

**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 | 李震宇 |
| 学号 | 1170300110 |
| 班号 | 1703001 |
| 电子邮件 | 1137859144@qq.com |
| 手机号码 | 18800419432 |

**目录**

[1 实验目标概述 1](#_Toc10383430)

[2 实验环境配置 1](#_Toc10383431)

[3 实验过程 1](#_Toc10383432)

[3.1 Static Program Analysis 1](#_Toc10383433)

[3.1.1 人工代码走查（walkthrough） 1](#_Toc10383434)

[3.1.1.1 某些函数之前没有写注释 1](#_Toc10383435)

[3.1.1.2 形如/\*\*……的注释第一行没有写内容。 2](#_Toc10383436)

[3.1.1.3 形如/\*\*……的注释第一行写了内容没有写句号结束。 2](#_Toc10383437)

[3.1.1.4 每行的缩进问题。 2](#_Toc10383438)

[3.1.1.5 空格的添加问题。 2](#_Toc10383439)

[3.1.1.6 注释中有tab的存在。 3](#_Toc10383440)

[3.1.2 使用CheckStyle和SpotBugs进行静态代码分析 3](#_Toc10383441)

[3.1.2.1 行间距问题 3](#_Toc10383442)

[3.1.2.2 import字典序问题 3](#_Toc10383443)

[3.1.2.3 参数注释不全 3](#_Toc10383444)

[3.1.2.4 不允许使用unicode编码 3](#_Toc10383445)

[3.1.2.5 变量声明距离第一次使用变量距离过远 4](#_Toc10383446)

[3.1.2.6 写了equals方法没写hashcode方法 4](#_Toc10383447)

[3.1.2.7 equals方法没有考虑到传入对象为null的情况 4](#_Toc10383448)

[3.2 Java I/O Optimization 5](#_Toc10383449)

[3.2.1 多种I/O实现方式 5](#_Toc10383450)

[3.2.1.1 Input读入文件设计： 5](#_Toc10383451)

[3.2.1.2 Output写文件设计： 7](#_Toc10383452)

[3.2.1.3 I/O策略间的切换 8](#_Toc10383453)

[3.2.2 多种I/O实现方式的效率对比分析 9](#_Toc10383454)

[3.2.2.1 读收集语法文件时间的方式： 9](#_Toc10383455)

[3.2.2.2 写文件收集语法文件时间方式： 10](#_Toc10383456)

[3.2.2.3 表格对比 10](#_Toc10383457)

[3.2.2.4 图形对比 11](#_Toc10383458)

[3.3 Java Memory Management and Garbage Collection (GC) 11](#_Toc10383459)

[3.3.1 使用-verbose:gc参数 11](#_Toc10383460)

[3.3.2 用jstat命令行工具的-gc和-gcutil参数 12](#_Toc10383461)

[3.3.3 使用jmap -heap命令行工具 13](#_Toc10383462)

[3.3.4 使用jmap -clstats命令行工具 14](#_Toc10383463)

[3.3.5 使用JMC/JFR、jconsole或VisualVM工具 15](#_Toc10383464)

[3.3.6 分析垃圾回收过程 16](#_Toc10383465)

[3.3.7 配置JVM参数并发现优化的参数配置 17](#_Toc10383466)

[3.4 Dynamic Program Profiling 19](#_Toc10383467)

[3.4.1 使用JMC或VisualVM进行CPU Profiling 19](#_Toc10383468)

[3.4.2 使用VisualVM进行Memory profiling 21](#_Toc10383469)

[3.5 Memory Dump Analysis and Performance Optimization 21](#_Toc10383470)

[3.5.1 内存导出 21](#_Toc10383471)

[3.5.2 使用MAT分析内存导出文件 22](#_Toc10383472)

[3.5.3 发现热点/瓶颈并改进、改进前后的性能对比分析 24](#_Toc10383473)

[3.5.4 在MAT内使用OQL查询内存导出 26](#_Toc10383474)

[3.5.5 观察jstack/jcmd导出程序运行时的调用栈 28](#_Toc10383475)

[3.5.6 使用设计模式进行代码性能优化 29](#_Toc10383476)

[3.6 Git仓库结构 30](#_Toc10383477)

[4 实验进度记录 30](#_Toc10383478)

[5 实验过程中遇到的困难与解决途径 30](#_Toc10383479)

[6 实验过程中收获的经验、教训、感想 30](#_Toc10383480)

[6.1 实验过程中收获的经验和教训 30](#_Toc10383481)

[6.2 针对以下方面的感受 31](#_Toc10383482)

# 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

java + eclipse + spotBugs + checkstyle

https://github.com/ComputerScienceHIT/Lab5-1170300110.git

# 实验过程

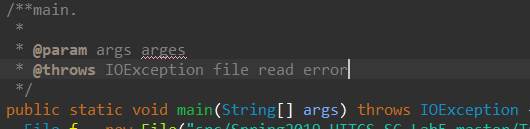
## Static Program Analysis

### 人工代码走查（walkthrough）

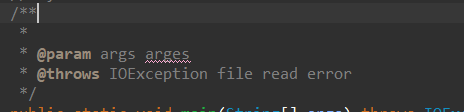
#### 某些函数之前没有写注释



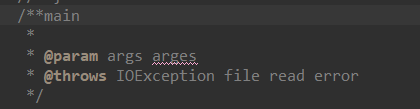
改成：



#### 形如/\*\*……的注释第一行没有写内容。



#### 形如/\*\*……的注释第一行写了内容没有写句号结束。



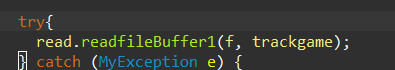
#### 每行的缩进问题。



改成：



#### 空格的添加问题。



改为：



#### 注释中有tab的存在。

将tab用空格替换了。之后tab显示不出来了就不截图了。

### 使用CheckStyle和SpotBugs进行静态代码分析

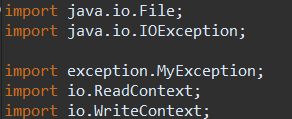
CheckStyle发现的问题：

#### 行间距问题

比如package和import之间需要隔一行。

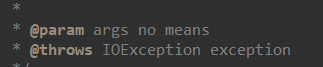


#### import字典序问题



它期望import的顺序按字典序，但是eclipse中包的顺序是编译器自动调整的。我认为它checkStyle时这个要求有些问题。

#### 参数注释不全



@param或@throws后的每一项都需要解释，有缺项。

#### 不允许使用unicode编码



因为设计GUI时使用的时windowbuilder它会自动将汉字转换为unicode编码，所以这里需要依次将unicode编码再转回汉字编码。

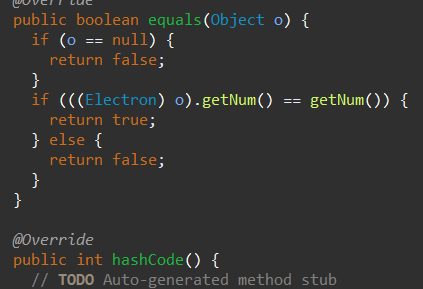
#### 变量声明距离第一次使用变量距离过远



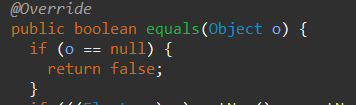
超过四行就会希望用final修饰。

spotBugs发现的问题：

#### 写了equals方法没写hashcode方法



#### equals方法没有考虑到传入对象为null的情况



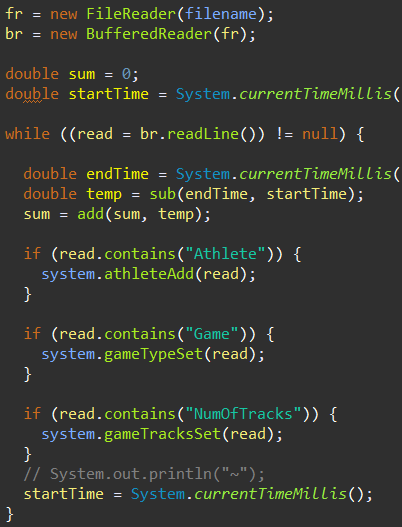
## Java I/O Optimization

### 多种I/O实现方式

#### Input读入文件设计：

##### Buffer

使用BufferReader读入文件：



##### Scanner

使用Scanner读入文件：



##### Stream

使用Stream读入文件：



#### Output写文件设计：

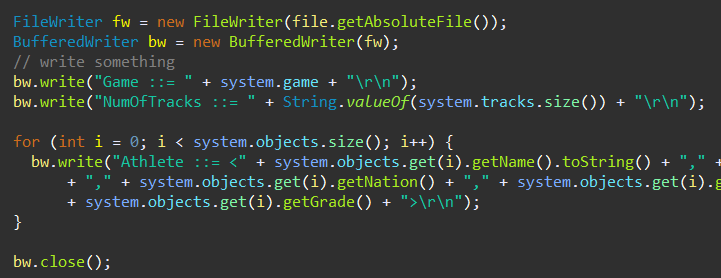
##### FileWriter

使用FileWriter写文件：



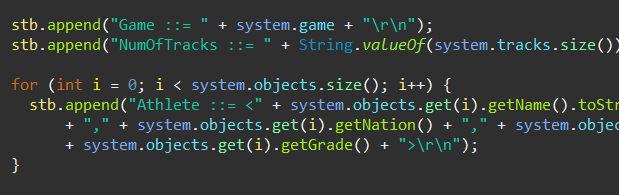
##### BufferWriter

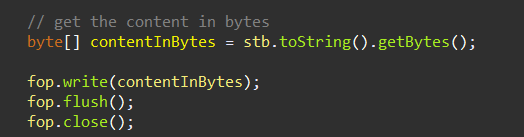
使用BufferWriter写文件：



##### Stream

使用Stream写文件：

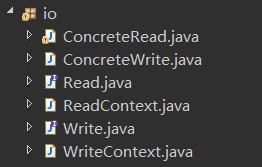




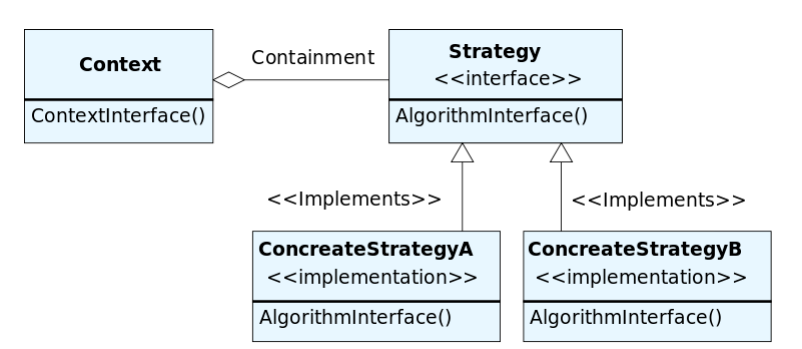
#### I/O策略间的切换

我使用了Strategy设计模式的变式。

目录：



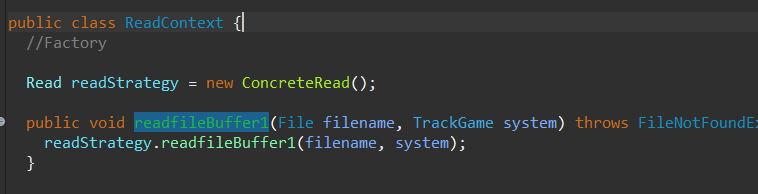
画出类图：



我把所有的ConcreateStrategyABC都写到了一起。

在Context中选择使用ConcreteInterface中的不同的读写文件方法。

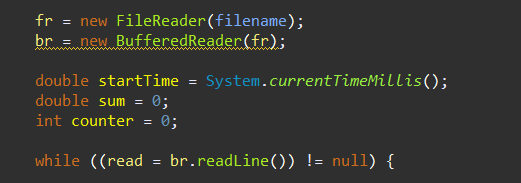
Context中的部分内容截图：



### 多种I/O实现方式的效率对比分析

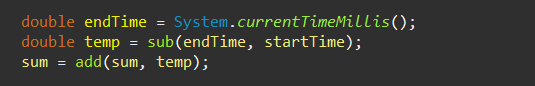
#### 读收集语法文件时间的方式：

在读取一行文件之前记录startTime

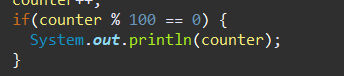


在读取之后记录endTime

求endTime和startTime之差，然后加到sum上。



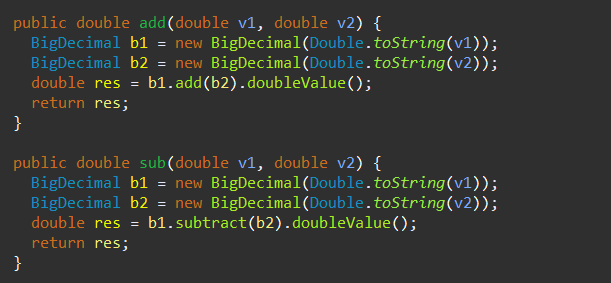
每个一百行打印一下，确保程序还在运行：



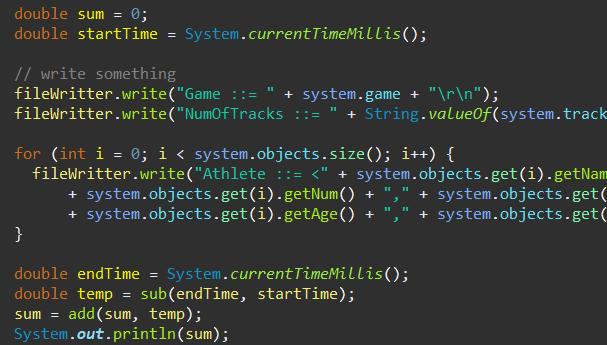
因为是小数计算，需要保证精确性，额外写了两个函数：

分别对应小数的加和减：

使用BigDecimal类来辅助操作。

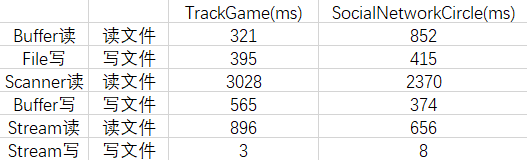


#### 写文件收集语法文件时间方式：

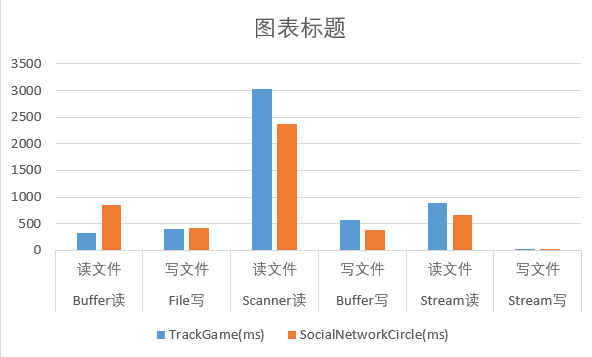


在写入的前后设置时间因子，计算相减的时间差打印出来即可。

#### 表格对比



#### 图形对比



分析：

Scanner是读中最慢的毋庸置疑。

对于Buffer读和Stream读，其速度差距因文件而异的感觉。

对写文件来说File写和Buffer写差别不大，而Stream写入速度极快，但是前提是构造char数组需要了更多的时间。

## Java Memory Management and Garbage Collection (GC)

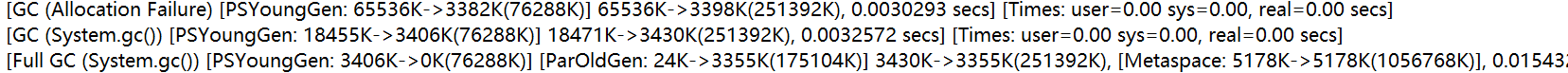
### 使用-verbose:gc参数

参数配置：





gc.log文件：



1.Minor GC发生频率比较低，20万以上的数据读入产生了10次左右的Minor GC，Full GC发生频率极低，甚至我没捕捉到，使用了System.gc()方法产生Full GC。

2.花费时间：

Minor：

Full：

显然Minor GC要远快于Full GC。

3.堆大小的变化：

Minor：

Full: 

极大多数都是Minor GC，这种垃圾处理方式发生在新生区满时。并没有自然的发生Full GC，我已经执行了2万条改变轨道系统的操作，也没有发生GC。

Full GC触发条件：

（1）调用System.gc时，系统建议执行Full GC，但是不必然执行

（2）老年代空间不足

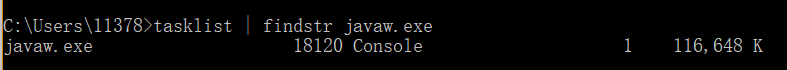
（3）方法去空间不足

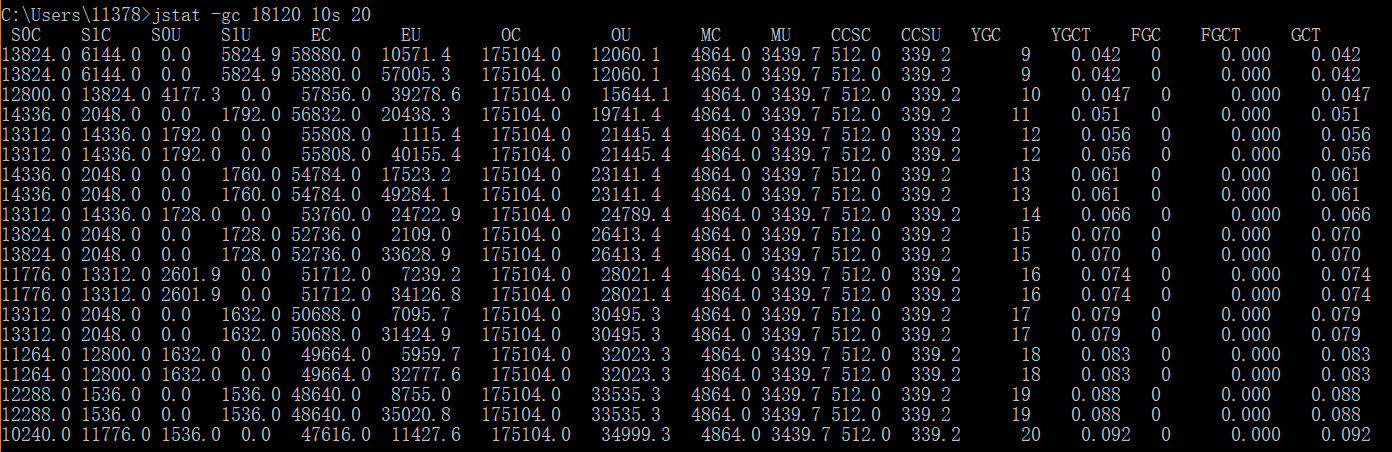
（4）通过Minor GC后进入老年代的平均大小大于老年代的可用内存

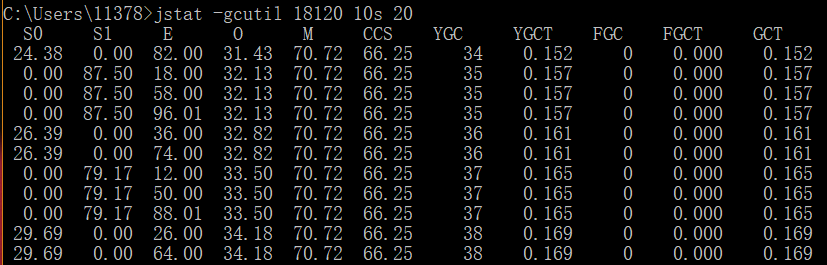
（5）由Eden区、From Space区向To Space区复制时，对象大小大于To Space可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小

因为没有发生，所以就是后四种情况没有出现。

### 用jstat命令行工具的-gc和-gcutil参数







S0C：年轻代中第一个存活区的大小

S1C：年轻代中第二个存活区的大小

S0U：年轻代中第一个存活区已使用的空间 (字节)

S1U：年轻代中第二个存活区已使用的空间 (字节)

EC:   Edem区大小

EU:   年轻代中Edem区已使用的空间 (字节)

OC:  老年代大小

OU:  老年代已使用的空间 (字节)

PC:   持久代大小

PU:　持久代已使用的空间 (字节)

YGC: 从应用程序启动到采样时young gc的次数

YGCT: 从应用程序启动到采样时young gc的所用的时间（s）

FGC:  从应用程序启动到采样时full gc的次数

FGCT: 从应用程序启动到采样时full gc的所用的时间

GCT: 从应用程序启动到采样时整个gc所用的时间

S0  — Heap上的 Survivor space 0 区已使用空间的百分比

S1  — Heap上的 Survivor space 1 区已使用空间的百分比

E   — Heap上的 Eden space 区已使用空间的百分比

O   — Heap上的 Old space 区已使用空间的百分比

P   — Perm space 区已使用空间的百分比

YGC — 从应用程序启动到采样时发生 Young GC 的次数

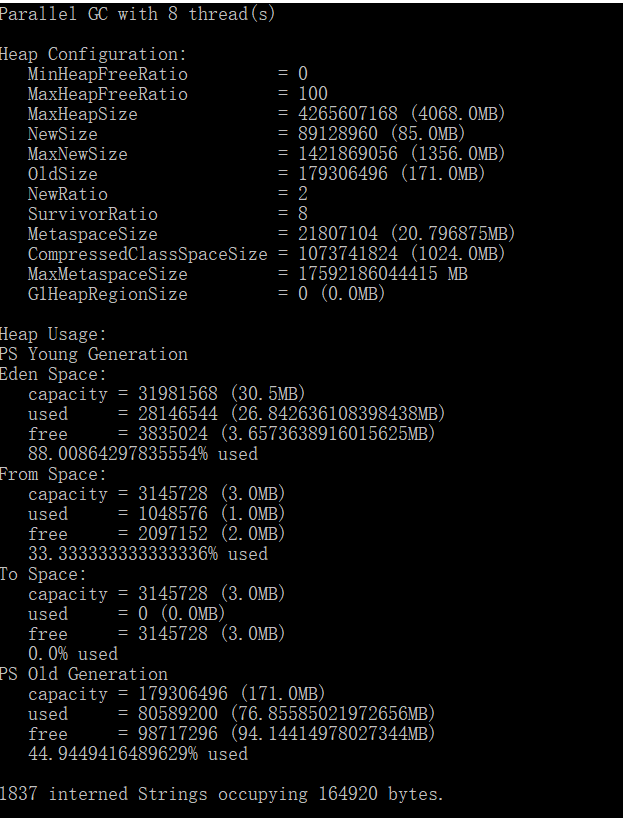
YGCT– 从应用程序启动到采样时 Young GC 所用的时间(单位秒)

FGC — 从应用程序启动到采样时发生 Full GC 的次数

FGCT– 从应用程序启动到采样时 Full GC 所用的时间(单位秒)

GCT — 从应用程序启动到采样时用于垃圾回收的总时间(单位秒)

### 使用jmap -heap命令行工具



#-XX:MinHeapFreeRatio设置JVM堆最小空闲比率

#-XX:MaxHeapFreeRatio设置JVM堆最大空闲比率

#-XX:MaxHeapSize=设置JVM堆的最大大小

#-XX:NewSize=设置JVM堆的‘新生代’的默认大小

#-XX:MaxNewSize=设置JVM堆的‘新生代’的最大大小

#-XX:OldSize=设置JVM堆的‘老生代’的大小

#-XX:NewRatio=:‘新生代’和‘老生代’的大小比率

#-XX:SurvivorRatio=设置年轻代中Eden区与Survivor区的大小比值

#-XX:PermSize=<value>:设置JVM堆的‘持久代’的初始大小

#-XX:MaxPermSize=<value>:设置JVM堆的‘持久代’的最大大小

Heap Usage:

New Generation (Eden + 1 Survivor Space): #新生代区内存分布，包含伊甸园区+1个Survivor区

Eden Space: #Eden区内存分布

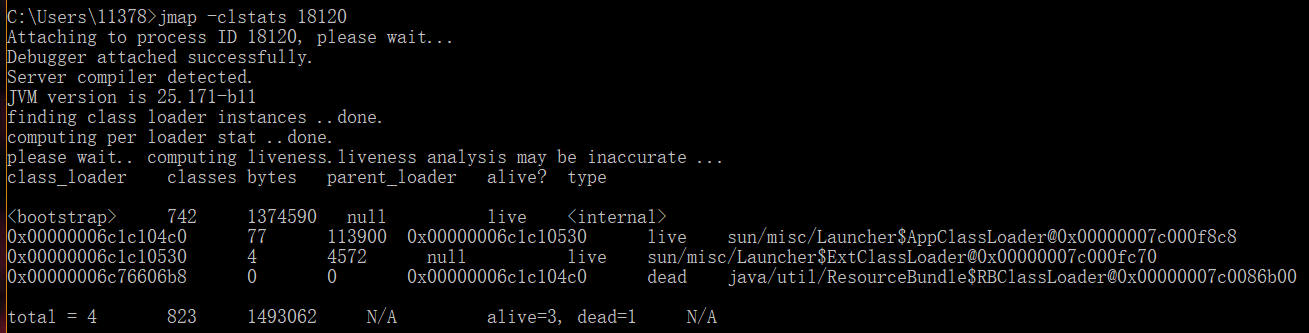
From Space: #其中一个Survivor区的内存分布

To Space: #另一个Survivor区的内存分布

tenured generation: #当前的Old区内存分布

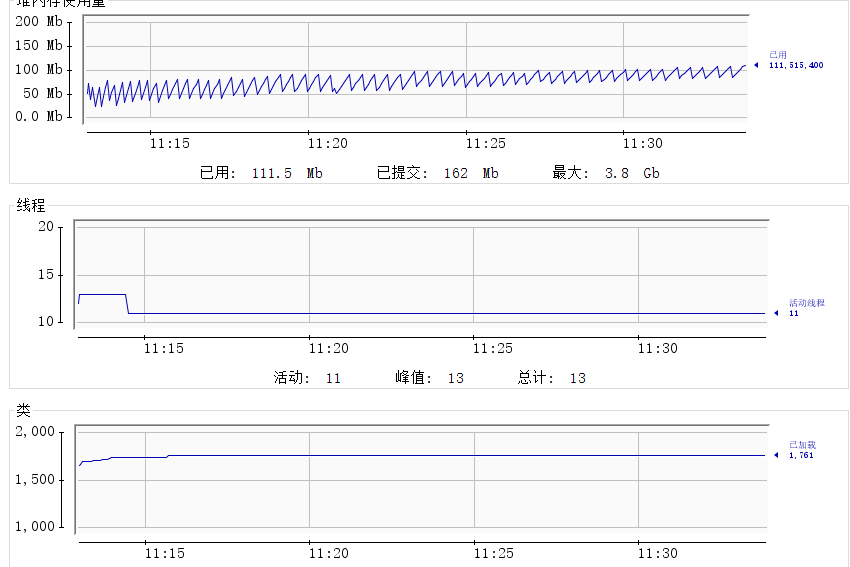
Perm Generation: #当前的 “持久代” 内存分布

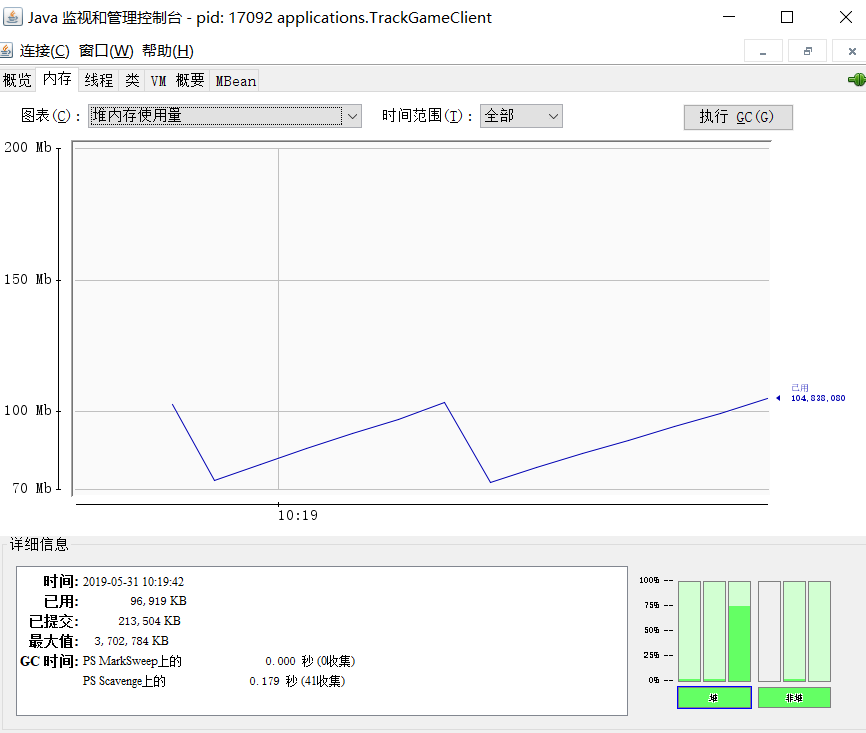
### 使用jmap -clstats命令行工具

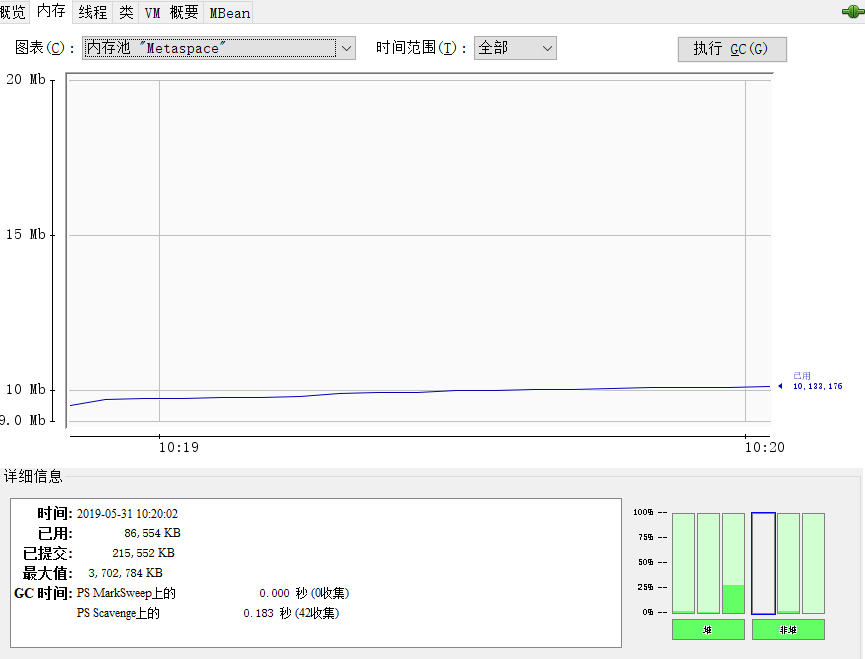


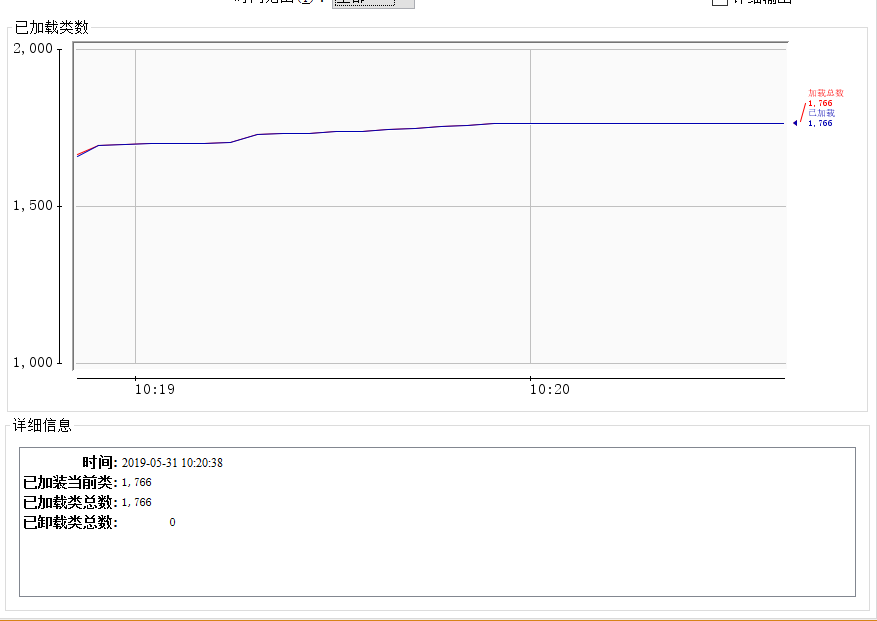
命令行工具查看 class loader 的统计信息，即程序 执行期间装载的 class 和 method 相关信息。（注：在 JDK8 以前的版本中， 使用的指令是 jmap -permstat） 所以把下面那个jmap -permstat命令行工具删除。

### 使用JMC/JFR、jconsole或VisualVM工具

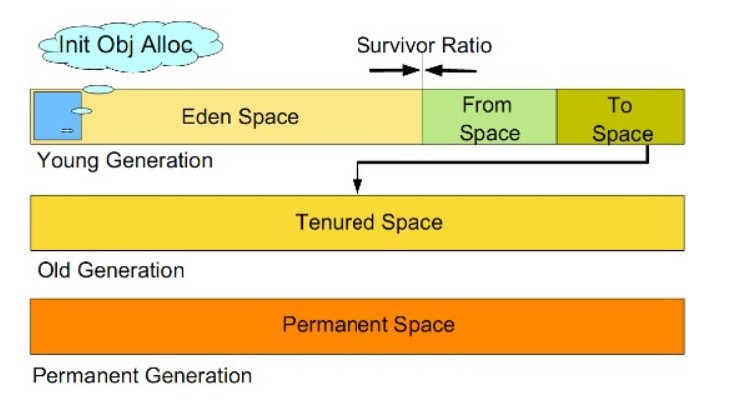








### 分析垃圾回收过程



分代的垃圾回收策略，是基于这样一个事实：**不同的对象的生命周期是不一样的**。因此，不同生命周期的对象可以采取不同的回收算法，以便提高回收效率。

**年轻代（Young Generation）**

1.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.新生代内存按照8:1:1的比例分为一个eden区和两个survivor(survivor0,survivor1)区。一个Eden区，两个 Survivor区(一般而言)。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survivor0区，然后清空eden区，当这个survivor0区也存放满了时，则将eden区和survivor0区存活对象复制到另一个survivor1区，然后清空eden和这个survivor0区，此时survivor0区是空的，然后将survivor0区和survivor1区交换，即保持survivor1区为空， 如此往复。

3.当survivor1区不足以存放 eden和survivor0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

4.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)

**年老代（Old Generation）**

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.内存比新生代也大很多(大概比例是1:2)，当老年代内存满时触发Major GC即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。

**持久代（Permanent Generation）**

用于存放静态文件，如Java类、方法等。持久代对垃圾回收没有显著影响，但是有些应用可能动态生成或者调用一些class，例如Hibernate 等，在这种时候需要设置一个比较大的持久代空间来存放这些运行过程中新增的类。

### 配置JVM参数并发现优化的参数配置

**java -Xmx3550m -Xms3550m -Xmn2g –Xss128k**

-Xmx3550m： 设置 JVM 最大可用内存为 3550M。

-Xms3550m： 设置 JVM 初始内存为 3550m。此值可以设置与-Xmx 相同，以避免每次垃圾

回收完成后 JVM 重新分配内存。

-Xmn2g： 设置年轻代大小为 2G。整个堆大小=年轻代大小 + 年老代大小 + 持久代大小。

持久代一般固定大小为 64m，所以增大年轻代后，将会减小年老代大小。此值对系统性能

影响较大， Sun 官方推荐配置为整个堆的 3/8。

-Xss128k： 设置每个线程的堆栈大小。 JDK5.0 以后每个线程堆栈大小为 1M，以前每个线

程堆栈大小为 256K。更具应用的线程所需内存大小进行调整。在相同物理内存下，减小这

个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，

经验值在 3000~5000 左右。

**java -Xmx3550m -Xms3550m -Xss128k -XX:NewRatio=4 -XX:SurvivorRatio=4 -XX:MaxPermSize=16m -XX:MaxTenuringThreshold=0**

-XX:NewRatio=4: 设置年轻代（包括 Eden 和两个 Survivor 区）与年老代的比值（除去持久代）。设置为 4，则年轻代与年老代所占比值为 1： 4，年轻代占整个堆栈的 1/5

-XX:SurvivorRatio=4： 设置年轻代中 Eden 区与 Survivor 区的大小比值。设置为 4，则两个 Survivor 区与一个 Eden 区的比值为 2:4，一个 Survivor 区占整个年轻代的 1/6

-XX:MaxPermSize=16m: 设置持久代大小为 16m。

-XX:MaxTenuringThreshold=0： 设置垃圾最大年龄。如果设置为 0 的话，则年轻代对象

不经过 Survivor 区，直接进入年老代。对于年老代比较多的应用，可以提高效率。如果将

此值设置为一个较大值，则年轻代对象会在 Survivor 区进行多次复制，这样可以增加对象

再年轻代的存活时间，增加在年轻代即被回收的概论

**java -Xmx3800m -Xms3800m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:ParallelGCThreads=20**

-XX:+UseParallelGC： 选择垃圾收集器为并行收集器。 此配置仅对年轻代有效。即上述配

置下，年轻代使用并发收集，而年老代仍旧使用串行收集。

-XX:ParallelGCThreads=20： 配置并行收集器的线程数，即：同时多少个线程一起进行垃

圾回收。此值最好配置与处理器数目相等。

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:ParallelGCThreads=20 -XX:+UseParallelOldGC

-XX:+UseParallelOldGC： 配置年老代垃圾收集方式为并行收集。 JDK6.0 支持对年老代并

行收集。

**java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:MaxGCPauseMillis=100**

-XX:MaxGCPauseMillis=100: 设置每次年轻代垃圾回收的最长时间，如果无法满足此时间，

JVM 会自动调整年轻代大小，以满足此值。

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:MaxGCPauseMillis=100 -XX:+UseAdaptiveSizePolicy

-XX:+UseAdaptiveSizePolicy：设置此选项后，并行收集器会自动选择年轻代区大小和相

应的 Survivor 区比例，以达到目标系统规定的最低相应时间或者收集频率等，此值建议使

用并行收集器时，一直打开。

**java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:ParallelGCThreads=20 -XX:+UseConcMarkSweepGC -XX:+UseParNewGC**

-XX:+UseConcMarkSweepGC： 设置年老代为并发收集。测试中配置这个以后，-XX:NewRatio=4 的配置失效了，原因不明。所以，此时年轻代大小最好用-Xmn 设置。

-XX:NewRatio=4 的配置失效了，原因不明。所以，此时年轻代大小最好用-Xmn 设置。

-XX:+UseParNewGC: 设置年轻代为并行收集。可与 CMS 收集同时使用。 JDK5.0 以上，

JVM 会根据系统配置自行设置，所以无需再设置此值。

**java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseConcMarkSweepGC -XX:CMSFullGCsBeforeCompaction=5 -XX:+UseCMSCompactAtFullCollection**

-XX:CMSFullGCsBeforeCompaction： 由于并发收集器不对内存空间进行压缩、整理，所

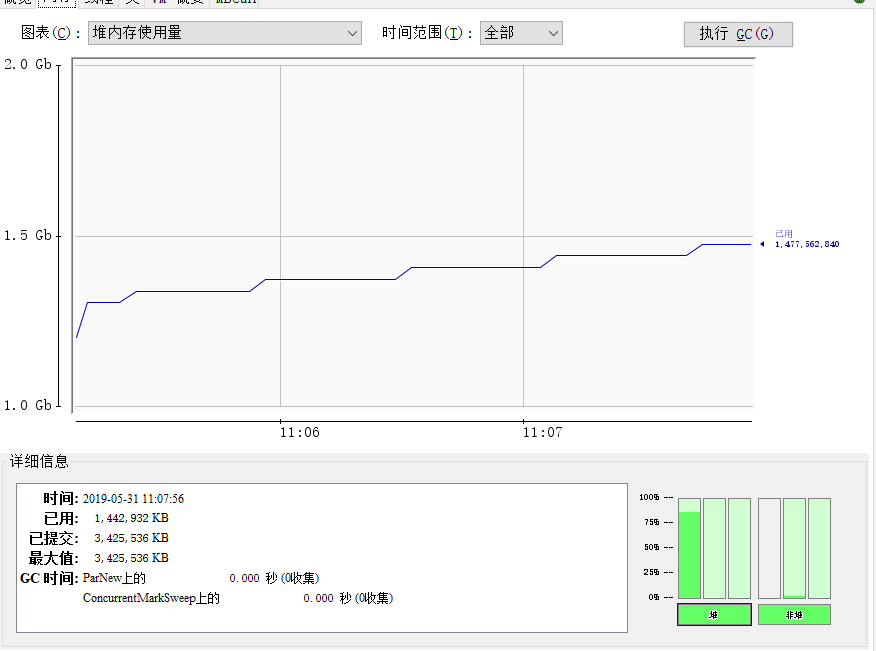
以运行一段时间以后会产生“碎片”，使得运行效率降低。此值设置运行多少次 GC 以后对内存空间进行压缩、整理。

-XX:+UseCMSCompactAtFullCollection： 打开对年老代的压缩。可能会影响性能，但是

可以消除碎片

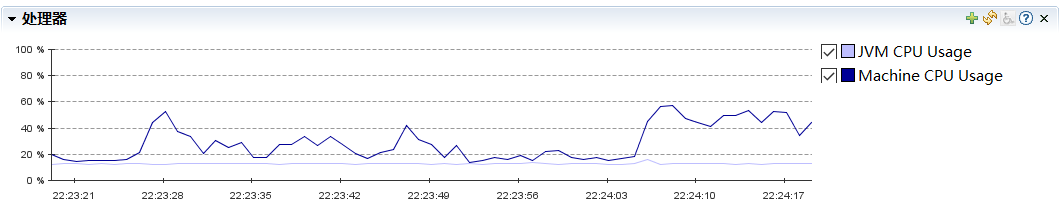
选择使用：java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseConcMarkSweepGC -XX:CMSFullGCsBeforeCompaction=5 -XX:+UseCMSCompactAtFullCollection

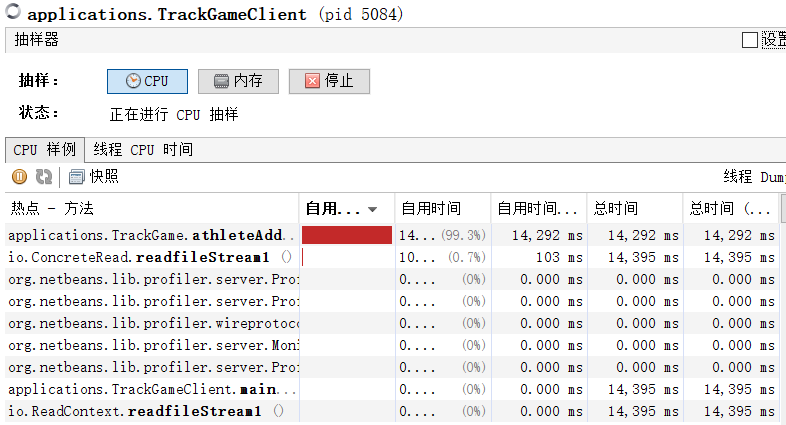
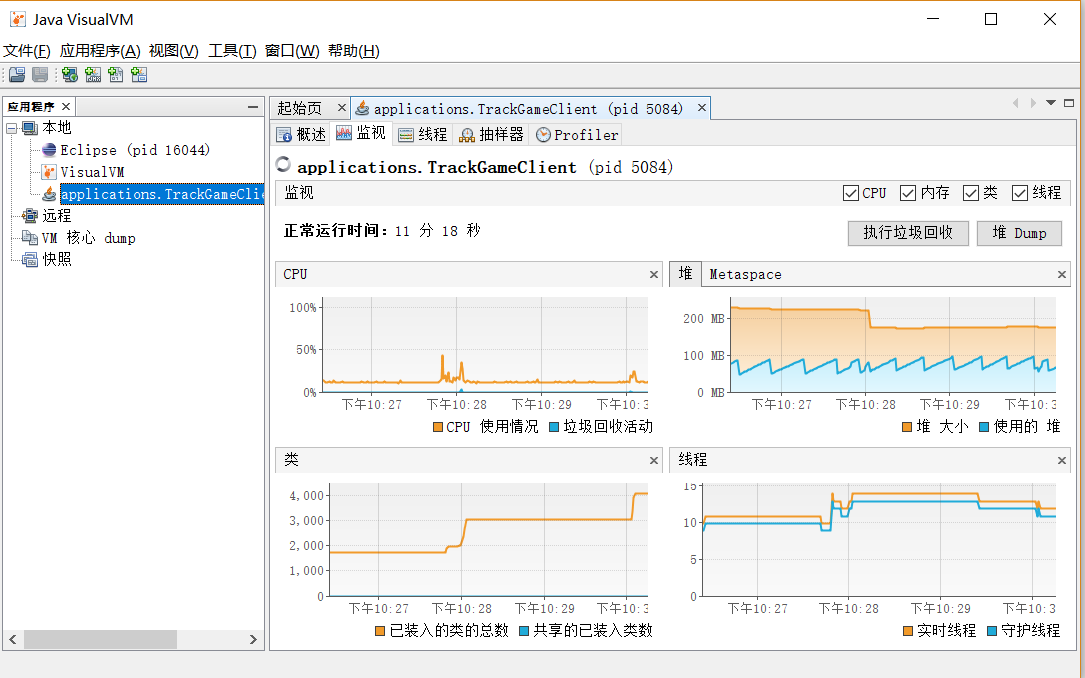
效果：



## Dynamic Program Profiling

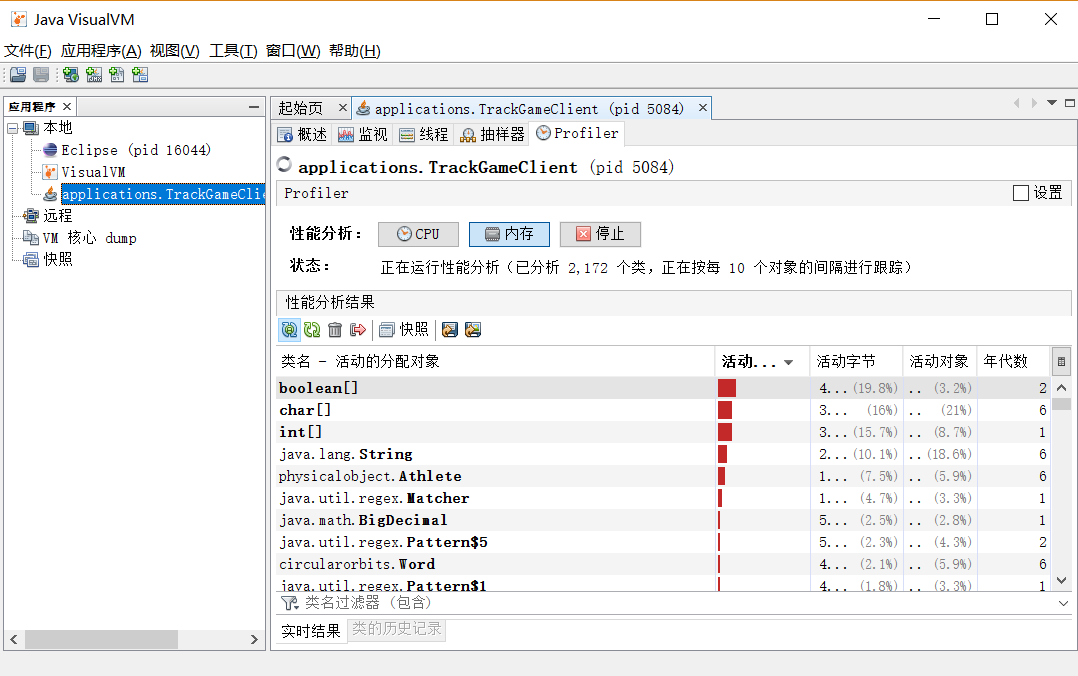
### 使用JMC或VisualVM进行CPU Profiling





比较正常的，因为正在读入文件构造轨道所以athleteAdd方法会被多次调用。

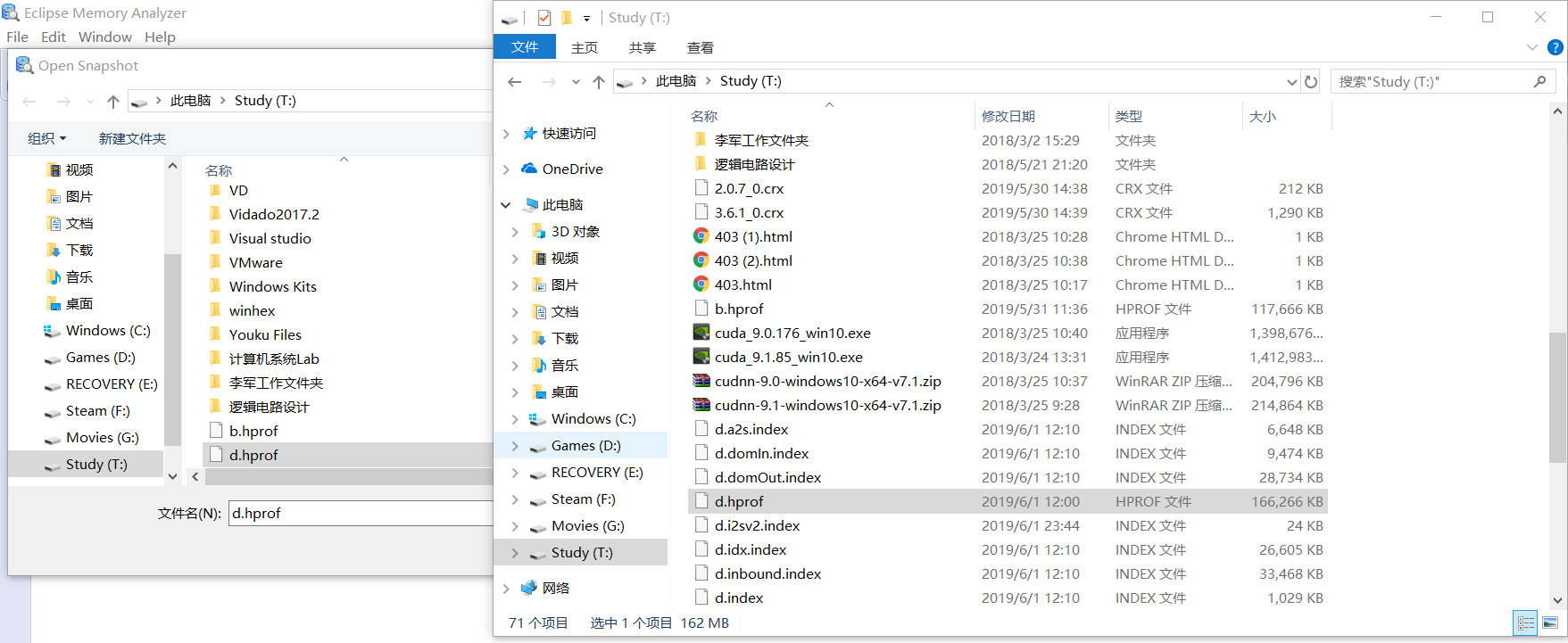
### 使用VisualVM进行Memory profiling



正在读入文件，所以是比较正常的。

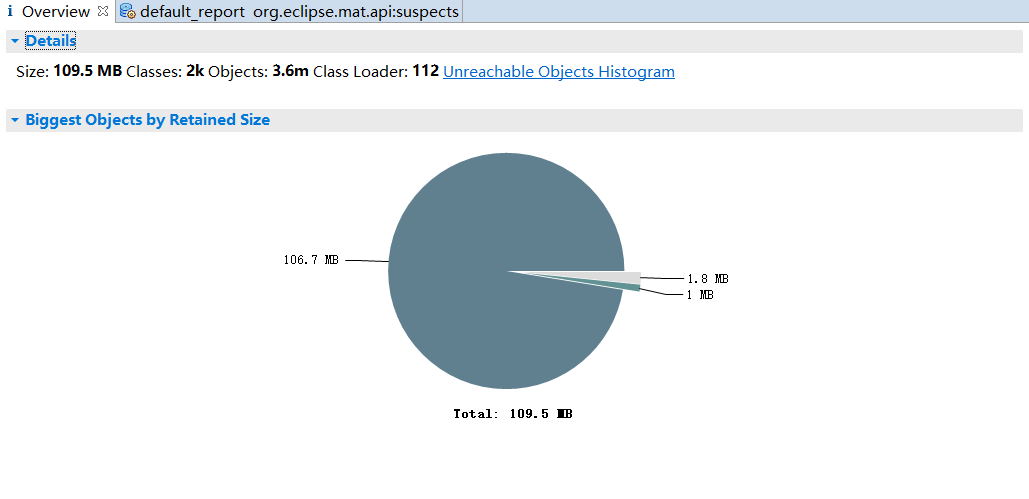
## Memory Dump Analysis and Performance Optimization

### 内存导出

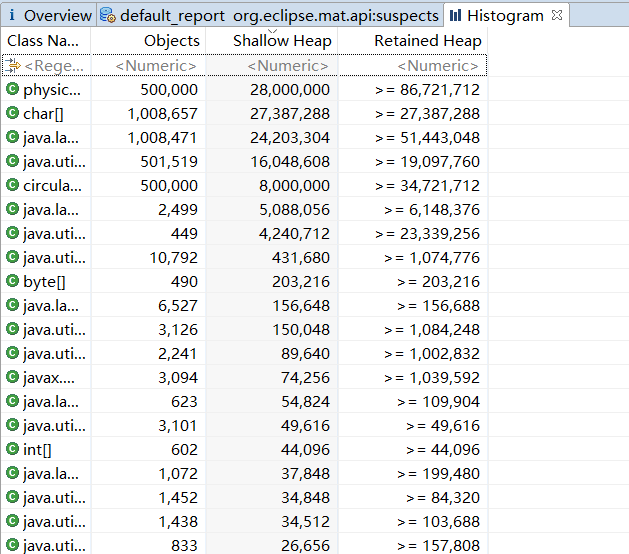


### 使用MAT分析内存导出文件

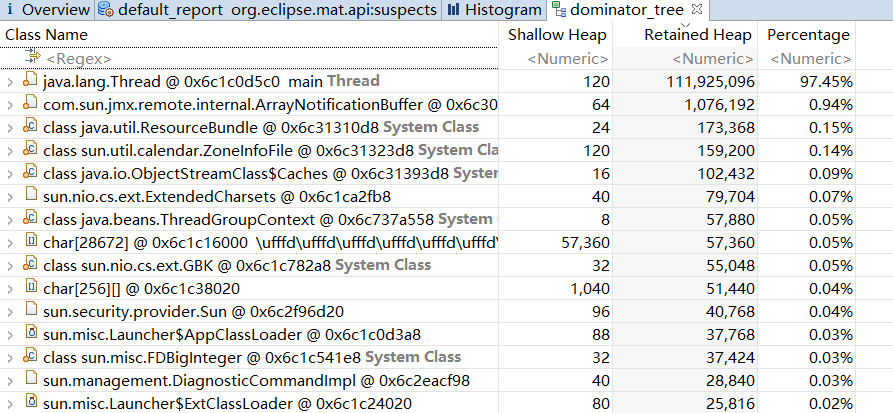
Overview:



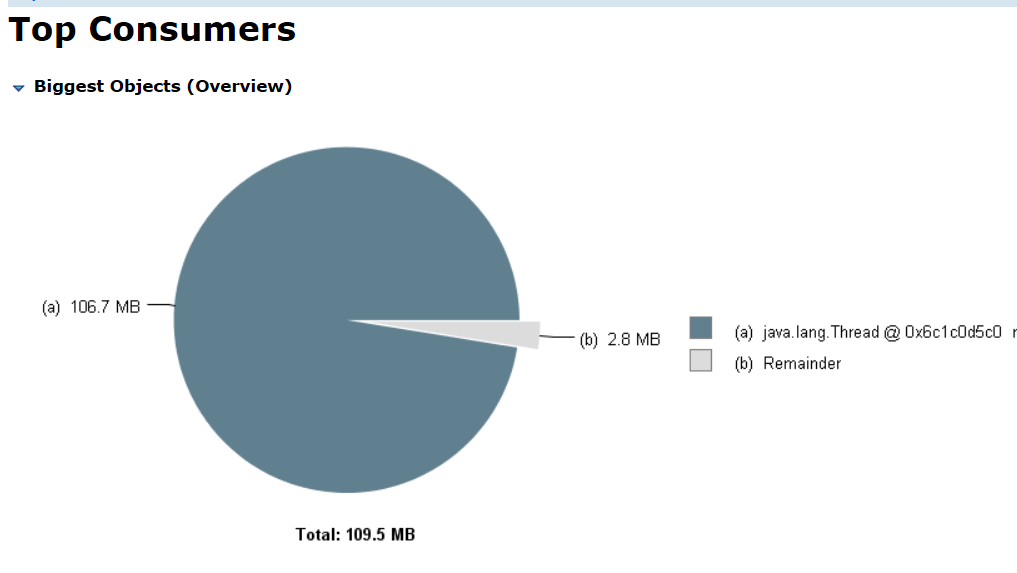
Histogram:



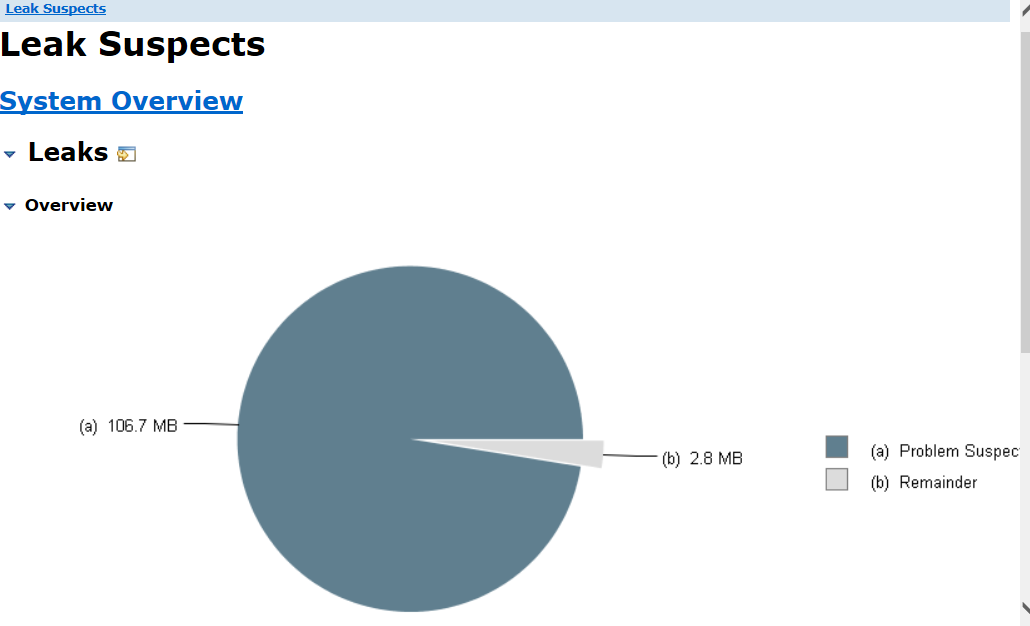
dominator tree:



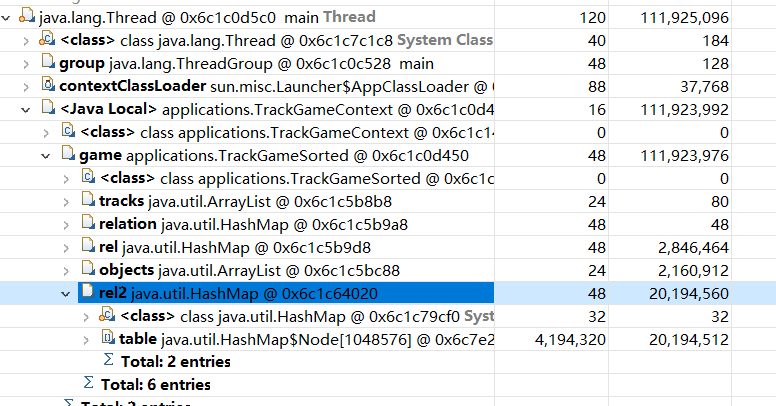
top consumer:



leak Suspects:

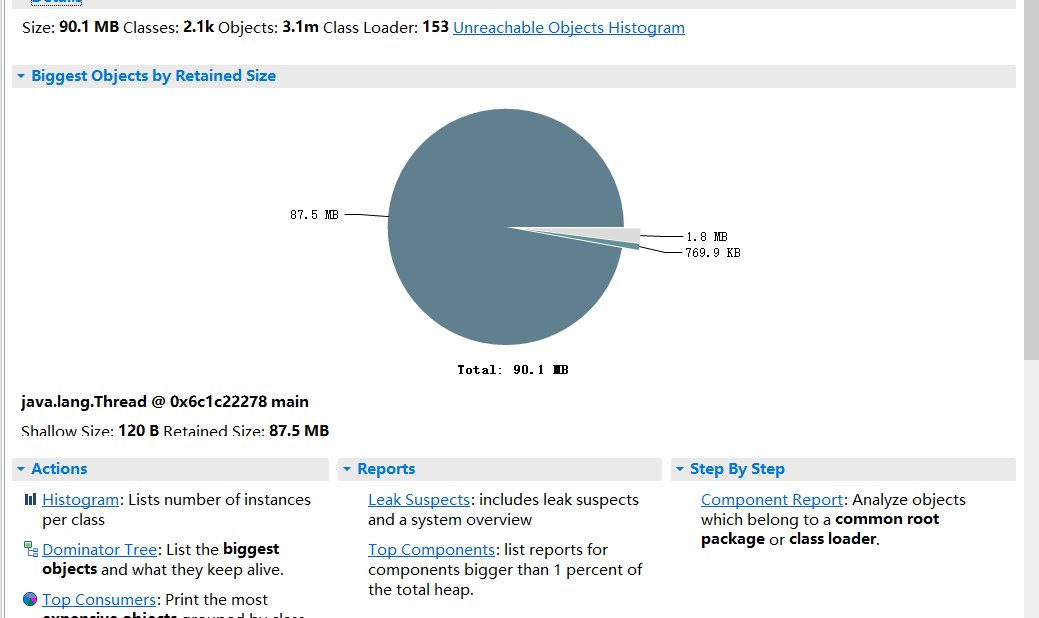


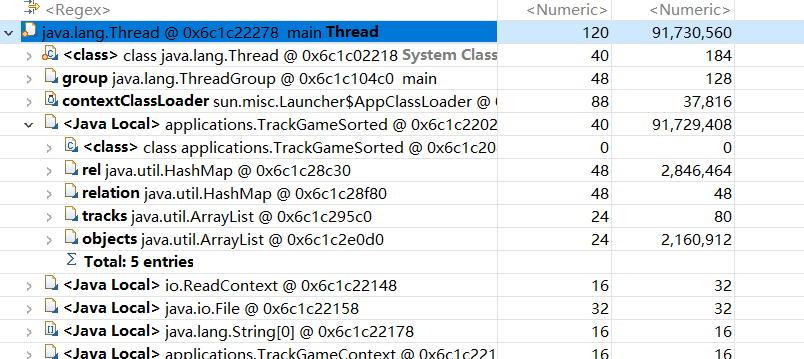
### 发现热点/瓶颈并改进、改进前后的性能对比分析



发现rel2占用很大的内存，但是却没什么用。可以尝试删除rel2来提高性能。

效果：

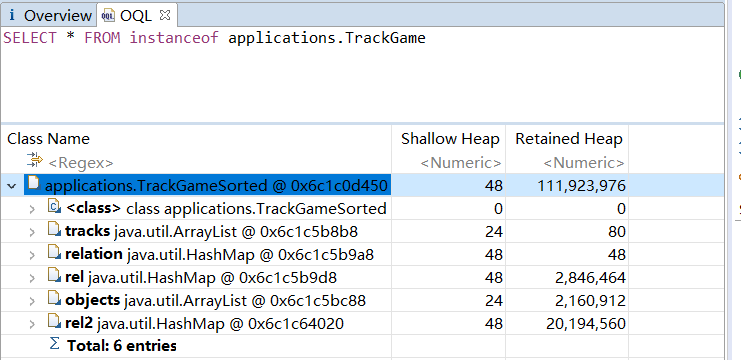




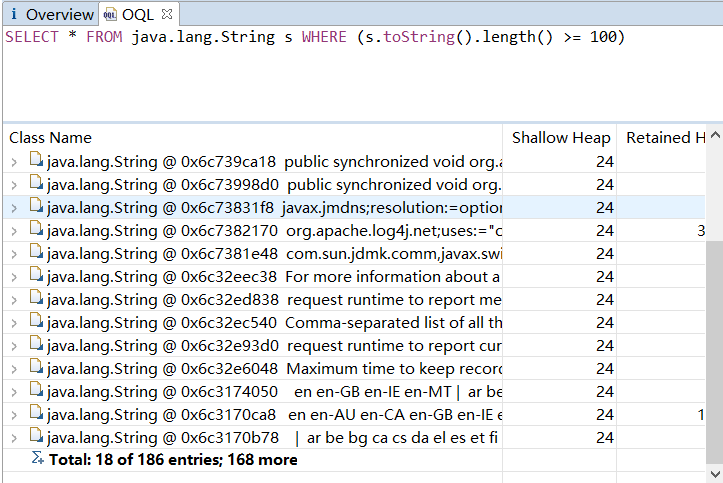
还是有很大提升的！

### 在MAT内使用OQL查询内存导出

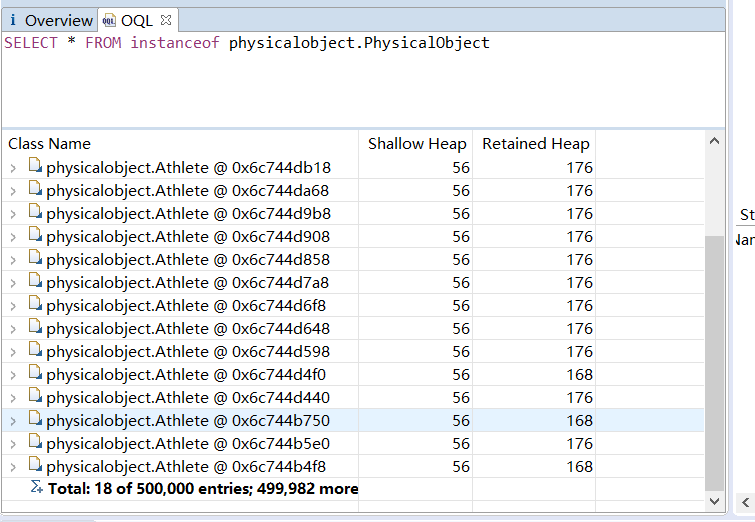
TrackGame对象：



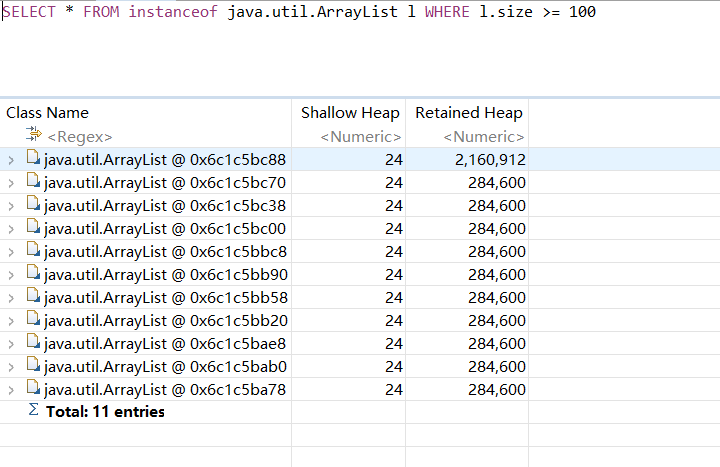
长度大于等于100的对象：



PhysicalObject（及其子类）的对象实例的数量和总占用内存大小

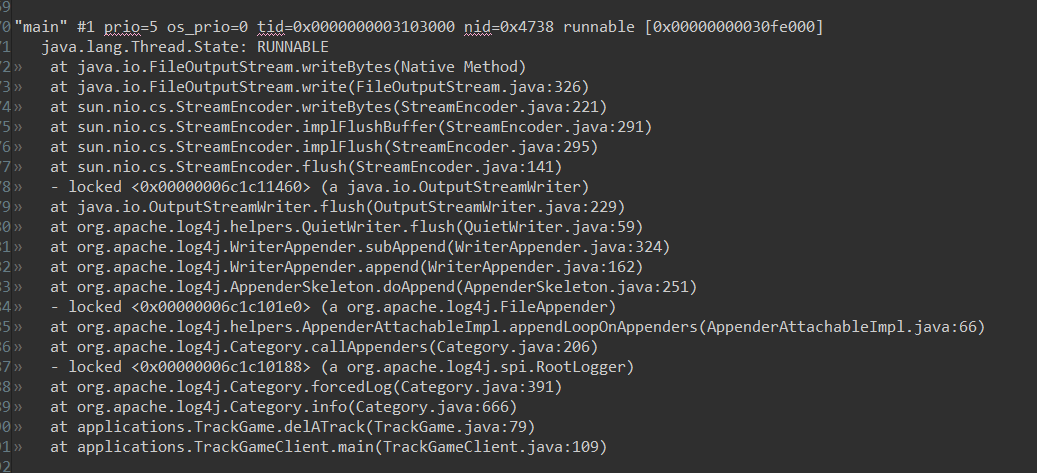


包含元素数量大于100的collection实例：

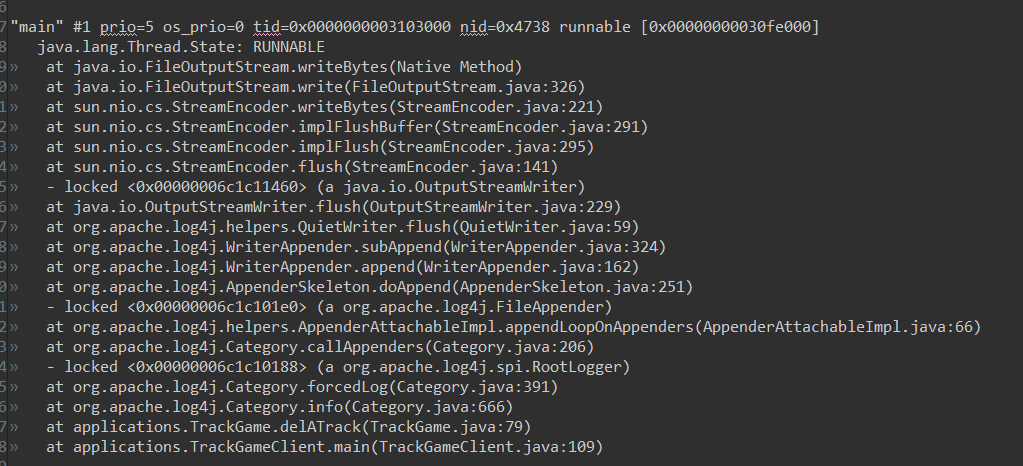


### 观察jstack/jcmd导出程序运行时的调用栈

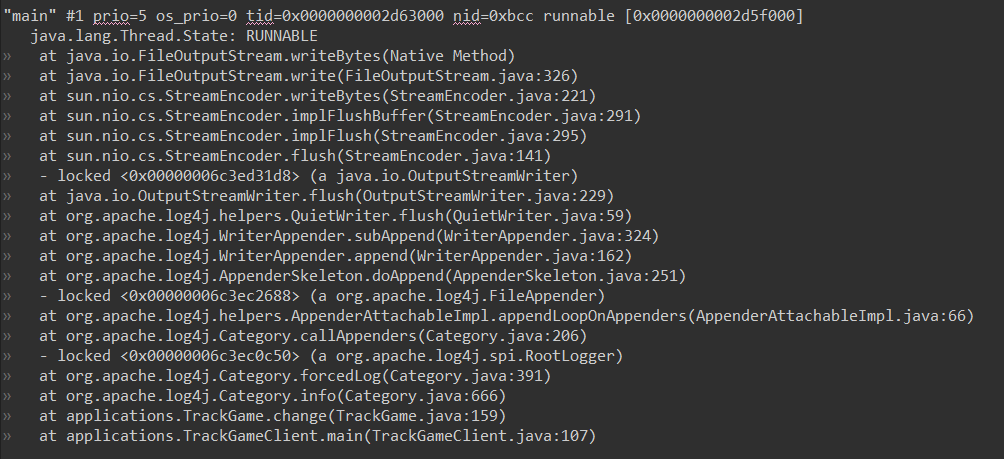
增加新轨道:



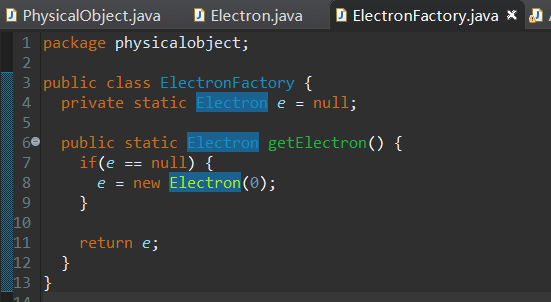
删除轨道:



交换运动员:

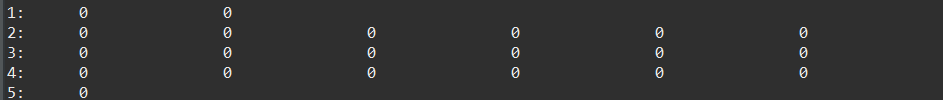


### 使用设计模式进行代码性能优化

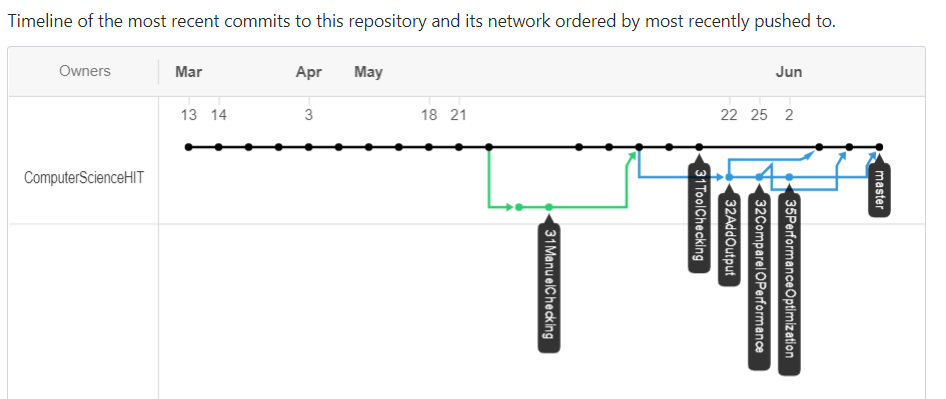


实现了一个。用单例模式实现ElectronFactory，所有外层电子都一样，只有轨道信息不同，而轨道信息存在rel/rel2中，所以可以干脆使用单例模式。

效果，成功：



## Git仓库结构



# 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 |
| 5.21 | 晚上 | 完成3.1 | 差好多 |
| 5.22 | 晚上 | 完成3.1 | 完成 |
| 5.23 | 晚上 | 完成3.2 | 未完成 |
| 5.24 | 晚上 | 完成3.2 | 未完成 |
| 5.25 | 晚上 | 完成3.2 | 完成 |
| 5.28 | 晚上 | 完成3.3 | 未完成 |
| 5.29 | 晚上 | 完成3.3 | 完成 |
| 6.1 | 晚上 | 完成3.4 | 完成 |
| 6.2 | 晚上 | 完成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 |
| 读文件设计模式不太会。 | 百度查看博客园相关文章。 |
| 读文件时间太长，构造系统太大，出现bug | 优化数据结构 |
| 不会使用调试工具 | 百度 |

# 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

在实现数据结构时就要考虑到数据量的问题，当输入大规模数据时，数组很可能会越界，要提前设计好数据结构，不然一改就要全动，很麻烦。

## 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。
11. 运行很美。
12. 没什么帮助。
13. 不太能。
14. 记不太清去年学的了。
15. 自动的更便利，但不够随心所欲。有的吧。
16. 能，优化很大。
17. 能发现，不过改起来不太容易。
18. 还好。
19. 工作量有点大，难度一般，deadline也还行。
20. 没什么意见和建议。