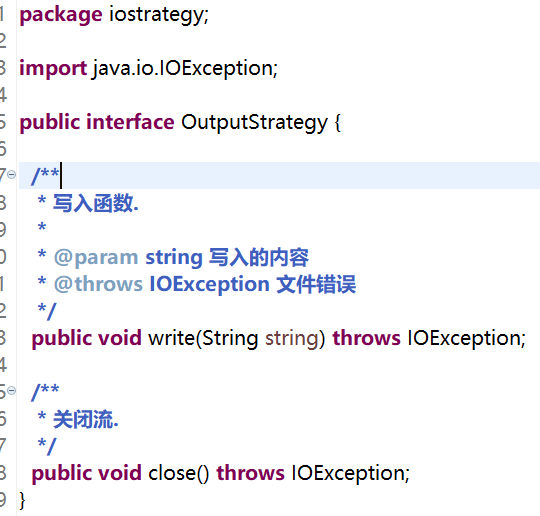
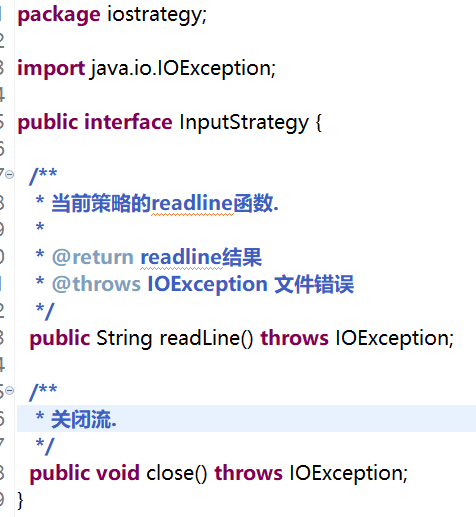
区别：手动检查只能发现一些很明显的错误，明显用工具检查更方便快速，但是手动检查加强一下影响能养成好习惯。

## Java I/O Optimization

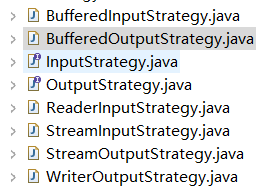
### 多种I/O实现方式

实现了Stream，Reader/Writer，Buffer三种读写方式，具体实现：

先声明两个接口，分别是读写策略。



随后实现以下六个子类，分别实现各自的readline方法和close方法。

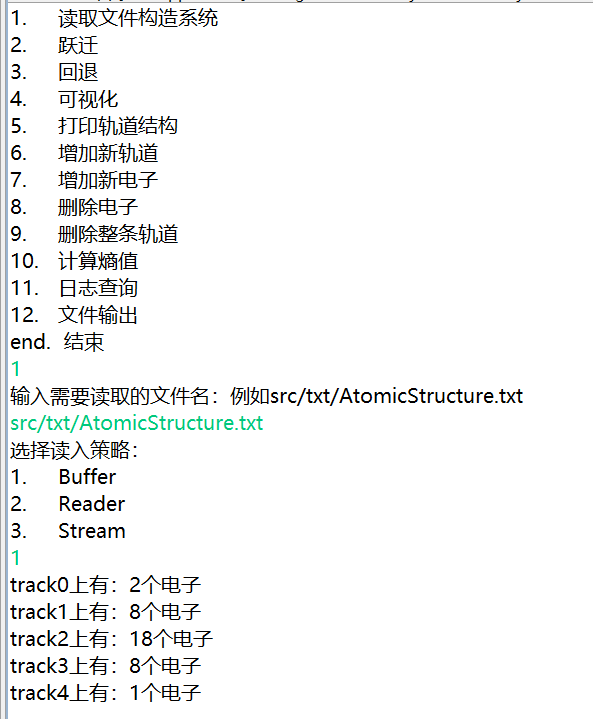
分别使用三种方法实现接口，

使用时，选择要使用的读写策略，使用文件名构造一个新的strategy，之后传入构造轨道结构的函数当做输入流。

切换只需要在控制台选择



控制台演示：



### 多种I/O实现方式的效率对比分析

时间计算方式：

在读取文件的循环的头尾加上





最后输出：

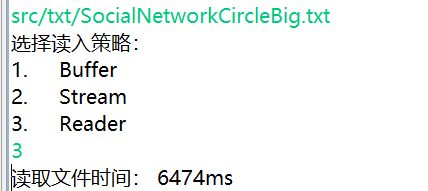
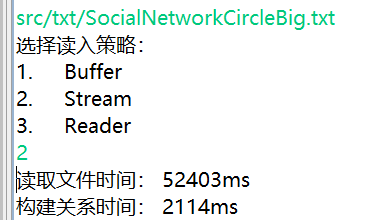


这里以socialnetwork为例：（读取的是79万行的测试文件，为了区分以前的，文件名后面加了Big）

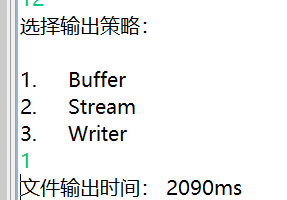
分别测试三种读写方式的用时：

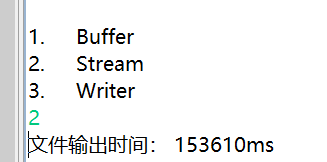
读入：

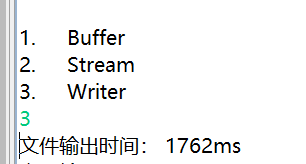




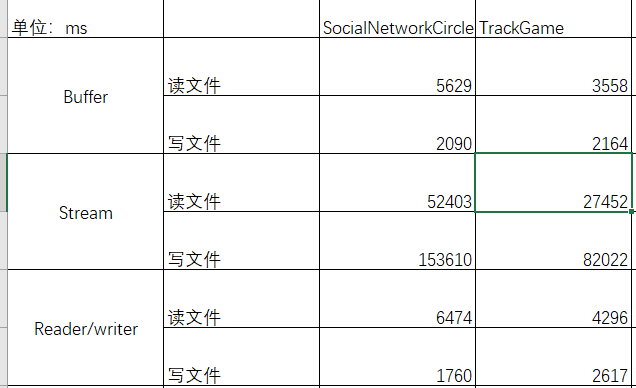
写出：



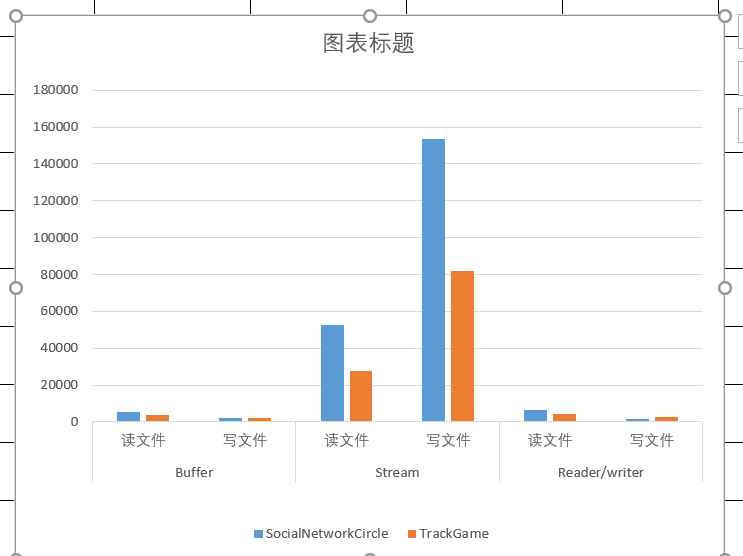




数据统计：



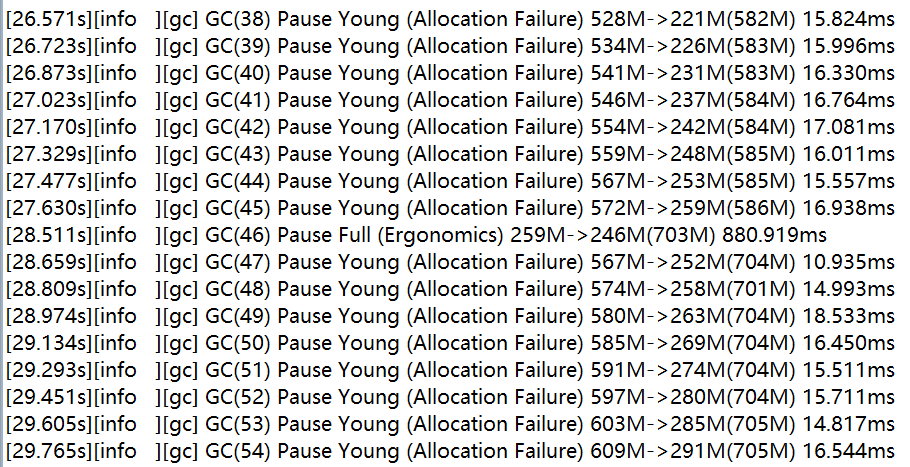
图表：



## Java Memory Management and Garbage Collection (GC)

### 使用-verbose:gc参数

使用SocialNetWork做测试，读入大文件：



由于本人电脑内存有24G，jvm默认最大heapsize默认最大内存是物理内存的1/4，所以修改了最大堆内存

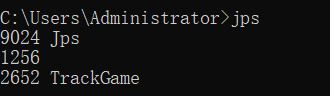
使用：-Xmx1024m。

调整最大heapsize为1024MB，使用trackgame的大文件反复读取多次之后终于是出现了full GC的情况，可以看见：

1. full gc的出现频率远远低于minor gc。
2. 耗费的时间上看，minor gc的用时也远远低于full gc，其中的原理也显而易见：full gc需要整理年轻代，老年代和永久代，代价自然很高

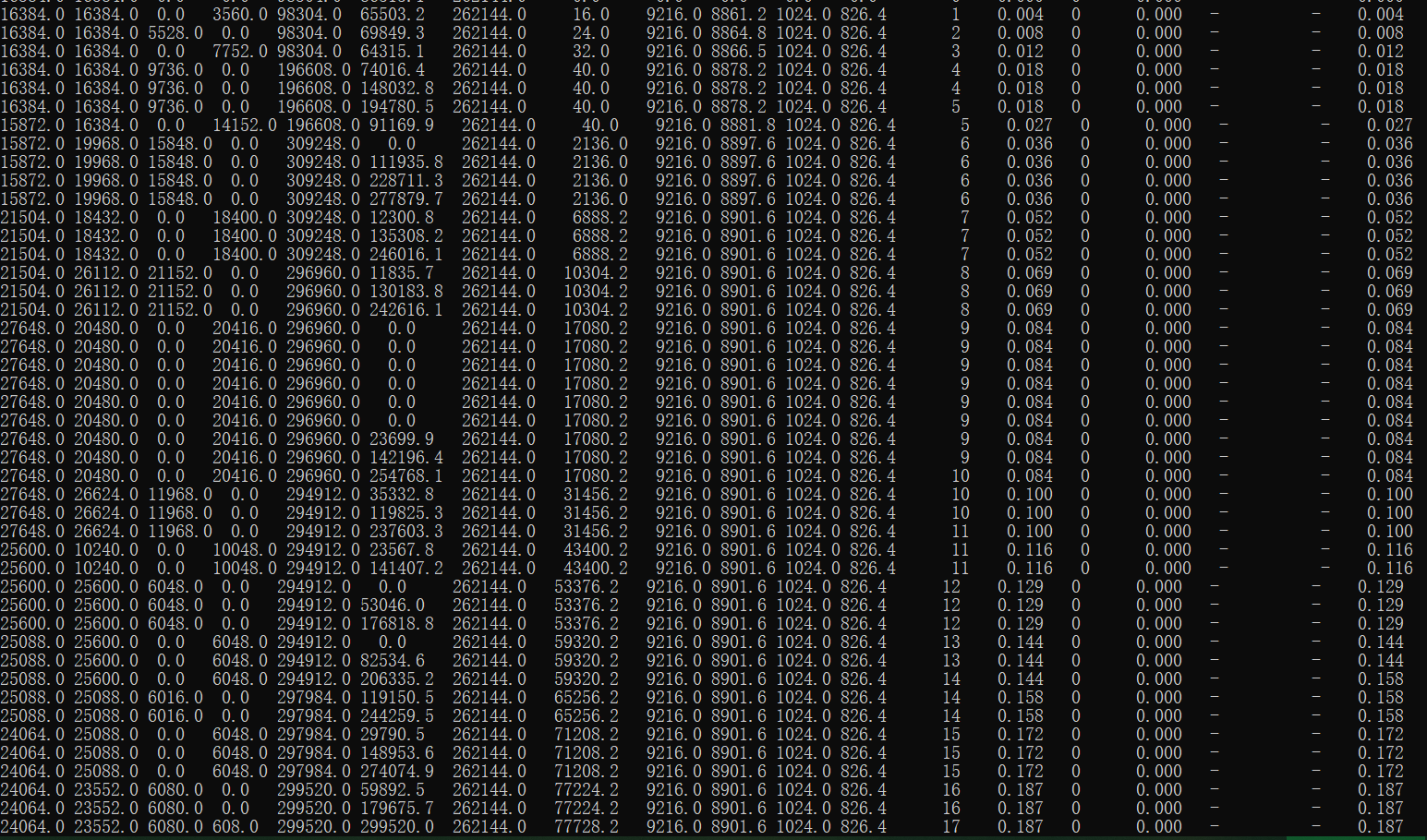
### 用jstat命令行工具的-gc和-gcutil参数

使用jps命令获取进程ID：



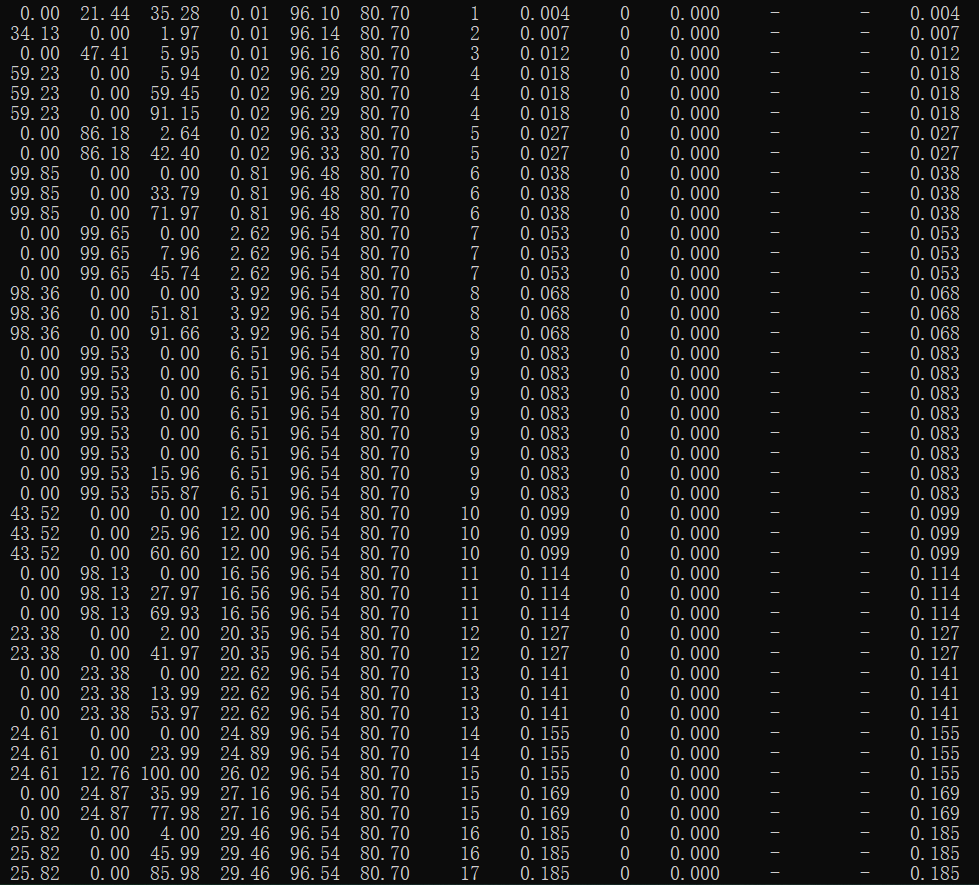
使用-gc参数：





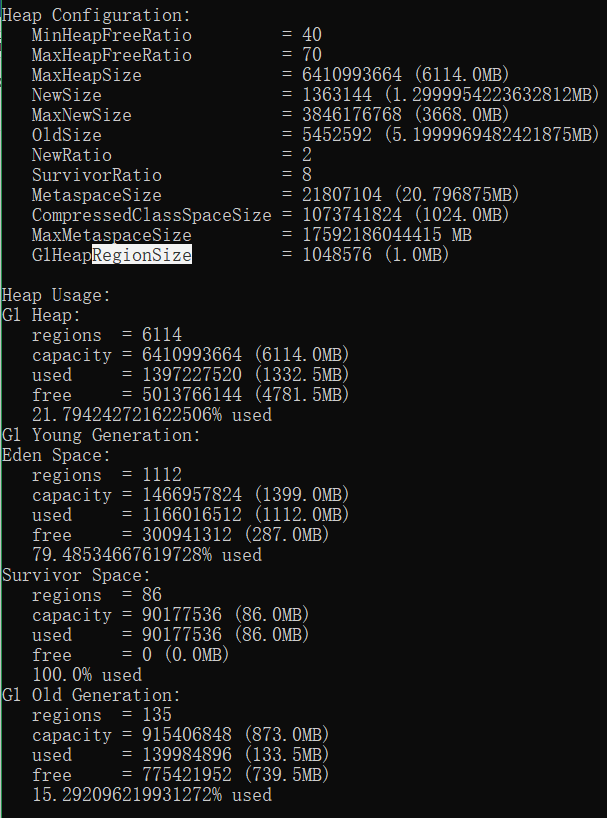
使用-gcutil参数：





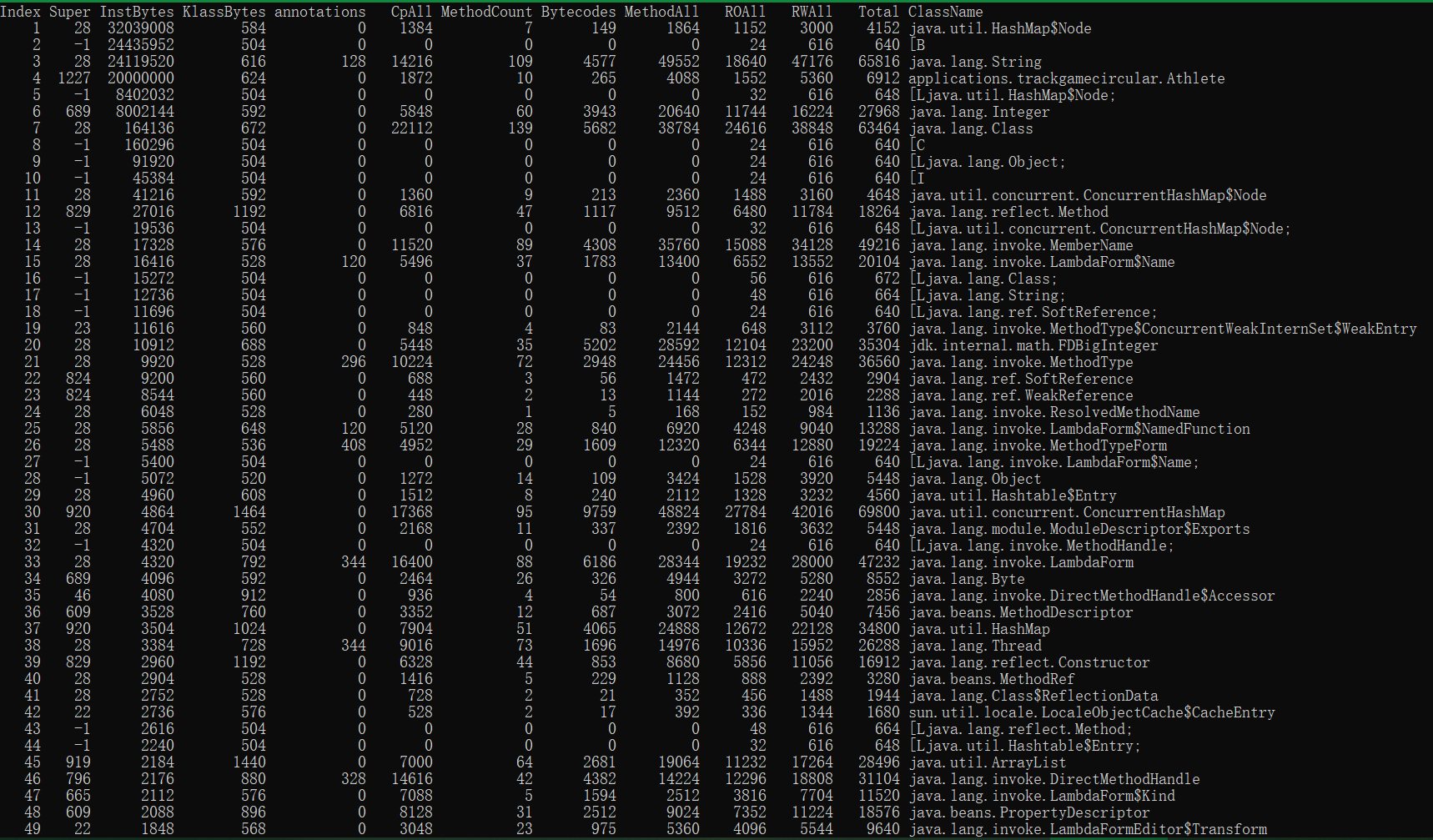
读取文件操作没发现异常，全部是minor gc，full gc次数为0.

### 使用jmap -heap命令行工具



可以查看jvm此时的一些参数，这里的最大heap size是我改动过的1G。

### 使用jmap -clstats命令行工具

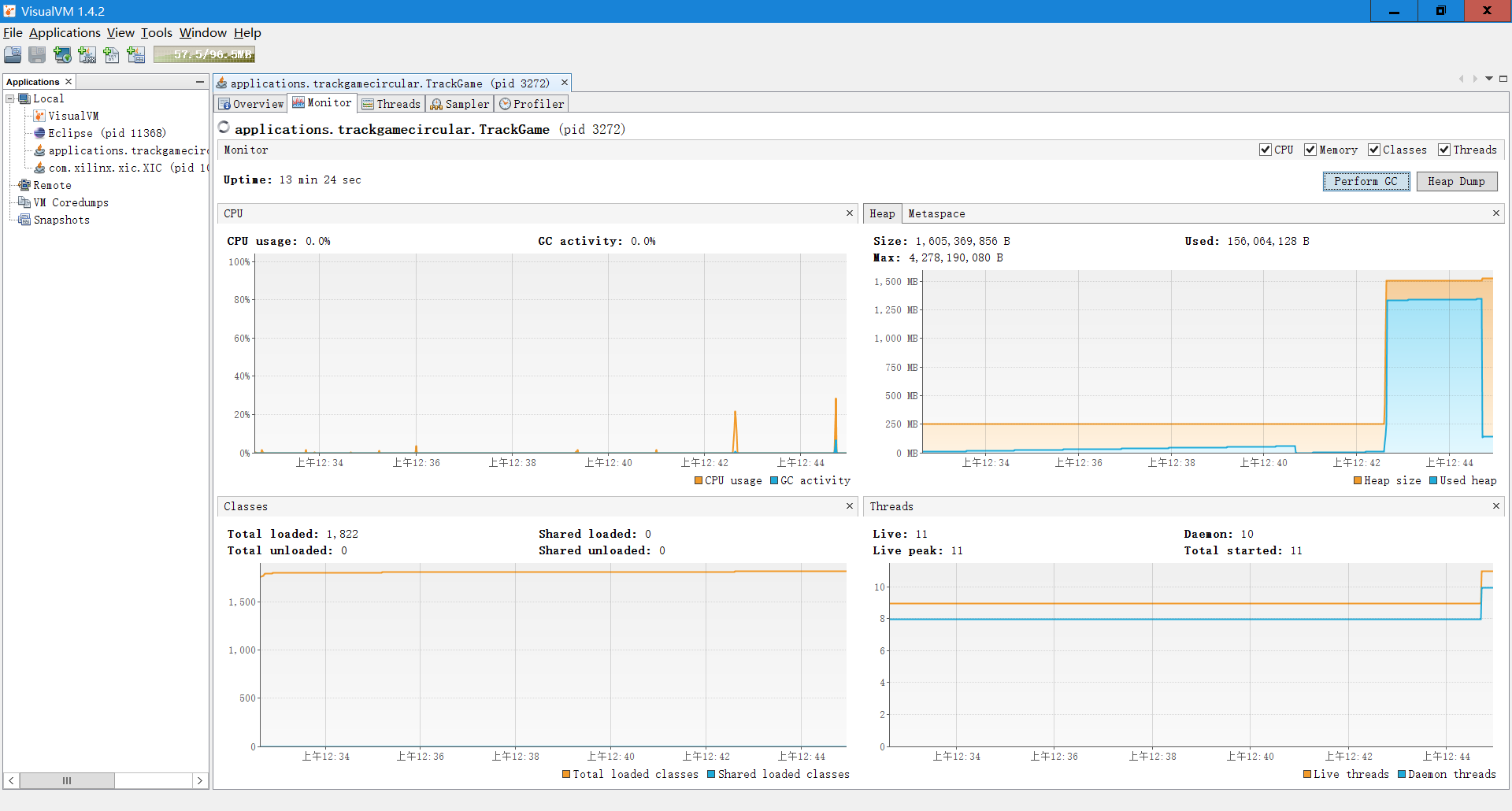


下面还有很多不截取了，可以注意到第四行显示了我申请了多少个运动员对象。

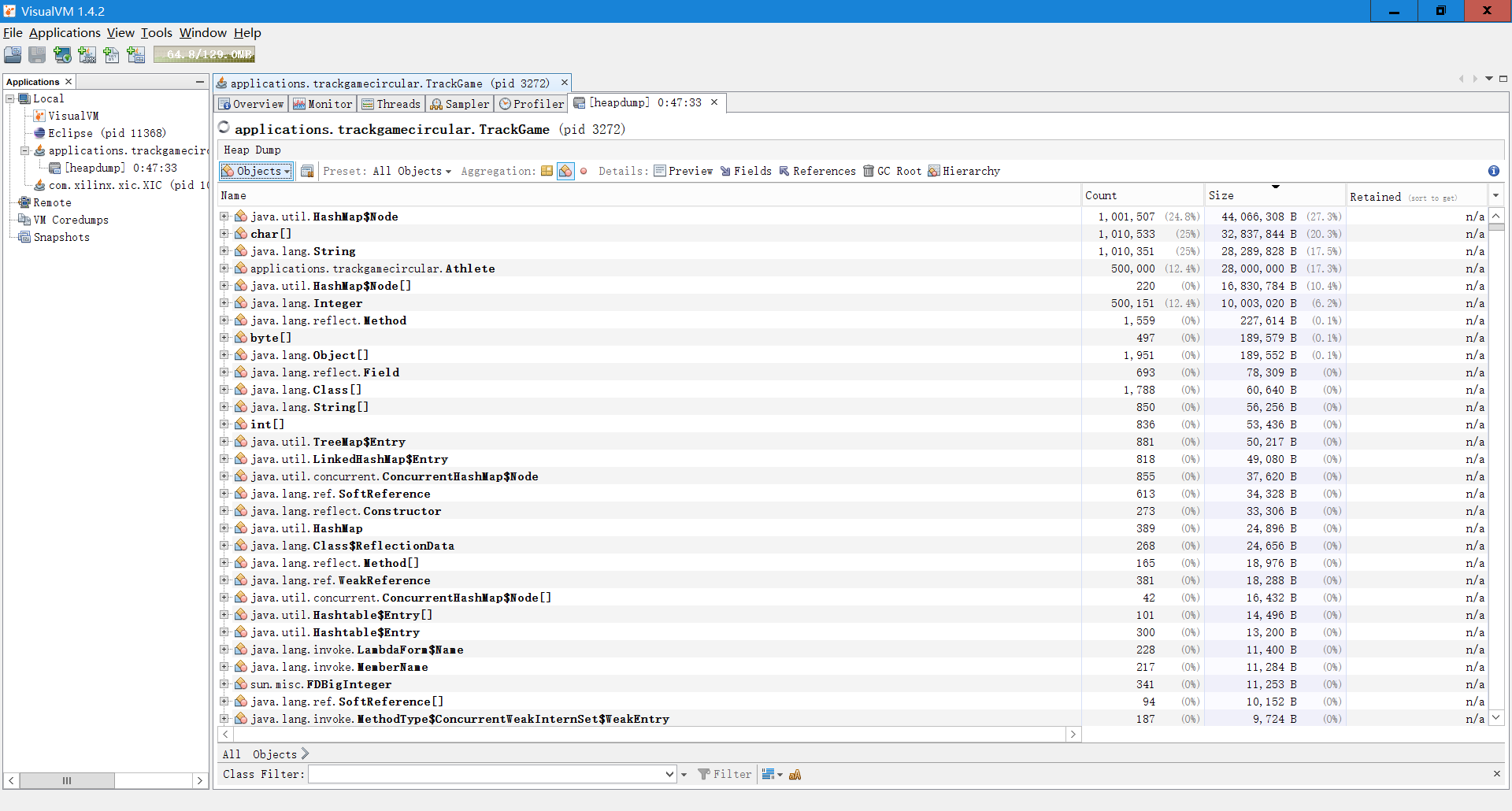
### 使用jmap -permstat命令行工具

-permstat就是java8之前的-clstats参数，目录有误。

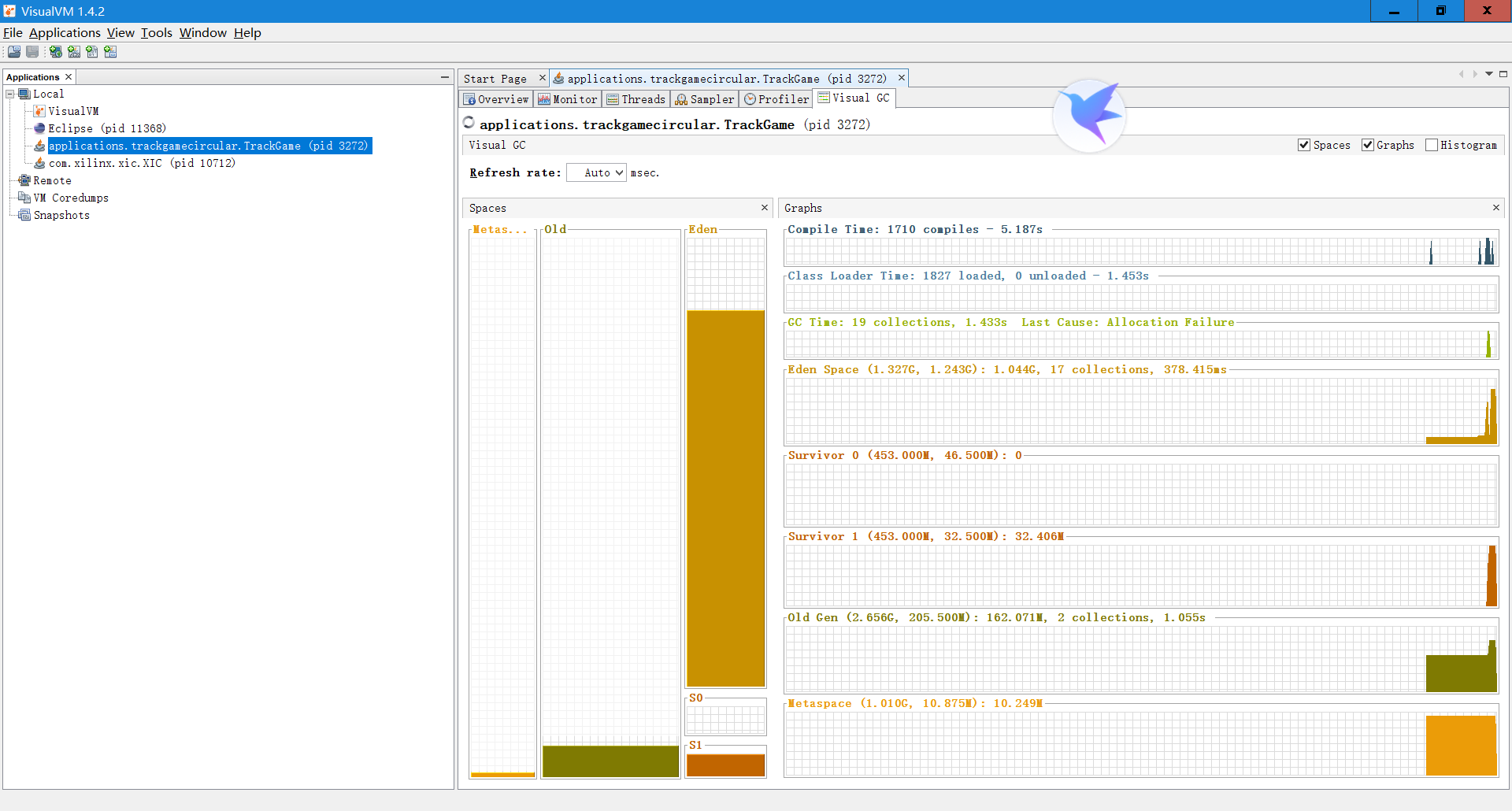
### 使用JMC/JFR、jconsole或VisualVM工具



heap的分配和使用：



使用visualGC插件查看



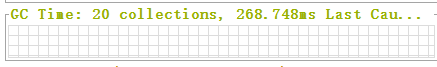
### 分析垃圾回收过程

垃圾回收过程：

1. 当新的对象申请之后，会被储存在新生代的Eden区中，从它的名字也很好理解，Eden，就是诞生的地方的意思。
2. 当eden区放满了之后，进行一次minor GC，对eden区的对象进行筛选，第一次是将存活的对象复制到S0区，之后将eden区域清空，下一次的minor GC，S0和S1的角色互换。
3. 存活过一定次数的minor GC之后的对象就会被放置到老年代之中。
4. 在程序进行了很多次minor GC之后，就有可能老年代的空间已经不够用了，此时会发生一次full GC，相比于minor GC，full GC的时间开销比较大，因为full GC针对老年代，年轻代和永久代。

### 配置JVM参数并发现优化的参数配置

最开始设置的是-Xmx32m –Xms4m马上发现根本不够用的，最大heapsize扩大到1g时的效果如下：（只是读取文件）：



2g时：



根据控制变量的思想，发现扩大最大堆大小似乎有利于加快gc速度，

于是改为4g最大堆大小



再改为8g：



可以发现，此时过大的空间对gc速度的损害已经暴露出来了，特别是巨大的eden空间，这样我们就确定最大的堆大小。

接下来我们设置最小堆大小为2g

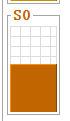


可以发现最小堆大小扩大可以有效得减少minorGC得次数

观察发现，eden space的大小会一直扩大到1.2g左右，于是我们通过指定新生代的大小：使用-Xmn 1536m来指定大小：



虽然时间差异不大但是也减小了回收次数。

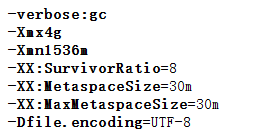


发现S0和S1在过程中始终处于放不满的状态，于是-XX:SurvivorRatio=8调节一下

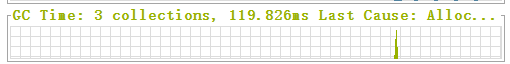
我们还发现metaspace的空间一直没有什么使用，于是使用：-XX:MetaspaceSize=30m

和-XX:MaxMetaspaceSize=30m

所以最后使用的参数如下

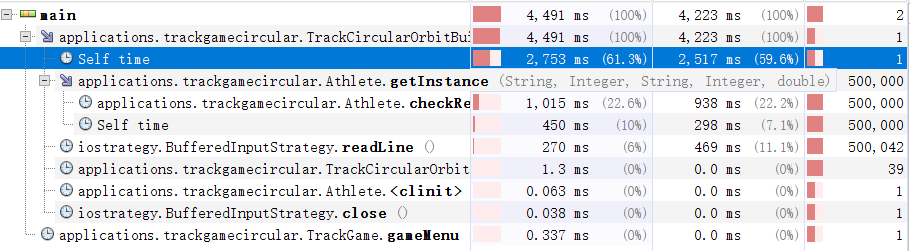


相比default设置也是快了不少：



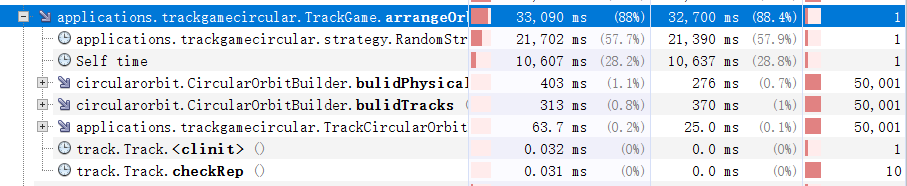
## Dynamic Program Profiling

### 使用JMC或VisualVM进行CPU Profiling



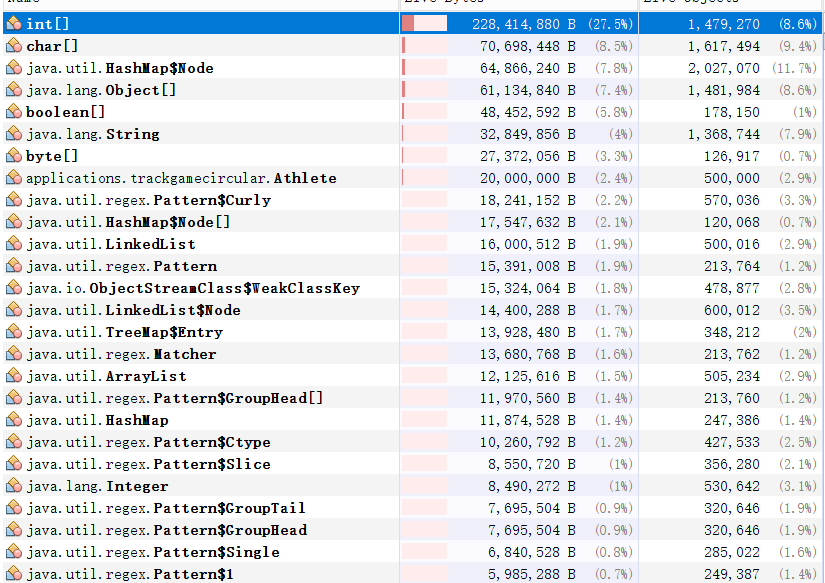
先看文件读取环节，主要是构建函数的时间开销打，主要开销为：正则表达式匹配，和文件读取的readline函数，构造person实例时也可以发现，主要是checkrep占据了时间，是因为我在checkrep中检查了各个运动员参数是否合法，同样也是基本正则匹配的开销。

再看安排比赛的开销：



和我预想的一样，时间开销还是主要花在了strategy的执行上，所以，开销都是合理的，只是有点慢。

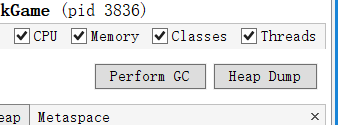
### 使用VisualVM进行Memory profiling



总体来说正常，除了int和char数组，最高的就是hashmap的node了，这是为了快速查找时申请的hashmap，之后时pattern，因为多次匹配字符串的原因，这一类对象也特别多，运动员的人数正好是500000。总体合理。

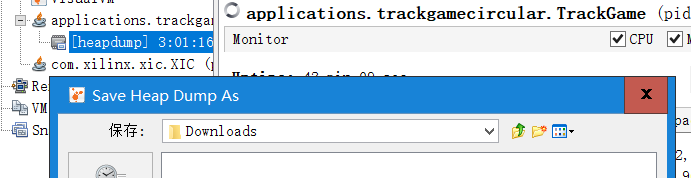
## Memory Dump Analysis and Performance Optimization

### 内存导出



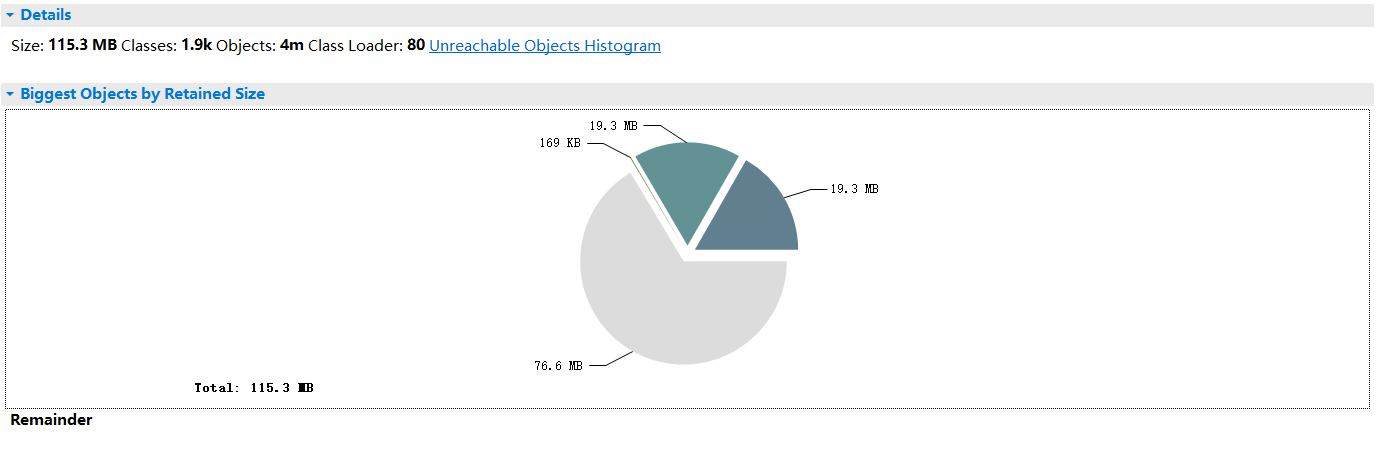
点击heap dump之后

右键就可以另存为了。

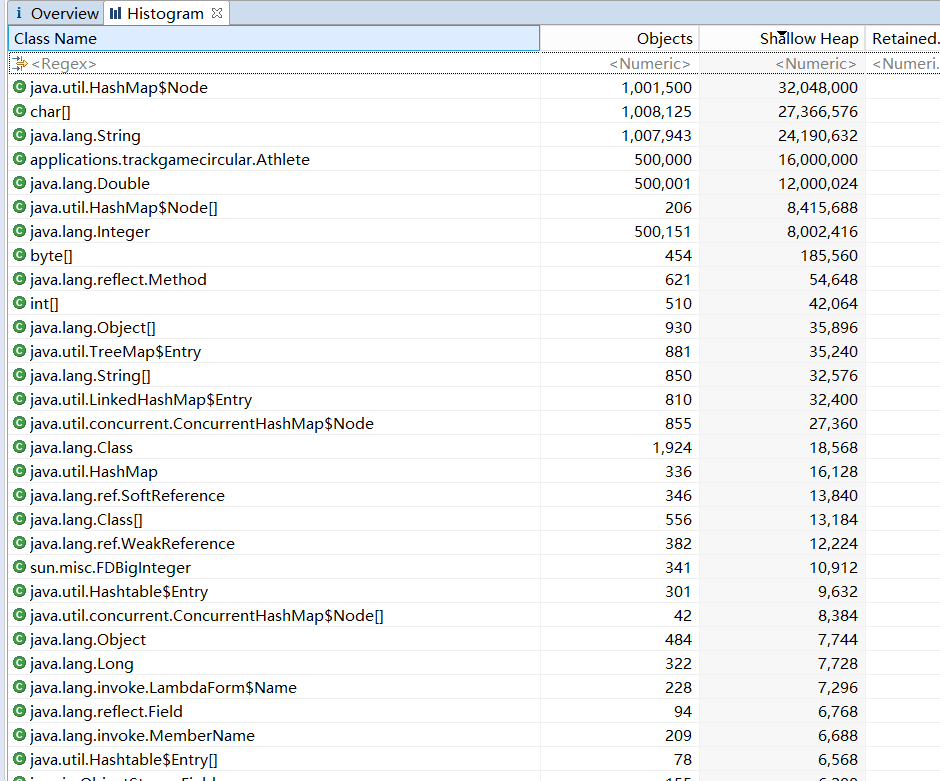


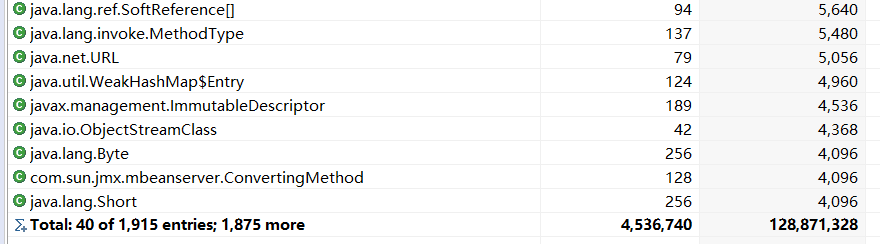
### 使用MAT分析内存导出文件

Overview：



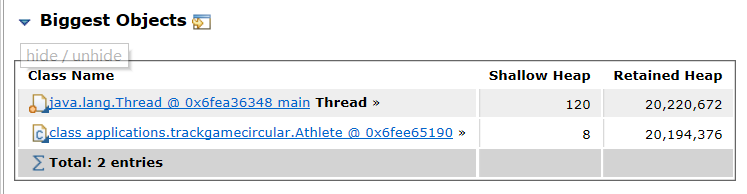
Histogram：

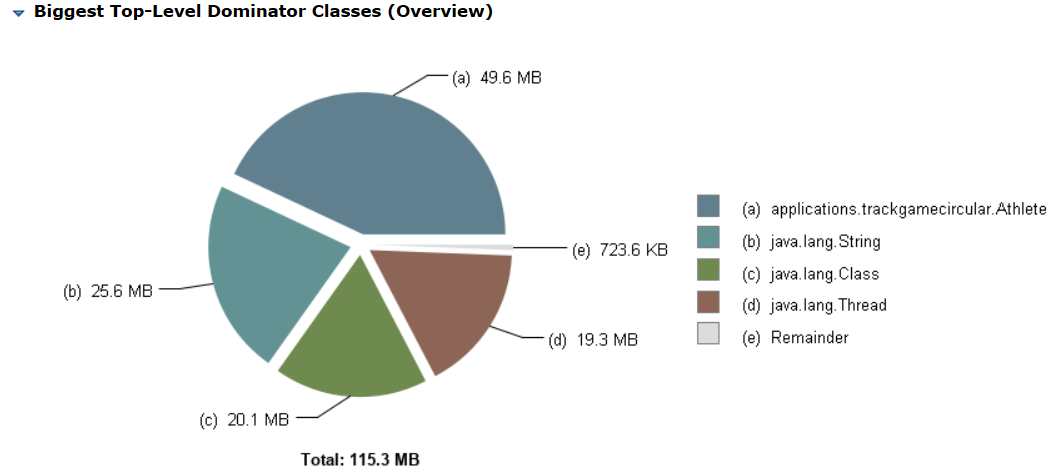


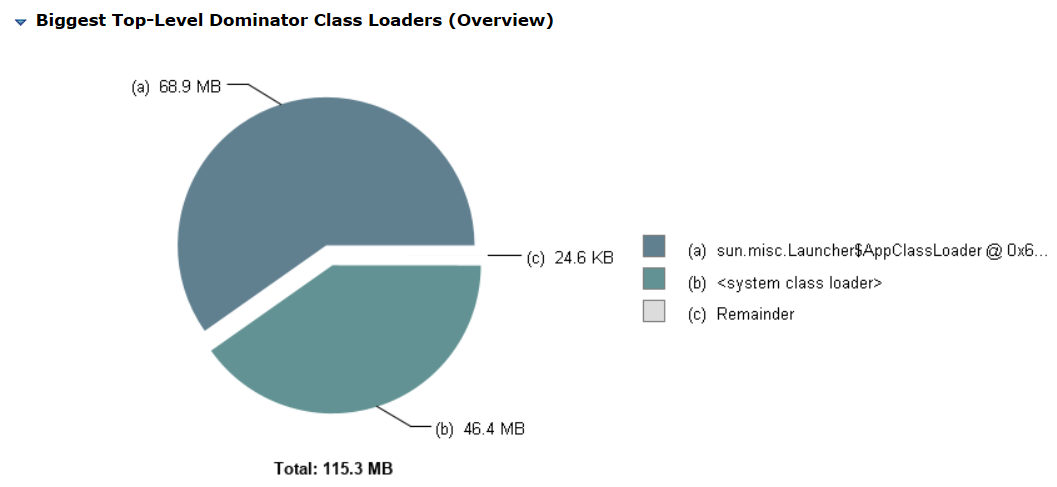


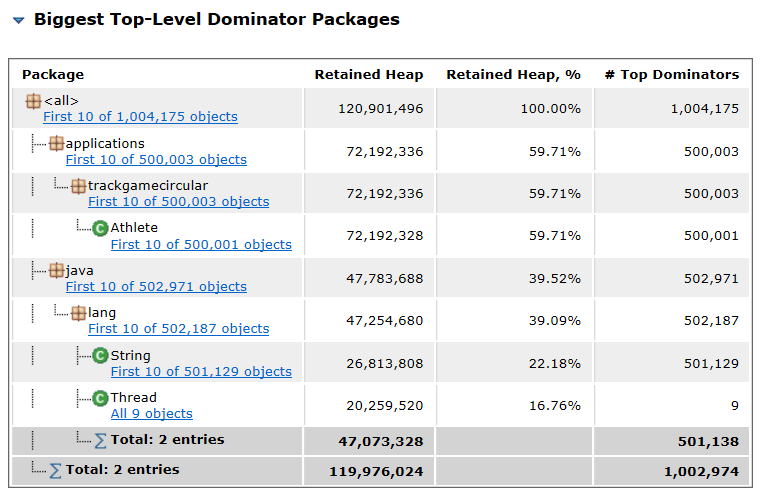
Top consumer：



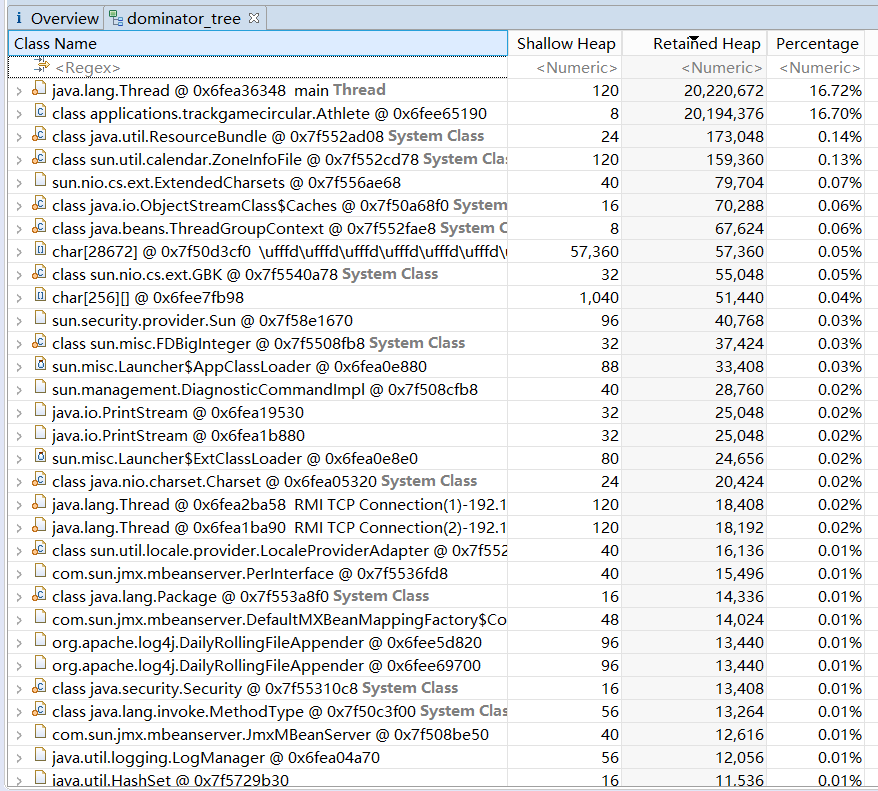


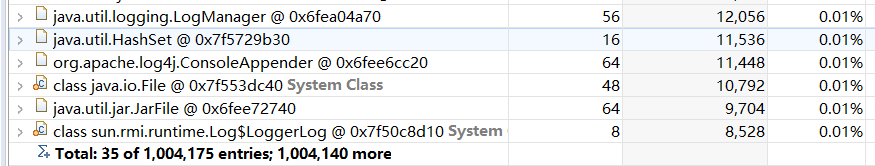




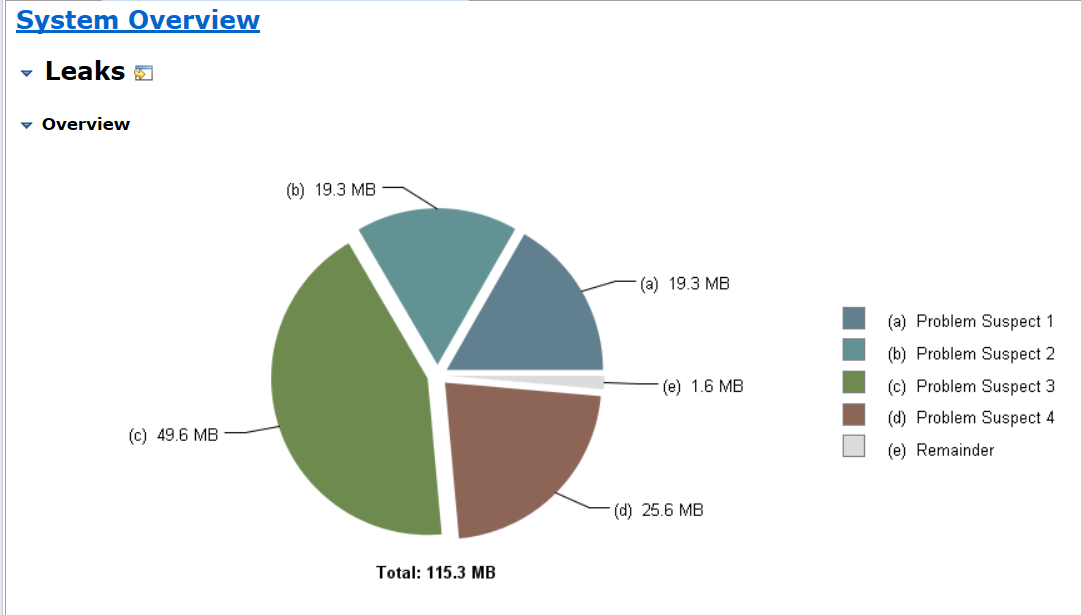


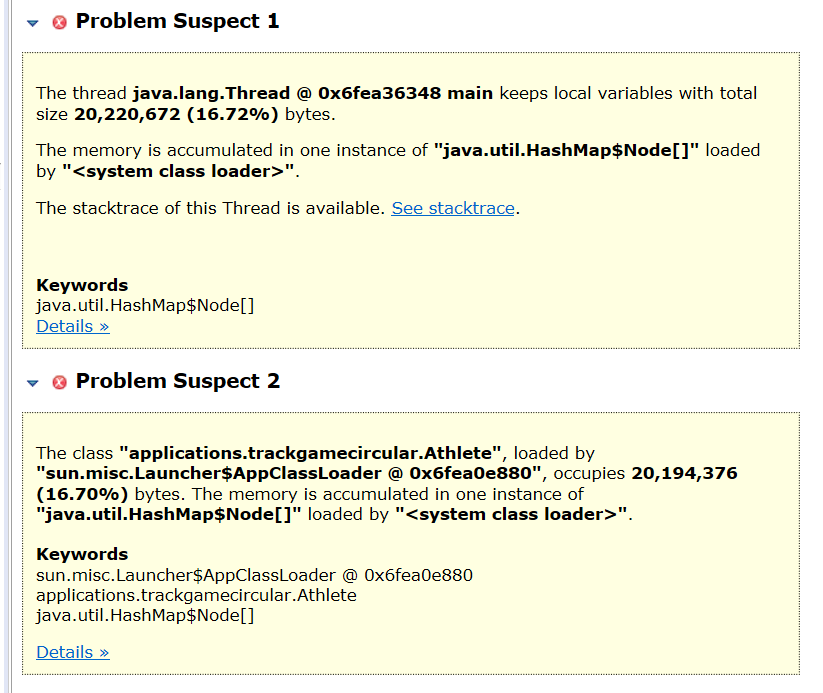
Dominator tree：

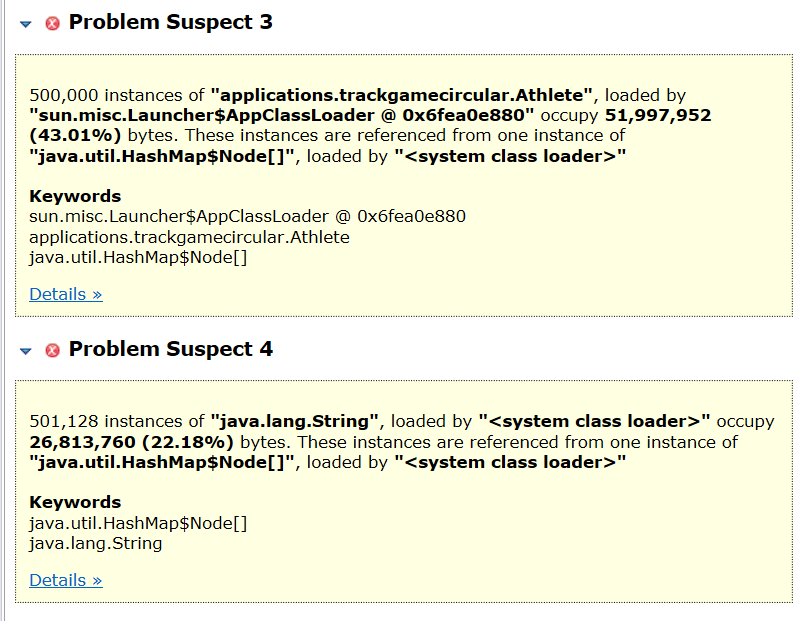




Leak Suspects：





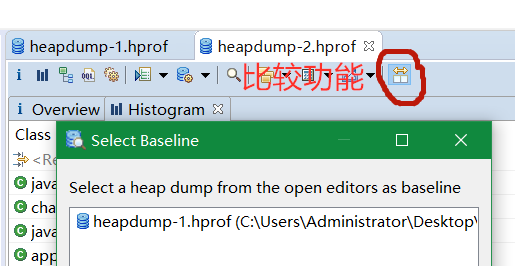


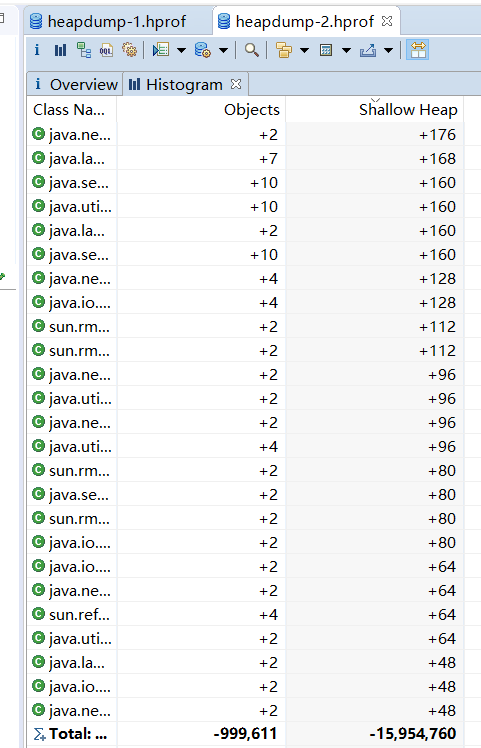
### 发现热点/瓶颈并改进、改进前后的性能对比分析



通过观察Histogram，我发现我申请了500000个Double和Integer对象，但是这个其实是可以通过使用double和int直接避免的，于是修改了构造函数和传递的参数。

重新生成heapdump之后使用MAT的比较功能

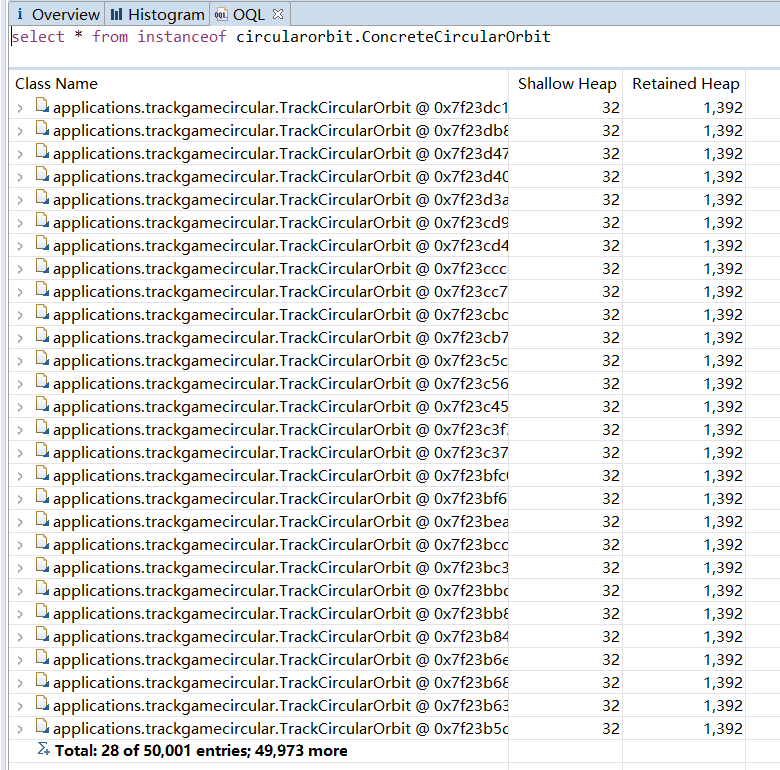




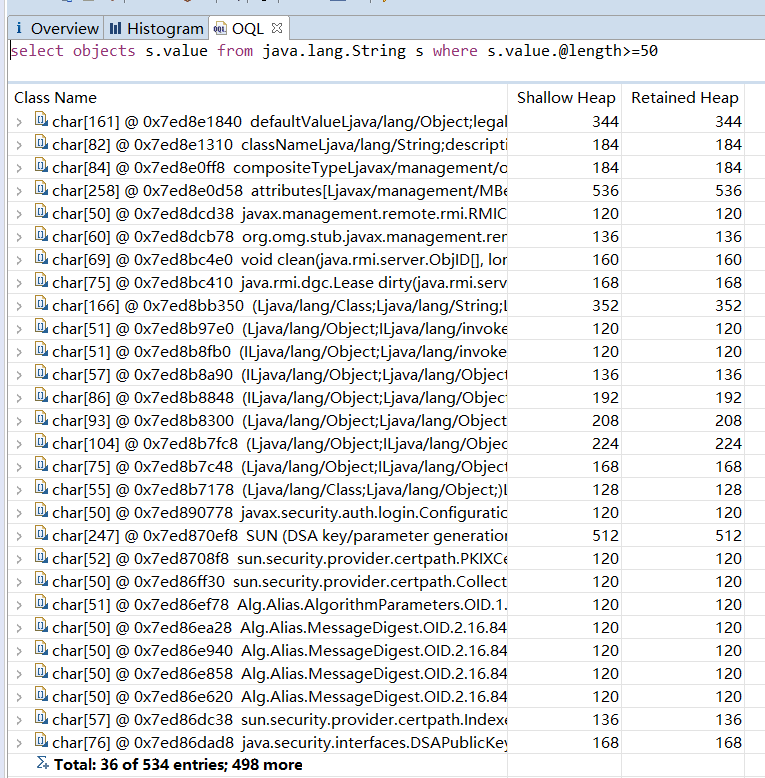
可以发现减少了很多对象。

### 在MAT内使用OQL查询内存导出

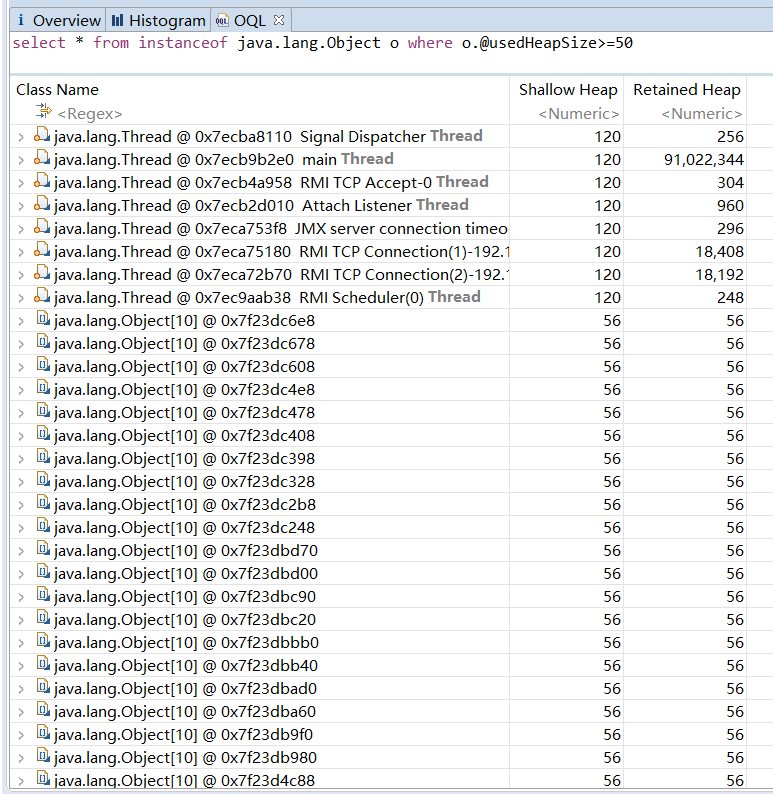
查询TrackCircularOrbit：

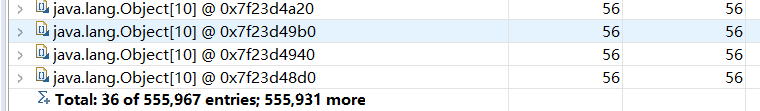


查询长度大于50的字符串对象：

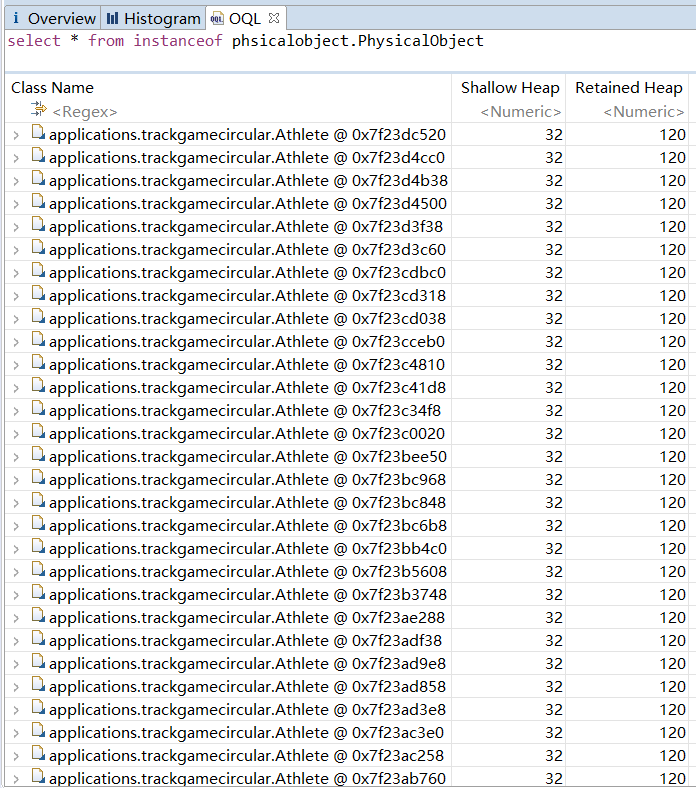


大于特定大小50的任意对象：



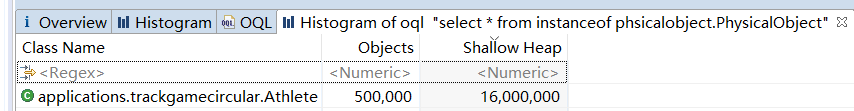


PhysicalObject的所有子类：





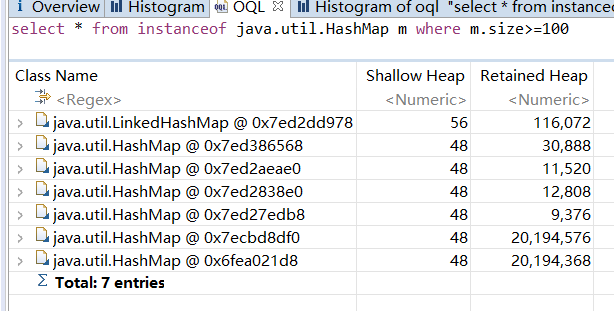
总占用内存：



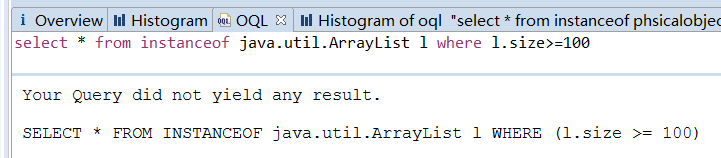
查询下大于100个物体的所有collection

因为没法一次性查所有collection，所以就查了使用的几个

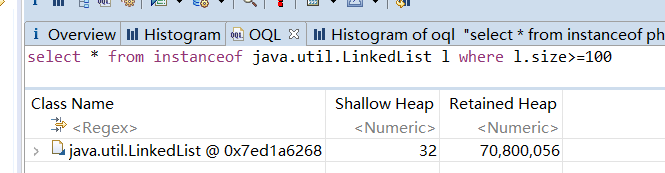
Hashmap：



ArrayList：结果没有



LinkedList：

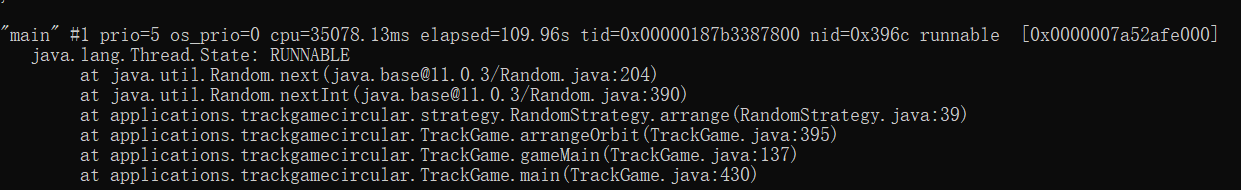


### 观察jstack/jcmd导出程序运行时的调用栈

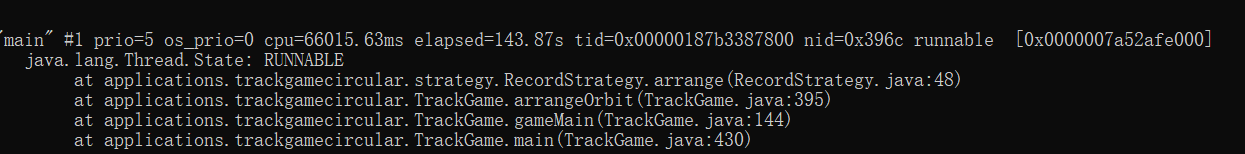
读取文件时：



随机分配赛道：

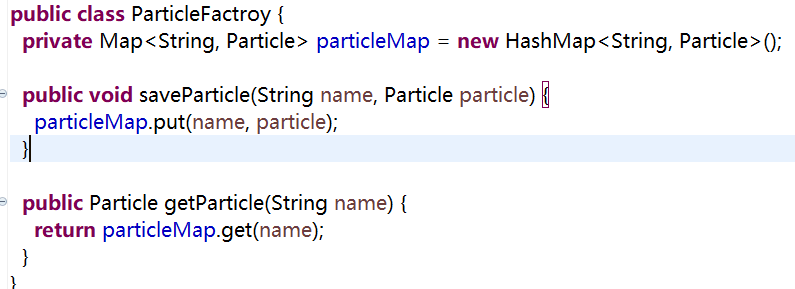


有序分配赛道：



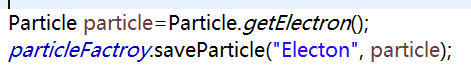
### 使用设计模式进行代码性能优化

##### flyweight设计模式



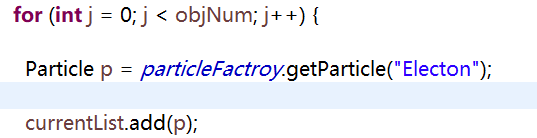
我的电子是基于Particle类实现的，现在实现一个ParticleFactroy，能根据传入不同的名字返回不同的微粒对象（扩展到中子，质子也很好用）。

在使用之前



先save一个Electon到电子的对应。

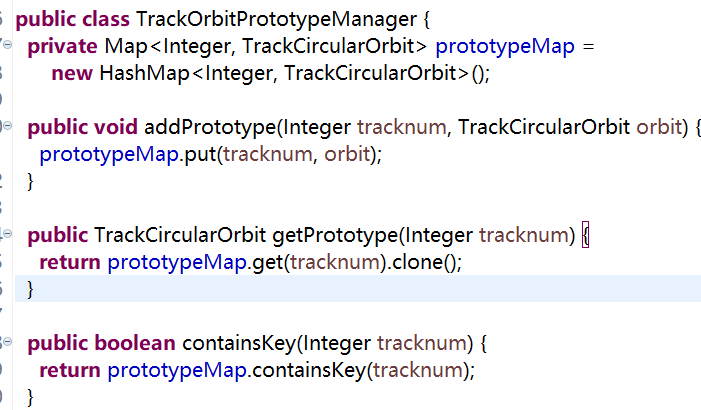
之后每次需要一个电子对象就用particleFactroy.getParticle方法。传入"Electon"。



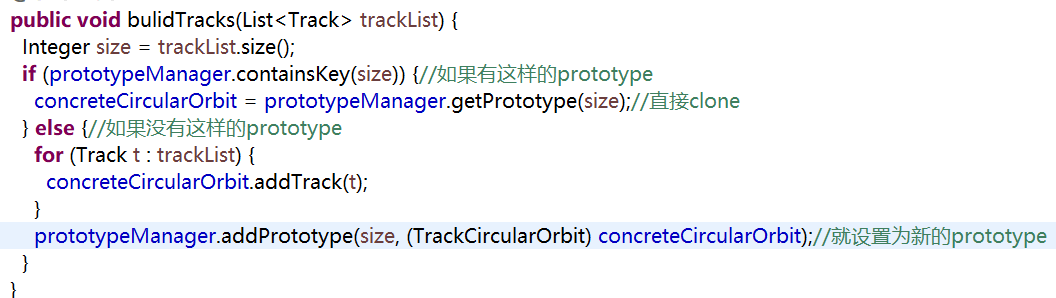
##### prototype模式

因为在安排比赛的过程中会重复声明很多有一样的数目的轨道的Orbit，所以使用prototype设计模式

建立一个新的类管理所有的prototype



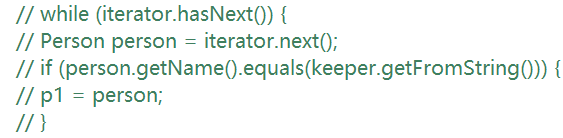
之后每次建立具有一样的轨道数目的Orbit，先检查下有没有这样的prototype，有就直接clone一个，没有就构建好之后设置为prototype



##### 大量使用hashmap

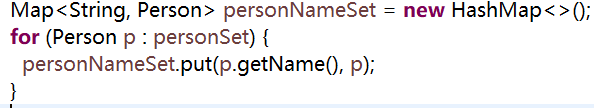
相比于其他的collection查找，hashmap快得不是一星半点，

以前我把person保存在一个list里，每次读取名字都要遍历整个list查找



之后的改进：

建立一个名字->person对象的map，每次只需要get一下，快了很多。

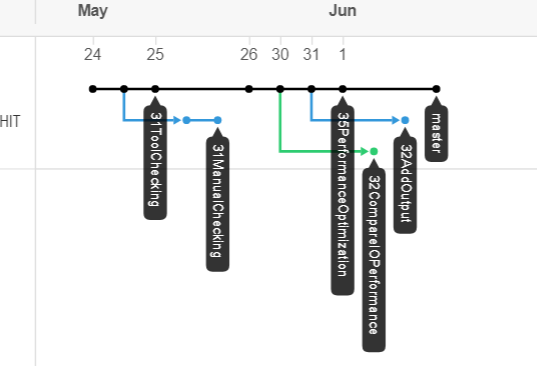


##### 其他的优化细节

包括使用stringbuilder代替“+”，用int替换一些Integer，用double替换Double等等。

## Git仓库结构

注：一开始写io的时候就是按照32CompareIOPerformance的要求来写的所以忘记了32AddOutput，所以这个分支会出现在32CompareIOPerformance的后面是后来补的。但是关系不大。



# 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 |
| 5.24 | 19:00-23:00 | 3.1 | 完成 |
| 5.25 | 8:00-23:00 | 3.2 | 完成 |
| 5.26 | 8:00-23:00 | 3.3 | 未完成，visualvm无法启动 |
| 5.27 | 19:00-23:00 | 3.3 | 完成 |
| 5.31 | 15:00-23:00 | 3.4 | 未完成 |
| 6.1 | 8:00-23:00 | 3.4+3.5 | 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 |
| Visualvm用不了了 | 重装jdk |
|  |  |
|  |  |

# 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？

综合考虑。看情况。

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？

会

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？

暂时还不能，复习的时候再体会一下

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？

计算机系统好像没有分区分策略。

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？

手动麻烦但是上限高，自动简单但是上限低。

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？

有，垃圾回收时间从220+ms降到了110+ms.

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？

有的工具有些老旧甚至不用了希望更新一下，一直在跟进的工具还是很好的。

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？

希望我有。

1. 关于本实验的工作量、难度、deadline。

比较麻烦，很多工具没用过需要自学，而且有的工具不知道出什么bug（visulavm），也查不到解决方法。

1. 到目前为止，你对《软件构造》课程的意见与建议。

很花时间。收获不少。