

**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 | 陈鋆 |
| 学号 | 1170300513 |
| 班号 | 11703005 |
| 电子邮件 | 295951960@qq.com |
| 手机号码 | 18059077917 |

**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

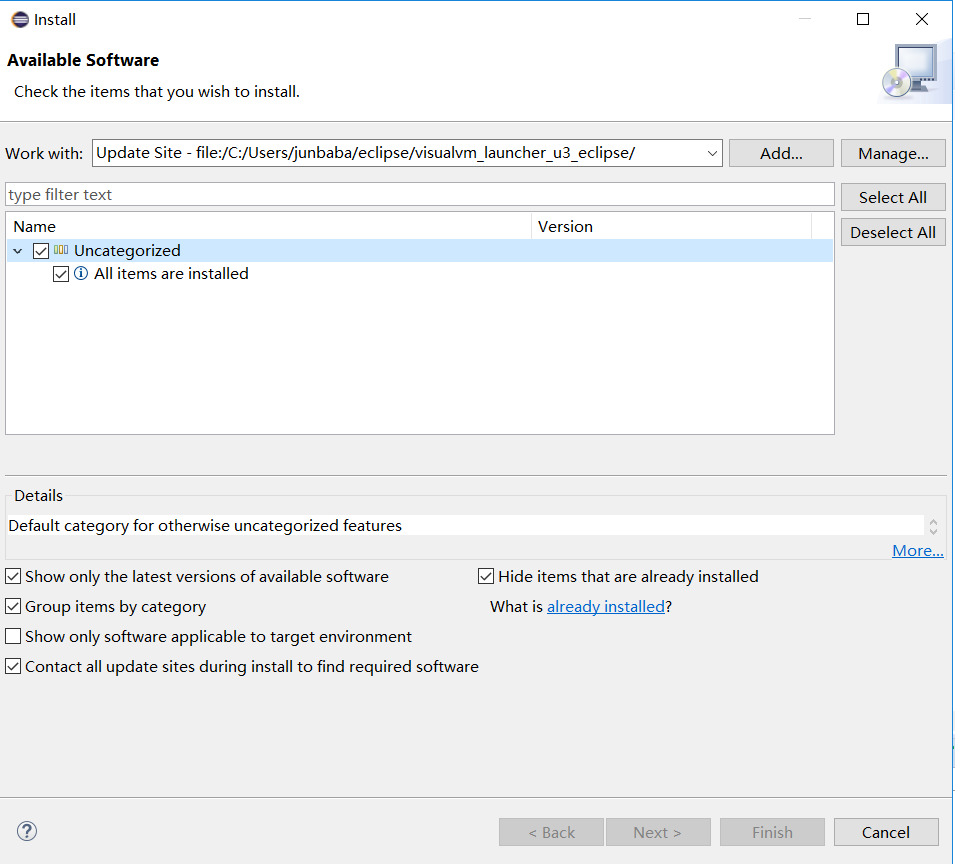
⚫ 代码重构

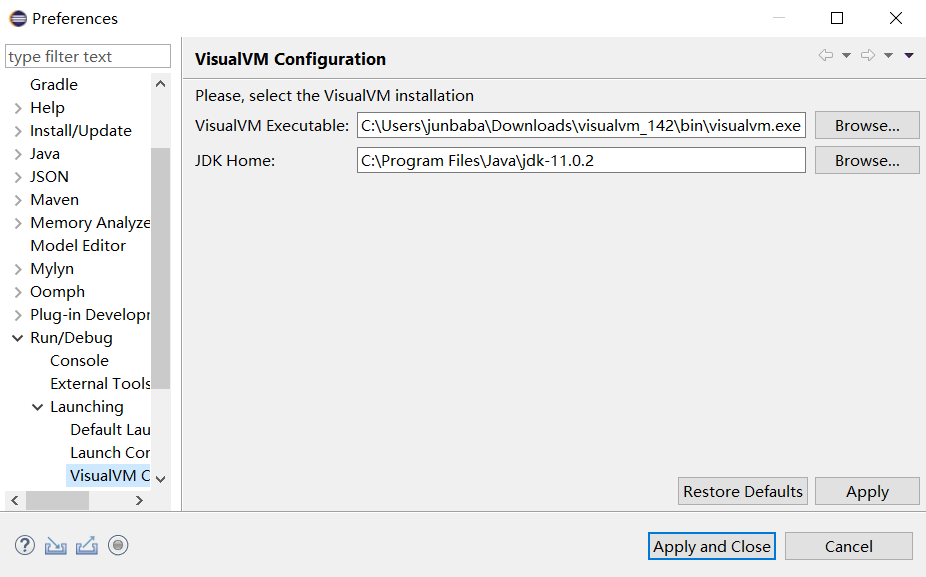
# 实验环境配置

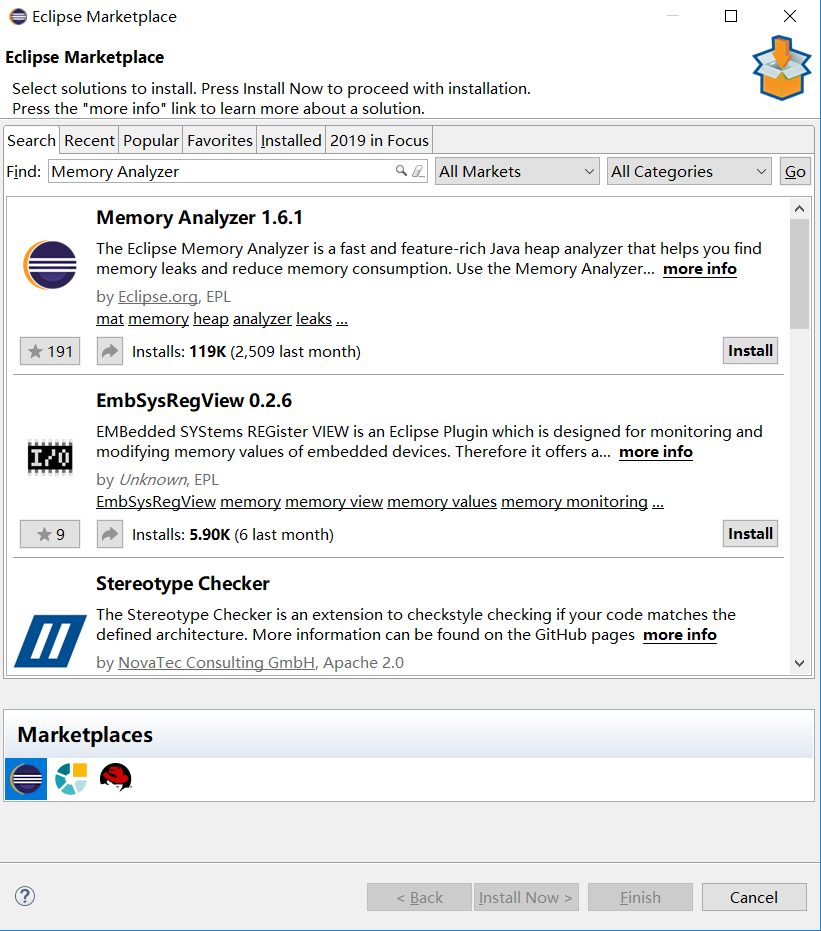
简要陈述你配置本次实验所需环境的过程，必要时可以给出屏幕截图。

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。

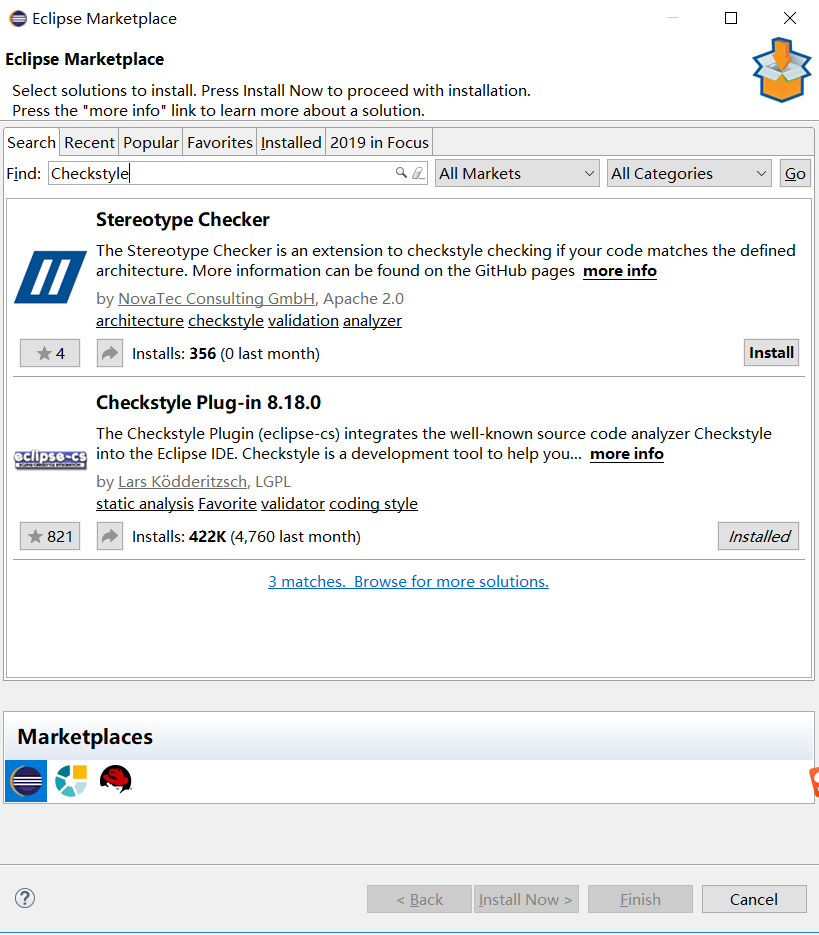
在 Eclipse IDE 中配置并运行 VisualVM 和 Memory Analyzer (MAT)：







配置并运行 CheckStyle 和 SpotBugs 工具



在这里给出你的GitHub Lab5仓库的URL地址（Lab5-学号）。

<https://github.com/ComputerScienceHIT/Lab5-1170300513>

# 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！）。

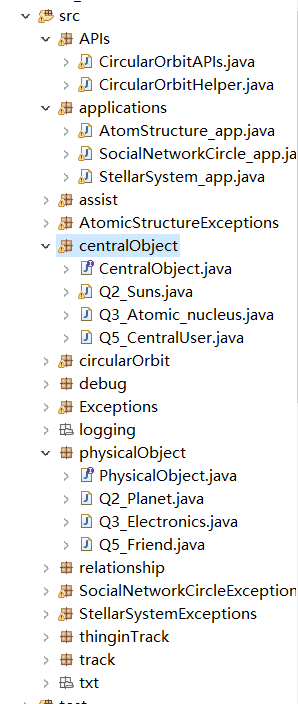
## Static Program Analysis

### 人工代码走查（walkthrough）

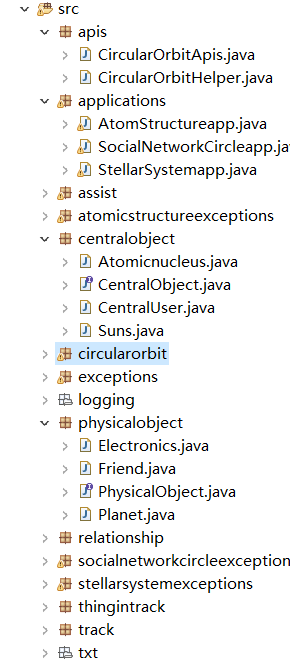
**使用Google的代码规范。**阅读规范，对 Lab4 代码进行人工代码走查（walkthrough and review），做出修改。主要关注命名、布局（空行、空格、缩进、分行等）、注释（java doc、RI、AF、safety from rep exposure、函数 spec 等）、文件/包的组织等。针对所发现的问题，对代码进行修改，消除这些问题。

命名上的区别：

改动前：

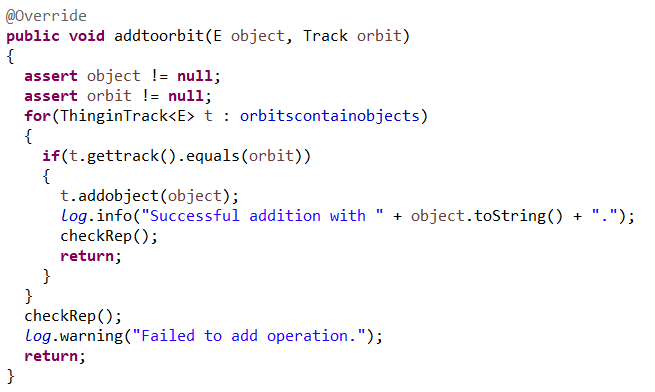


改动后：

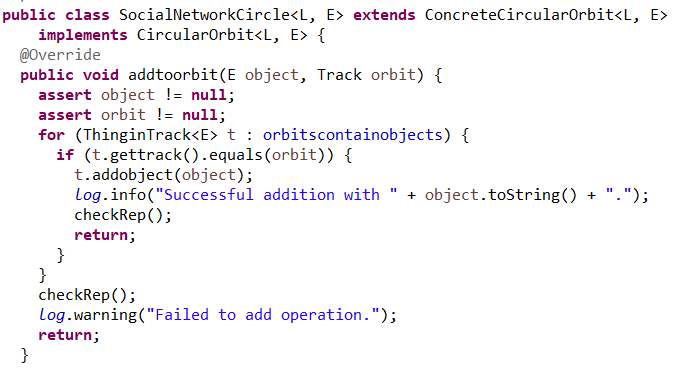


布局上的区别：

改动前：

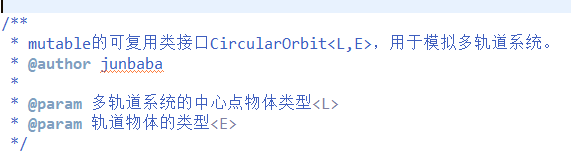


改动后：

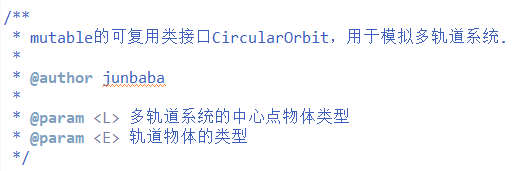


注释上的区别：

改动前：



改动后：



### 使用CheckStyle和SpotBugs进行静态代码分析

使用 CheckStyle 和 SpotBugs 工具对经过人工走查的 Lab4 代码进行自动的静态代码分析。通过查阅资料理解清楚这两个工具找出的所有问题，并手动进行修改，确保再次运行工具不会出现同样的问题。

**CheckStyle:**

命名上的区别：

改动前：



改动后：



布局上的区别：

改动前：

if都要加上{}：

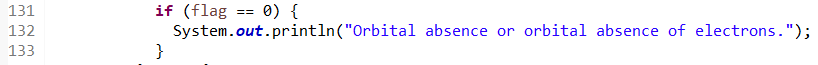


赋值要分行来进行：

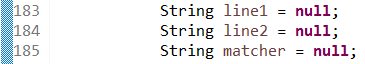


改动后：

if都要加上{}：

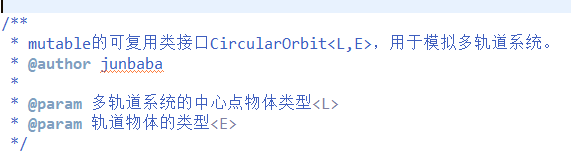


赋值要分行来进行：

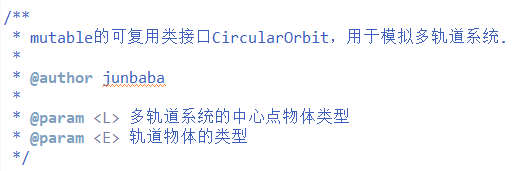


注释上的区别：

改动前：

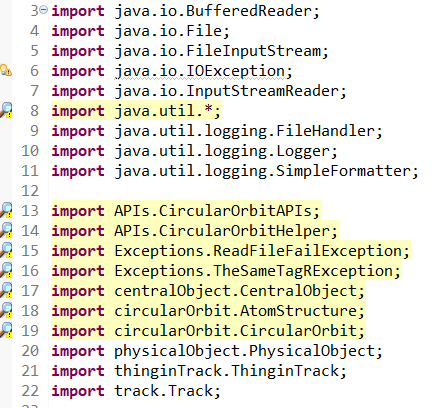


改动后：



文件/包的组织的区别：

改动前：



改动后：



**SpotBugs:**

Lab4中已经将存在的Bug隐患修改，故此次未发现问题。

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。

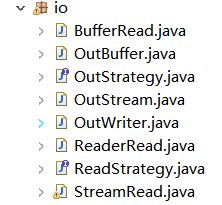
二者发现问题的能力均很强大，但在发现问题的类型上有所差异。

CheckStyle发现的问题多数是文法、格式、布局上的问题，使代码更符合java要求的规范；SpotBugs发现的问题则是潜在的那些Bug隐患，是代码尽量少出现那些常见的bug。

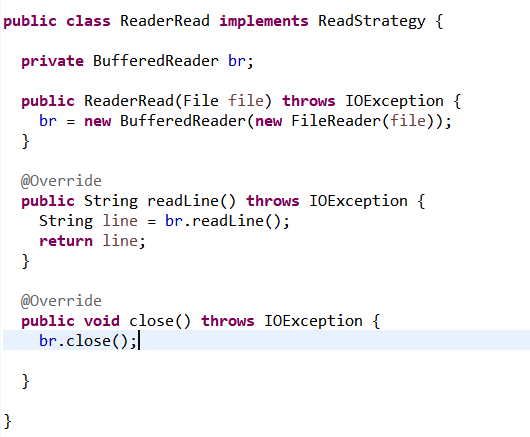
## Java I/O Optimization

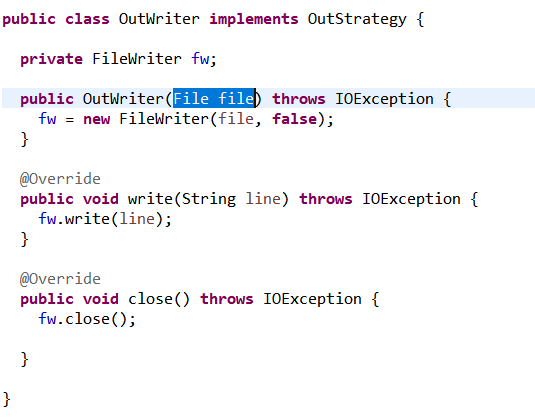
### 多种I/O实现方式

实现了哪些I/O方式来读写文件，具体如何实现的。

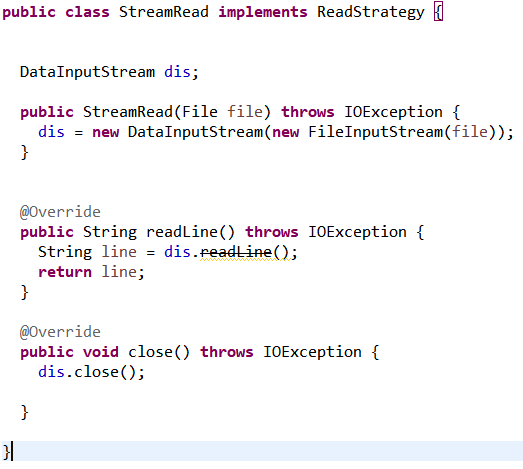


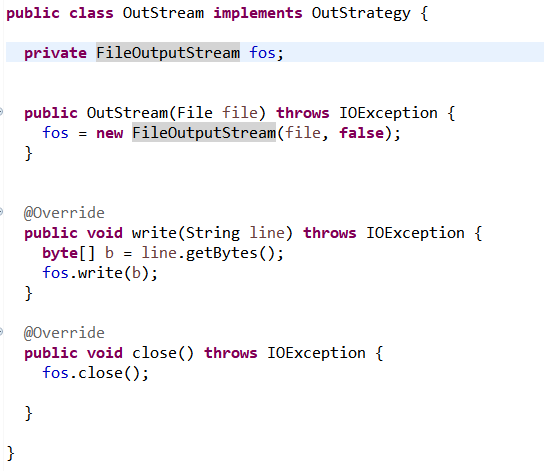
Reader/Writer方式：



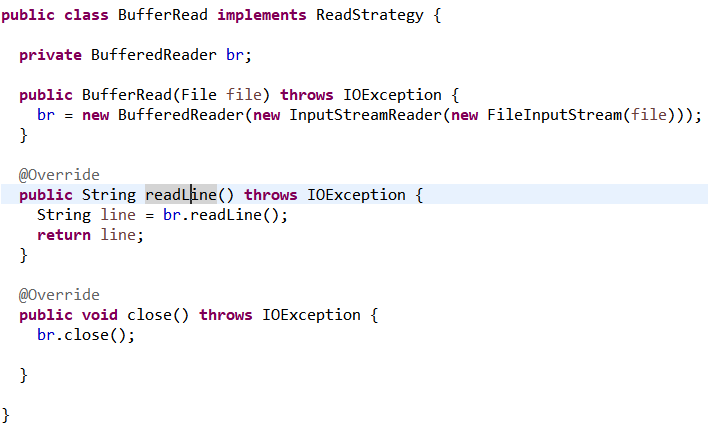


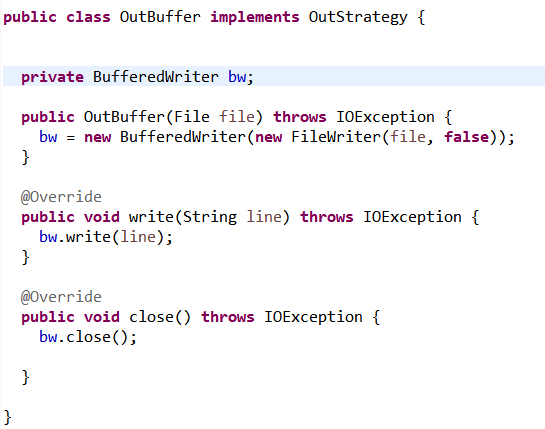
Stream方式：





Buffer方式：





如何用strategy设计模式实现在多种I/O策略之间的切换。

定义接口ReadStrategy和OutStrategy，分别用不同的I/O策略的类来实现这两个接口。

在这两个接口中甚至相应的静态工厂方法，来实现不同I/O策略类的初始化。

**ReadStrategy：**

**public** **interface** ReadStrategy {

/\*\*

\* 静态工厂方法，返回一个实现ReadStrategy的类.

\*/

**public** **static** ReadStrategy empty(File file, **int** choice) **throws** IOException {

**if** (choice == 1) {

**return** **new** ReaderRead(file);

} **else** **if** (choice == 2) {

**return** **new** StreamRead(file);

} **else** **if** (choice == 3) {

**return** **new** BufferRead(file);

} **else** {

**throw** **new** UnsupportedOperationException();

}

}

/\*\*

\* 按行读文件的方法.

\*

\* **@return** 文件的某一行.

\*/

**public** String readLine() **throws** IOException;

/\*\*

\* 关闭文件的方法.

\*/

**public** **void** close() **throws** IOException;

}

**OutStrategy：**

**public** **interface** OutStrategy {

/\*\*

\* 静态工厂方法，返回一个实现OutStrategy接口的类.

\*/

**public** **static** OutStrategy empty(File file, **int** choice) **throws** IOException {

**if** (choice == 1) {

**return** **new** OutWriter(file);

} **else** **if** (choice == 2) {

**return** **new** OutStream(file);

} **else** **if** (choice == 3) {

**return** **new** OutBuffer(file);

} **else** {

**throw** **new** UnsupportedOperationException();

}

}

/\*\*

\* 写文件的方法.

\*

\* **@param** line 要写入的字符串

\*/

**public** **void** write(String line) **throws** IOException;

/\*\*

\* 关闭文件的方法.

\*/

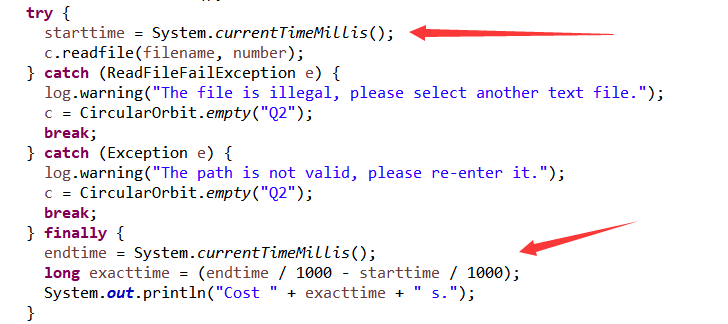
**public** **void** close() **throws** IOException;

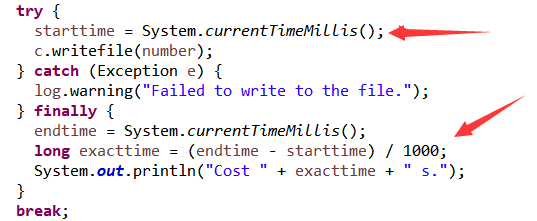
}

### 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。

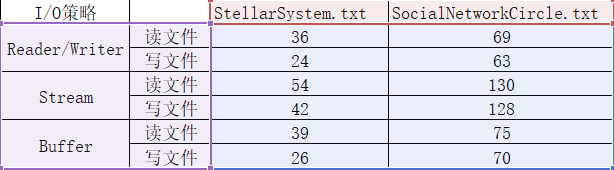
在开始读文件与读文件结束后分别记录系统时间，再作减法即可获得I/O语法文件的时间。



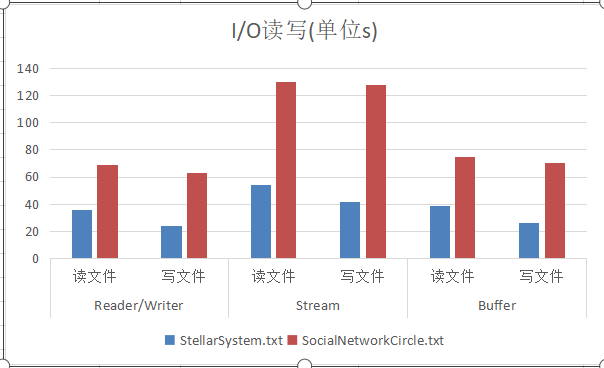


表格方式对比不同I/O的性能。（本处所展示的是单独读文件的时间。）

单位为秒：



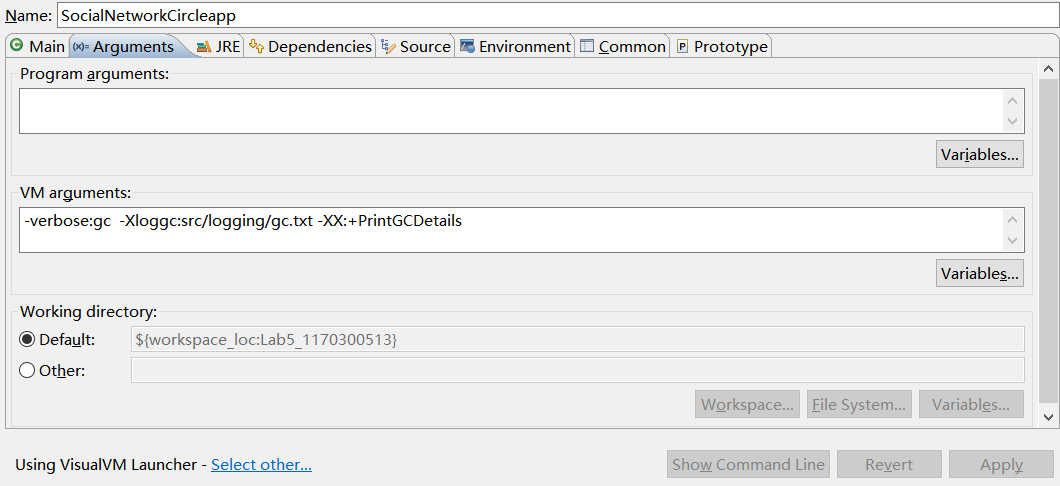
图形对比不同I/O的性能。



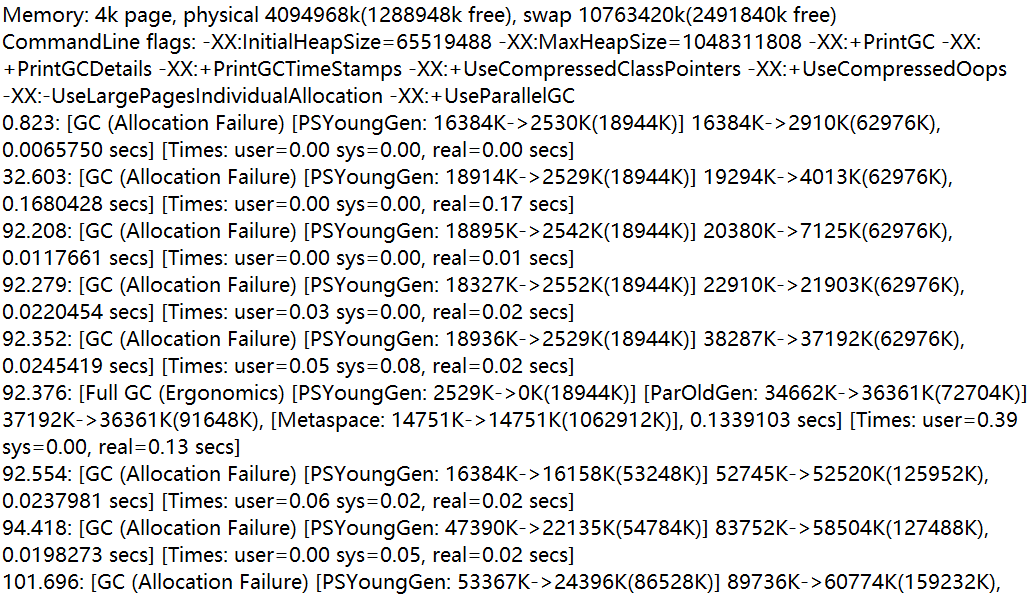
## Java Memory Management and Garbage Collection (GC)

### 使用-verbose:gc参数

设置参数：



生成log日志：



1. Full GC：42s/次 Minor GC：0.08s/次
2. Full GC：0.13s Minor GC：0.02s
3. PSYoungGen区域每过一段时间占有空间增加到一定大小后会调用GC使得其占有的空间大大减少；Metaspace区域基本保持不变；ParOldGen一段时间后占有的空间会相应增大。

### 用jstat命令行工具的-gc和-gcutil参数

1. jstat -gc

使用 jstat 命令行工具的-gc参数，对程序的内存使用和垃圾回收情况进行周期性监控，包括对 heap 中各区域（young、old、metaspace）的 size 和垃圾回收状况的监控与统计分析，根据输出结果判断程序的内存回收**情况正常**.

S0C：年轻代中第一个survivor（幸存区）的容量 (字节)

S1C：年轻代中第二个survivor（幸存区）的容量 (字节)

S0U：年轻代中第一个survivor（幸存区）目前已使用空间 (字节)

S1U：年轻代中第二个survivor（幸存区）目前已使用空间 (字节)

EC：年轻代中Eden（伊甸园）的容量 (字节)

EU：年轻代中Eden（伊甸园）目前已使用空间 (字节)

OC：Old代的容量 (字节)

OU：Old代目前已使用空间 (字节)

PC：Perm(持久代)的容量 (字节)

PU：Perm(持久代)目前已使用空间 (字节)

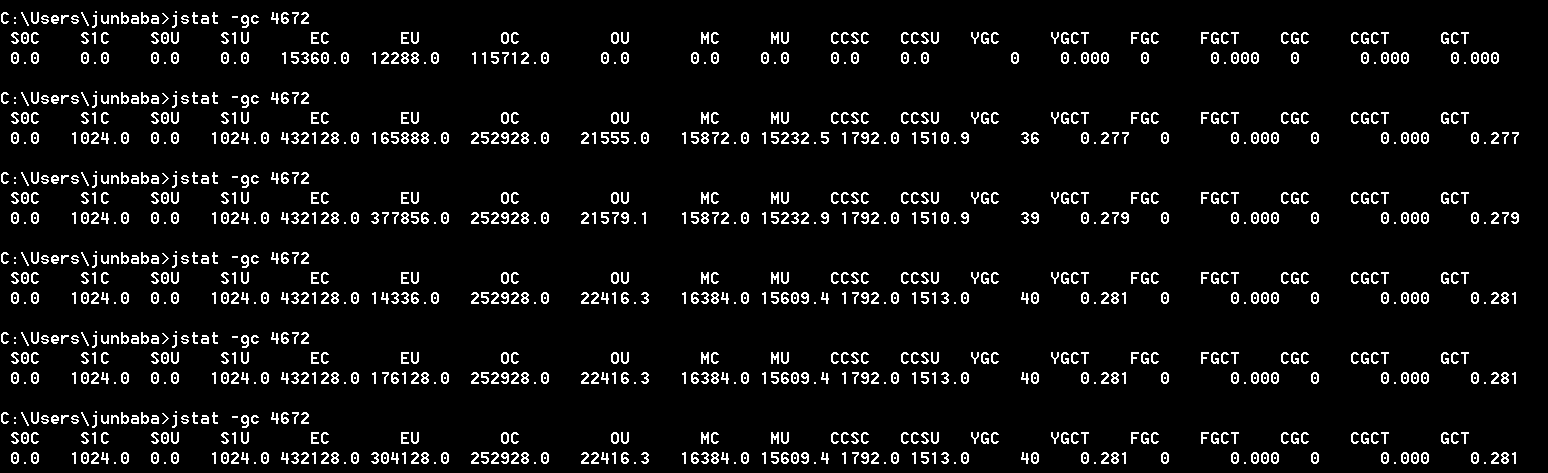
YGC：从应用程序启动到采样时年轻代中gc次数

YGCT：从应用程序启动到采样时年轻代中gc所用时间(s)

FGC：从应用程序启动到采样时old代(全gc)gc次数

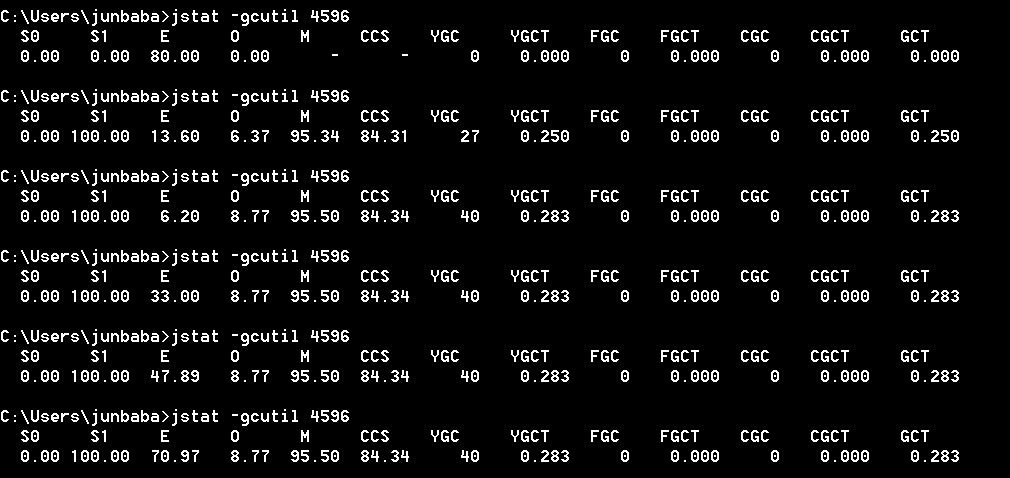
FGCT：从应用程序启动到采样时old代(全gc)gc所用时间(s)

GCT：从应用程序启动到采样时gc用的总时间(s)



1. java -gcutil

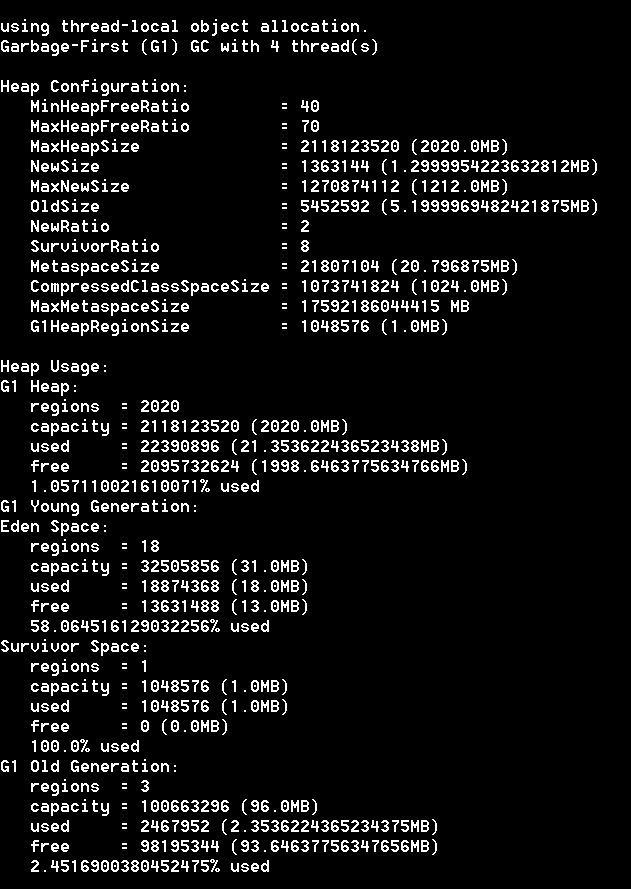
使用 jstat 命令行工具的-gcutil参数，对程序的内存使用和垃圾回收情况进行周期性监控，包括对 heap 中各区域（young、old、metaspace）的 size 和垃圾回收状况的监控与统计分析，根据输出结果判断程序的内存回收**情况正常**.



### 使用jmap -heap命令行工具

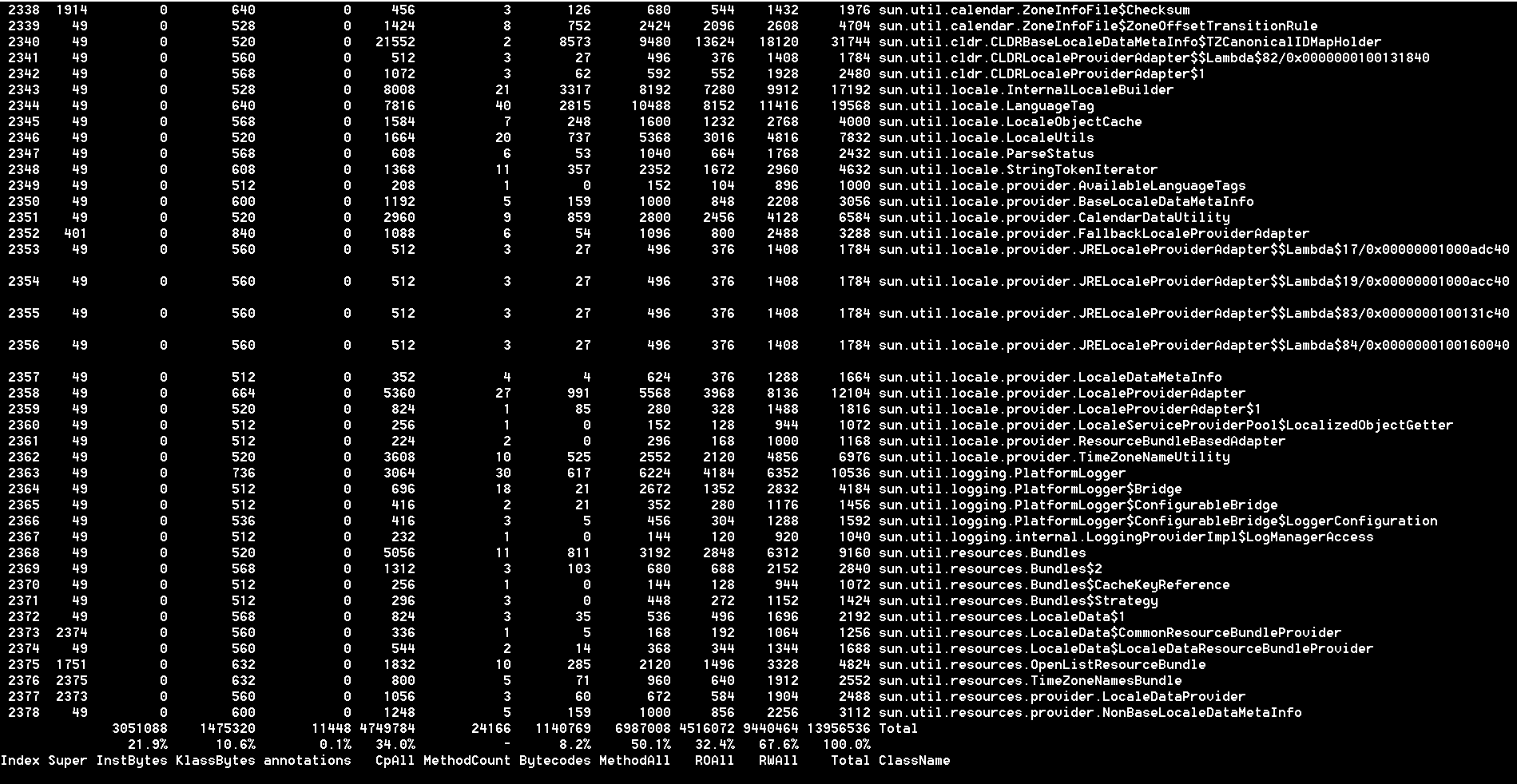
使用 jmap -heap命令行工具查询程序的内存使用信息，包括虚拟机当前所使用的 GC 策略、heap 的配置情况（各区域的大小等）、heap 的使用情况统计。





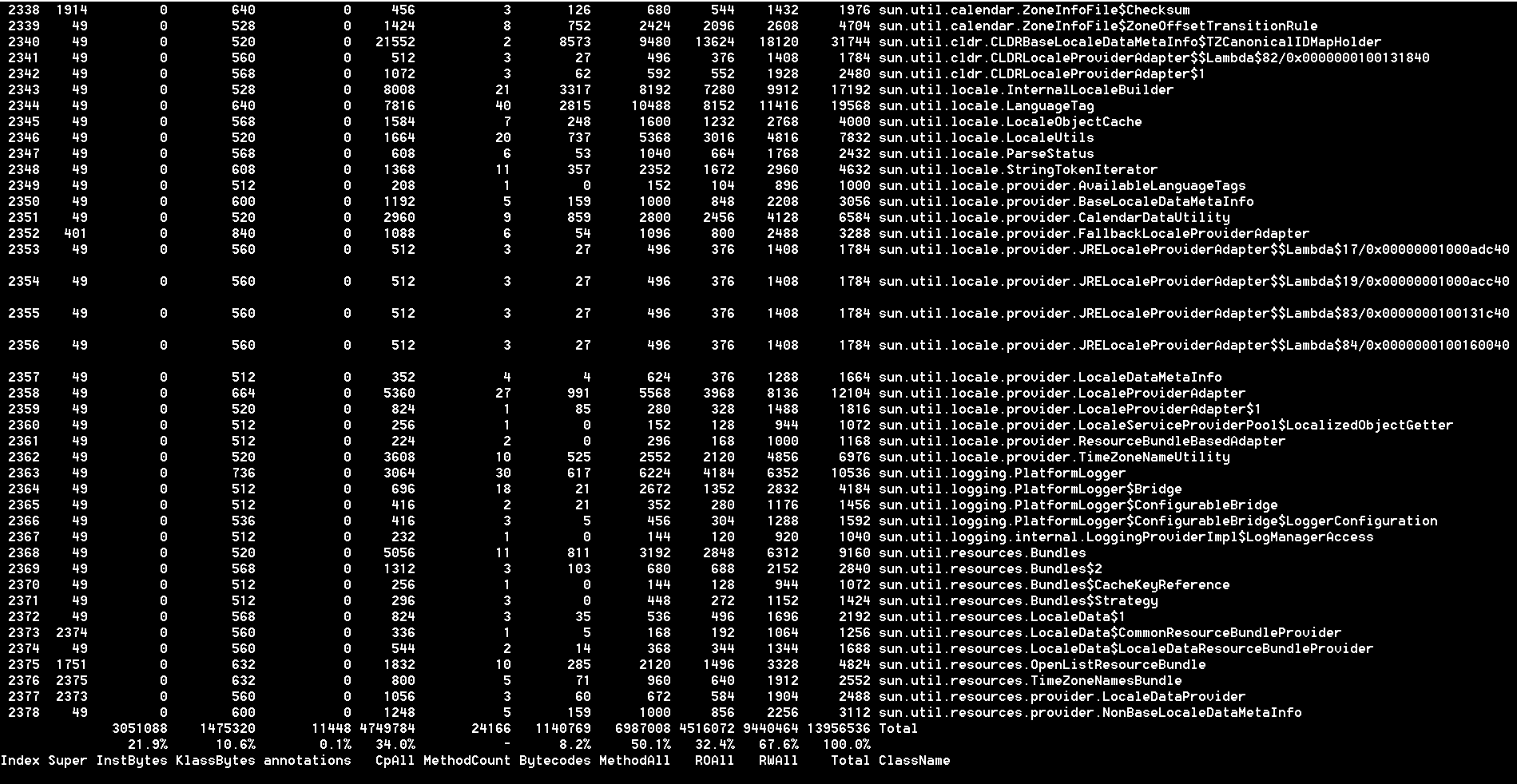
### 使用jmap -clstats命令行工具

使用 jmap -clstats 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。



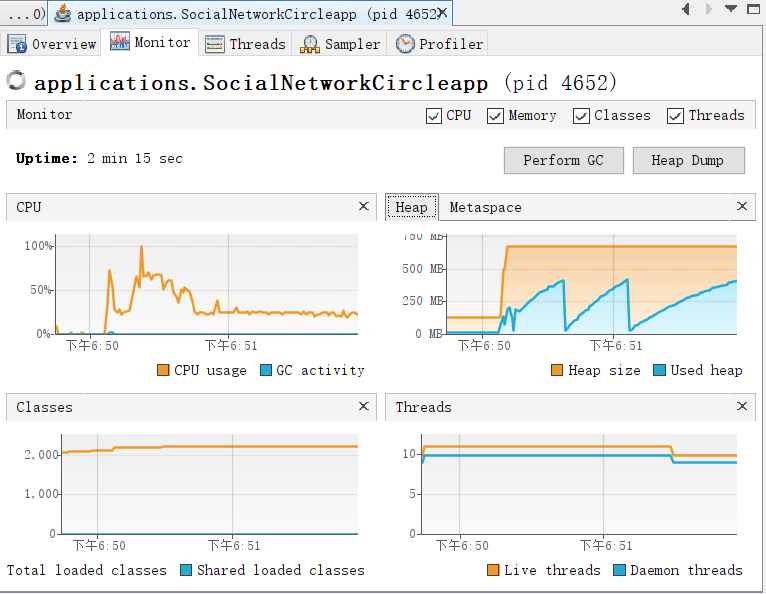
### 使用jmap -permstat命令行工具

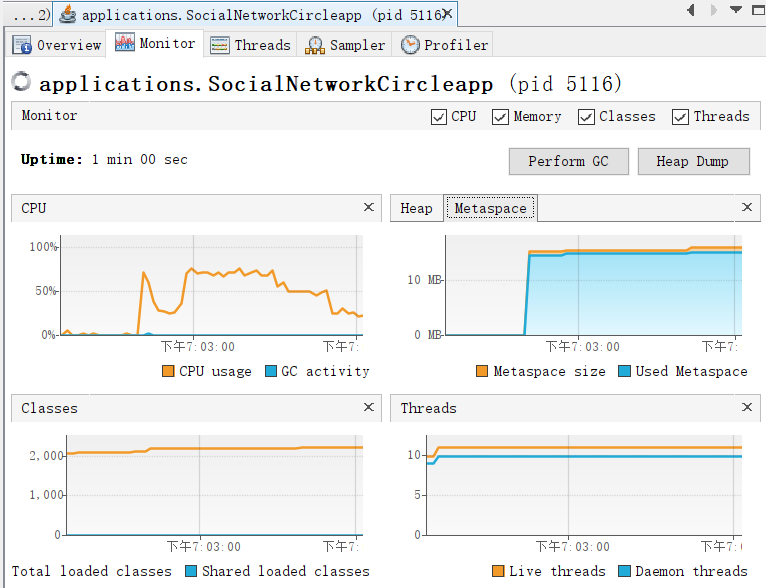
使用 jmap -permstat命令行工具查看 class loader 的统计信息，即程序执行期间装载的 class 和 method 相关信息。



### 使用JMC/JFR、jconsole或VisualVM工具

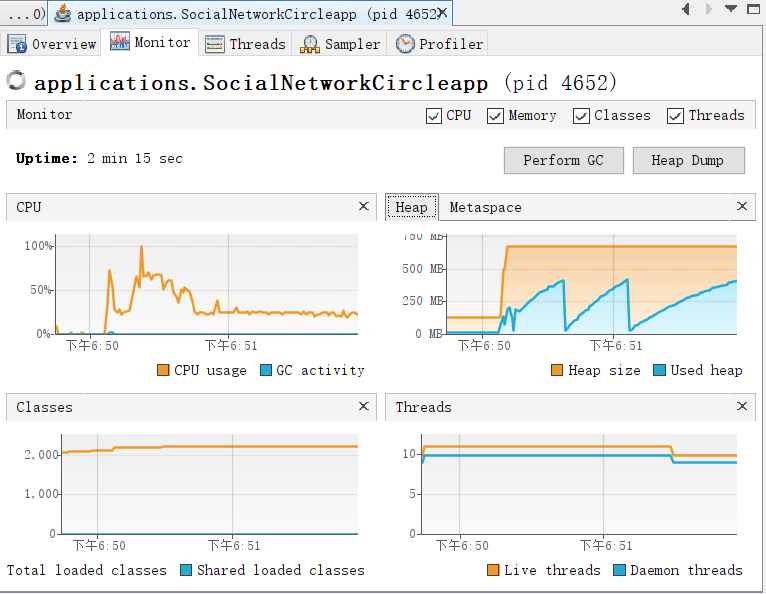
使用VisualVM 工具，尝试着可视化查看程序当前的内存使用，包括 heap 的分配与占用、meta space 情况、装载的类实例统计等。



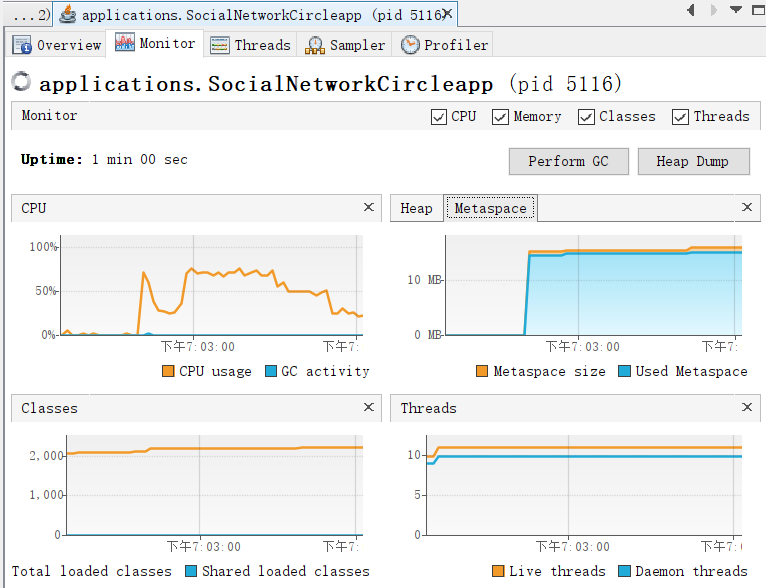


### 分析垃圾回收过程

在启动程序的时候，使用-verbose:gc 参数，在控制台输出程序的 GC 情况或同时输出至 log 文本文件中，控制台输出或 log 文件进行简要分析，可知垃圾回收过程：PSYoungGen区域每过一段时间占有空间增加到一定大小后会调用GC使得其占有的空间大大减少；ParOldGen一段时间后占有的空间会相应增大。这也恰好与3.3.6中的VisualVM 工具所展现出的Heap的变化规律相符。



在启动程序的时候，使用-verbose:gc 参数，在控制台输出程序的 GC 情况或同时输出至 log 文本文件中，控制台输出或 log 文件进行简要分析可知，垃圾回收过程中：Metaspace区域基本保持不变，这也恰好与3.3.6中的VisualVM 工具所展现出的Metaspace的变化规律相符。



### 配置JVM参数并发现优化的参数配置

使用JVM 参数设定功能，利用 java 命令行参数或者在 Eclipse 中配置这些参数，对你的程序的 heap 参数进行不同的设置（例如：初始和最大 heap size、young generation space、metaspace 的初始和最大size 、 NewRatio 、 SuvivorRatio 、 MinHeapFreeRatio 、MaxHeapFreeRatio、GC 模式等），在不同参数设定情况下分别重新执行上述使用-verbose:gc 参数、使用 jmap -heap 命令行工具查询程序的内存使用信息、使用 jmap -histo 命令行工具查询当前装载进内存的各类的实例数目及内存占用情况、使用VisualVM 工具，尝试着可视化查看程序当前的内存使用，包括 heap 的分配与占用、meta space 情况、装载的类实例统计等步骤。

对于GC的性能主要有2个方面的指标：吞吐量(工作时间不算，gc的时间占总的时间比)和暂停时间。

1. 堆大小：

默认情况下，vm会增加/减少heap大小以维持free space在整个vm中占的比例，这个比例由MinHeapFreeRatio和MaxHeapFreeRatio指定。一般而言，server端的app会有以下规则：

一般而言，server端的app会有以下规则：

（1）对vm分配尽可能多的内存；

（2）将Xms和Xmx设为一样的值。如果虚拟机启动时设置使用的内存比较小，这个时候又需要初始化很多对象，虚拟机就必须重复地增加内存。

（3）处理器核数增加，内存也跟着增大。

2. 年轻代：

（1）对于程序流畅性运行影响的因素是新生代的大小。新生代越大，minor collection越少；但是在堆大小固定情况下，新生代越大就意味着越小的老年代，就意味着更多的major collection。

（2）NewRatio反映的是新生代和老年代的大小比例。NewSize和MaxNewSize反映的是young generation大小的下限和上限，将这两个值设为一样就固定了young generation的大小（同Xms和Xmx设为一样）。

（3）SurvivorRatio也可以优化survivor的大小，不过这对于性能的影响不是很大。SurvivorRatio是Eden和Survior大小比例。一般而言，server端的app会有以下规则：首先决定能分配给vm的最大的堆大小，然后设定最佳的young generation的大小；如果堆大小固定后，增加新生代的大小意味着减小老年代大小。让老年代在任何时候够大，能够容纳所有存活的对象（留10%-20%的空余）。

3、年轻代大小选择：

（1）响应时间优先的应用：尽可能设大，直到接近系统的最低响应时间限制，在此种情况下，年轻代收集发生的频率也是最小的，同时,减少到达年老代的对象。

（2）吞吐量优先的应用：尽可能的设置大，可能到达Gbit的程度，因为对响应时间没有要求，垃圾收集可以并行进行，一般适合8CPU以上的应用。

（3）避免设置过小。当新生代设置过小时会导致：①YGC次数更加频繁；②可能导致YGC对象直接进入旧生代，如果此时旧生代满了，会触发FGC。

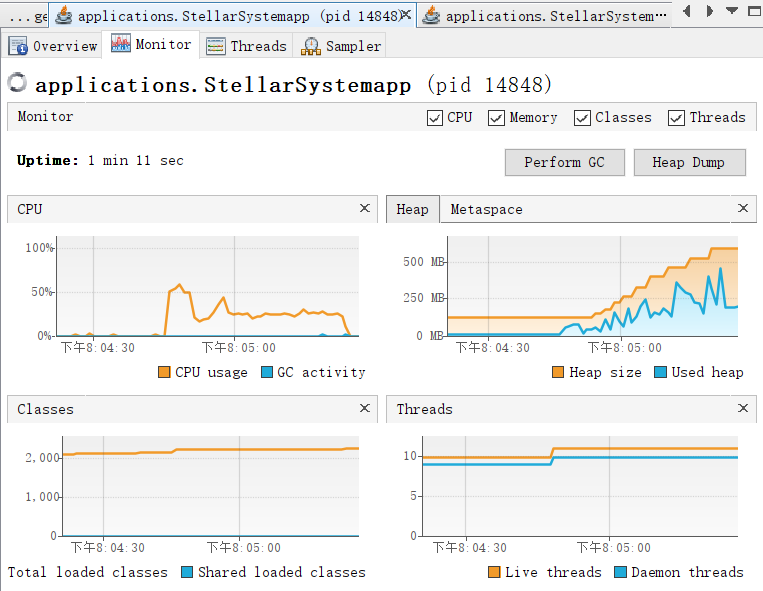
4、年老代大小选择：

（1）响应时间优先的应用：年老代使用并发收集器。如果堆设置小了，可以会造成内存碎片、高回收频率以及应用暂停而使用传统的标记清除方式；如果堆大了，则需要较长的收集时间。

（2）吞吐量优先的应用:一般吞吐量优先的应用都有一个很大的年轻代和一个较小的年老代，这样可以尽可能回收掉大部分短期对象，减少中期的对象，而年老代尽存放长期存活对象。

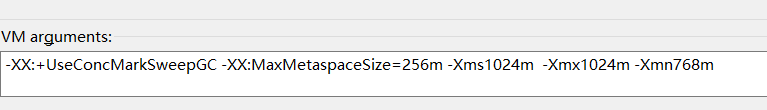
本处选择StellarSystem作为示范，其余两种文件亦同理：

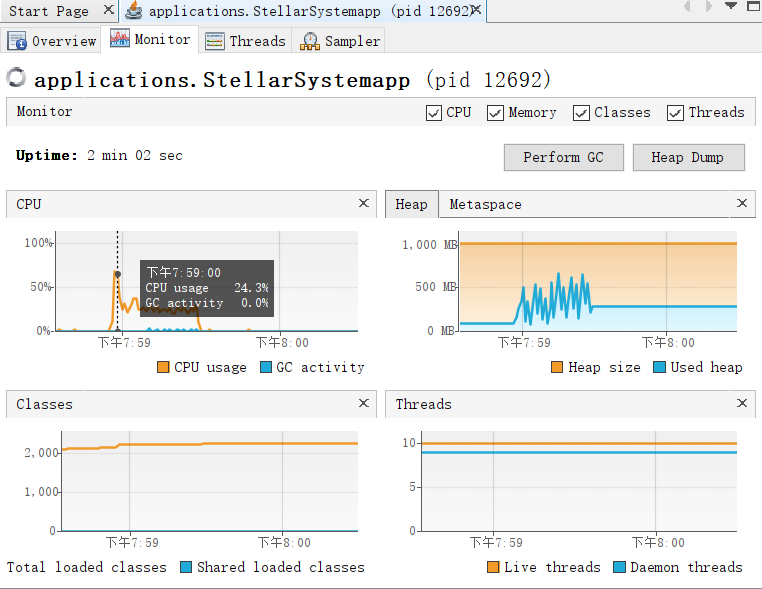
默认：





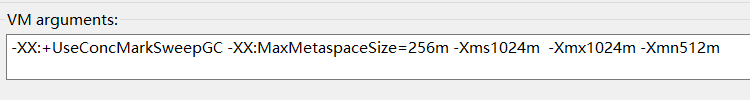
扩大后：

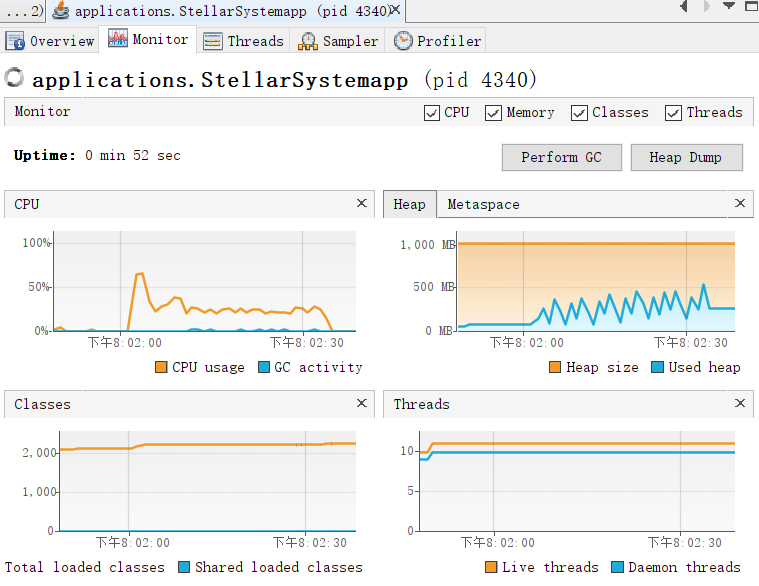






多次尝试后的**最优参数配置**：







## Dynamic Program Profiling

### 使用JMC或VisualVM进行CPU Profiling

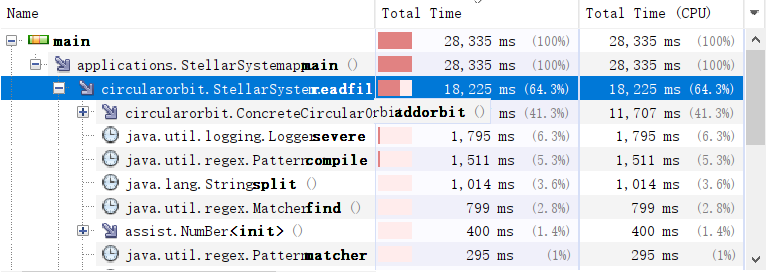
使用VisualVM启动你的程序，让程序读入文件SocialNetworkCircle\_Most.txt，并对生成的图进行各类操作，使用 profiler 发现执行时间热点和内存消耗热点。VisualVM profiler 设定为“自动更新结果”。

选择 CPU Profiling，动态监控程序的执行时间性能：

⚫ 设定要监控的类为你所开发的类（不包含 JDK 提供的公共类）；

⚫ 查看程序中执行的各方法所耗费的时间、所占的比例；

⚫ 着重关注耗费时间居前的各方法，根据你的程序结构来分析这些耗时最多的方法执行是否正常、合理；

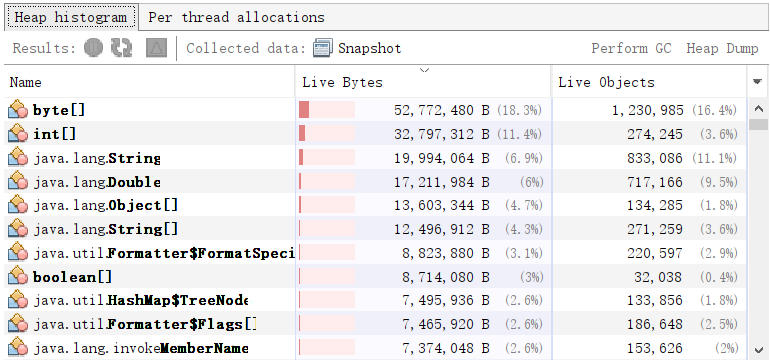


### 使用VisualVM进行Memory profiling

选择 memory profiling，动态监控程序的内存空间性能：

⚫ 查看内存空间中不同类型的对象的个数、所占内存空间大小、所占空间比例等；

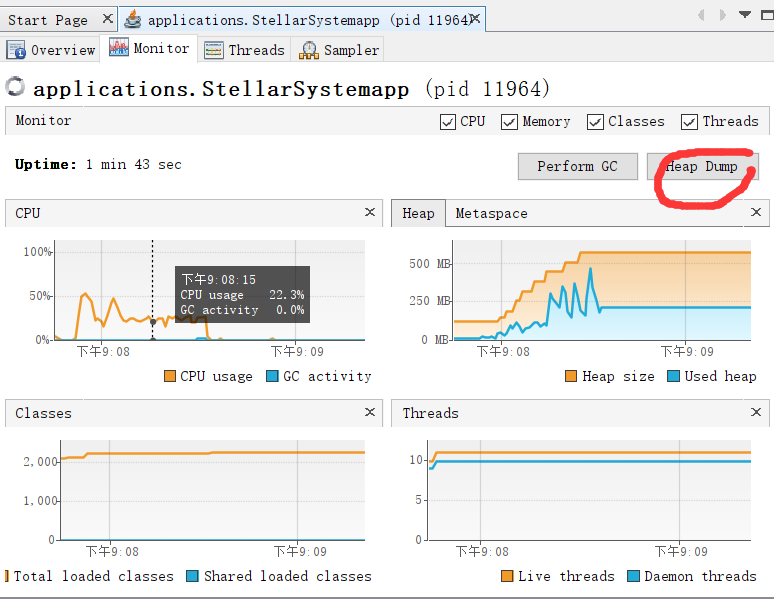
⚫ 着重关注内存占用居前的各类型，根据你的程序结构来分析这些耗费内存最多的类型执行是否正常、合理。



## Memory Dump Analysis and Performance Optimization

### 内存导出

在读取文件结束并生成 CircularOrbit 对象后，利用VisualVM 的内存导出(memory dump)功能，导出当前时刻的 HPROF 文件

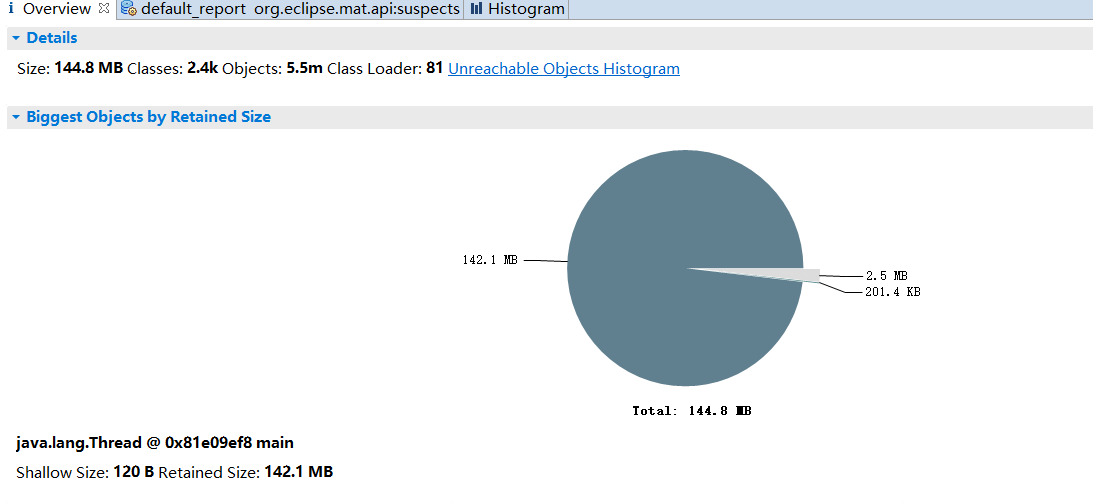


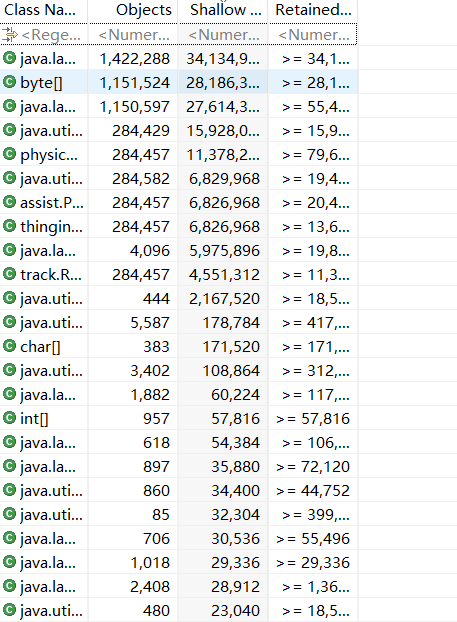
### 使用MAT分析内存导出文件

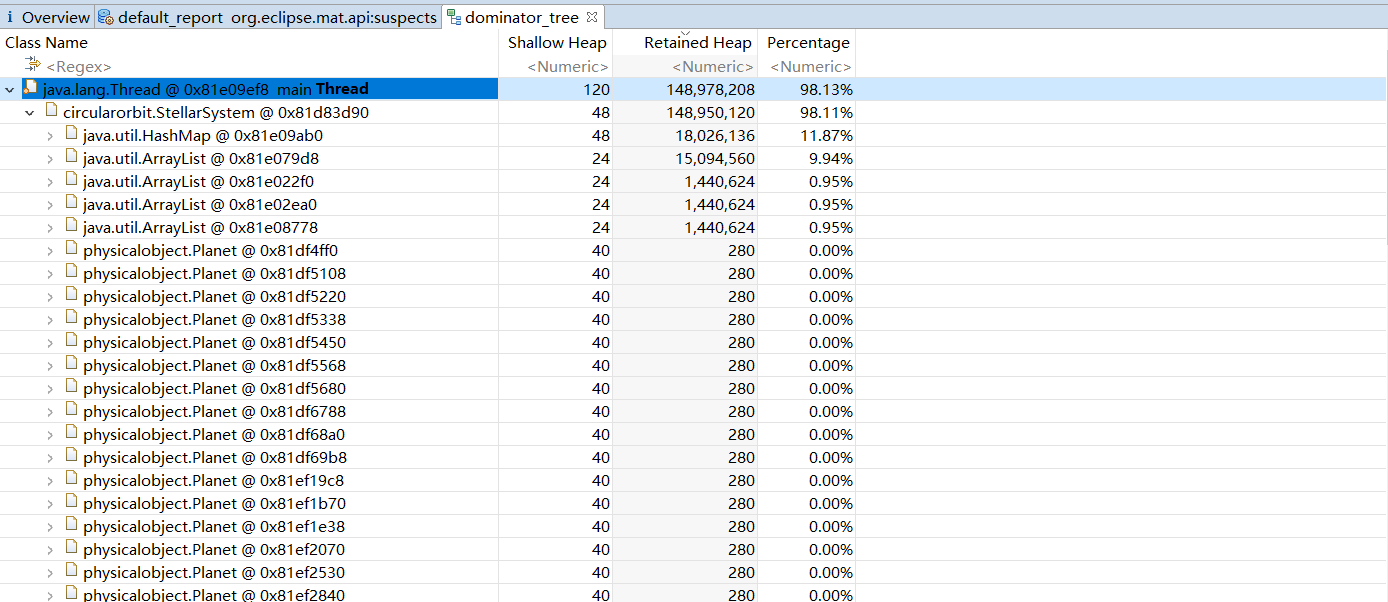
使用 MAT 来分析 3.5 节中产生的内存导出文件.HPROF。

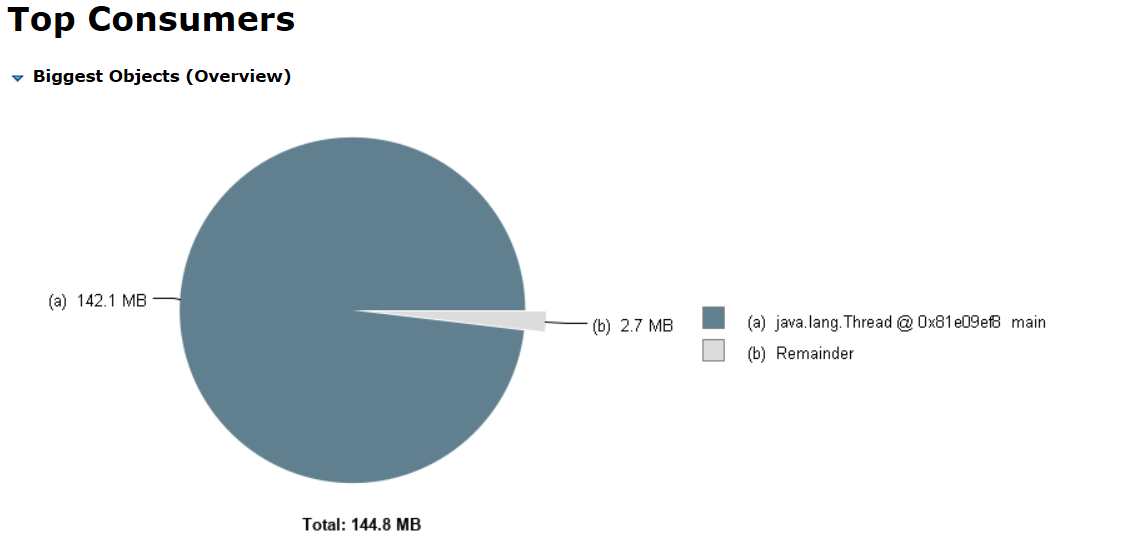
⚫ 查看其 Overview、histogram 视图（当前时刻内存中存储的各类型的实例数量以及所占用内存的情况）、dominator tree 视图（每个实例之所以未被 GC 的原因，即各实例之间的引用关系，以及与 root 之间的引用路径)、top consumers 视图（程序当前时刻的内存占用热点hotspot）。

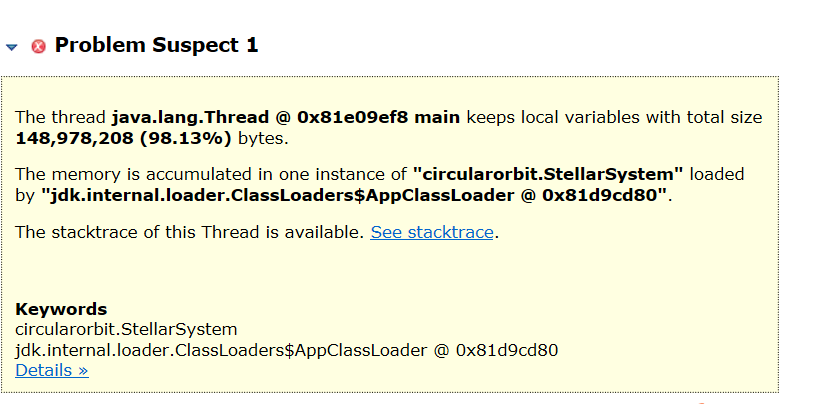
⚫ 查看 leak suspects report，看是否存在可能的内存泄露。







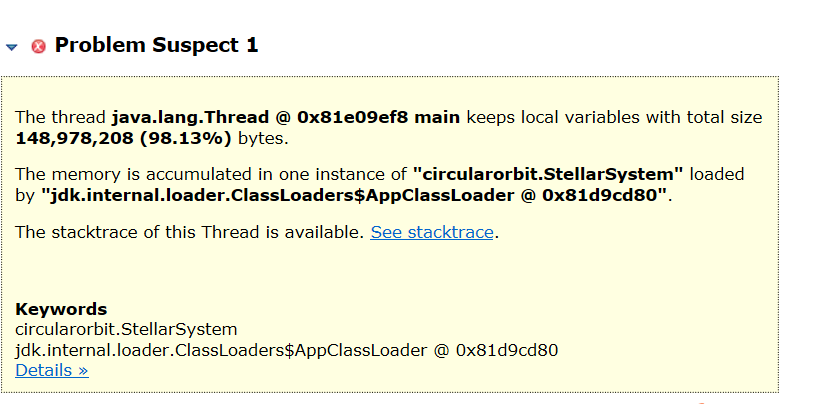




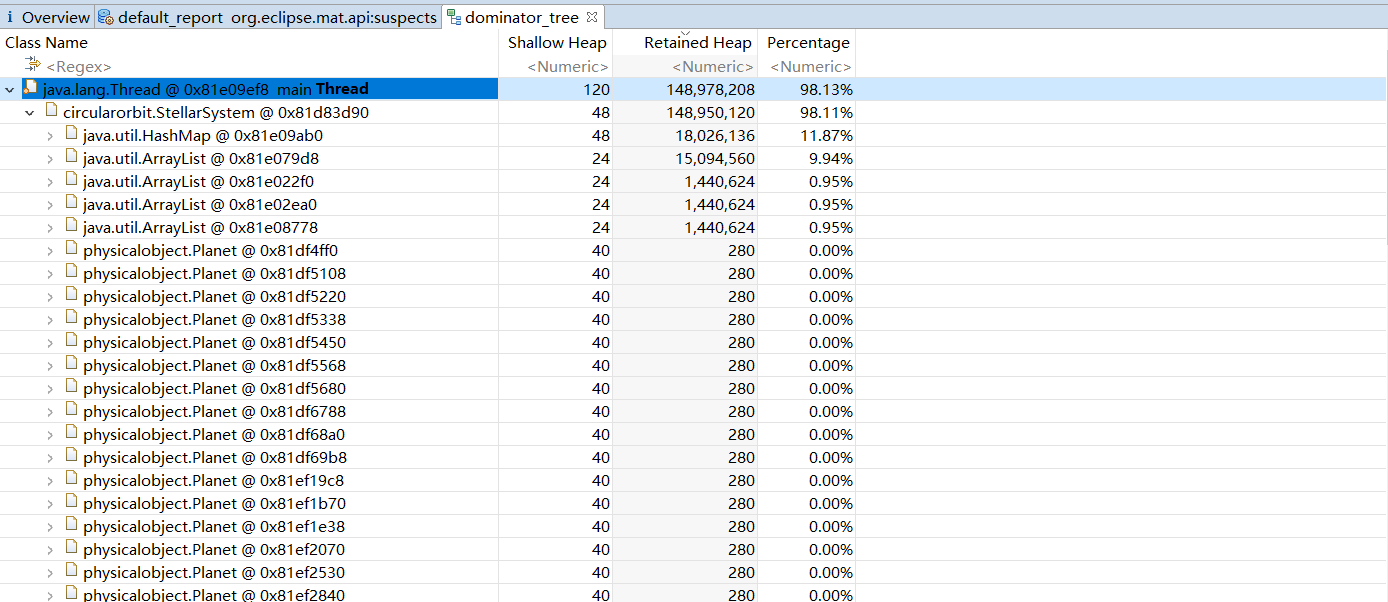
### 发现热点/瓶颈并改进、改进前后的性能对比分析

对热点（执行时间长、占用内存大）的代码区域进行改造优化，重新进行内存导出和 MAT 分析。

首先，修改前面所述的Problem Suspect。



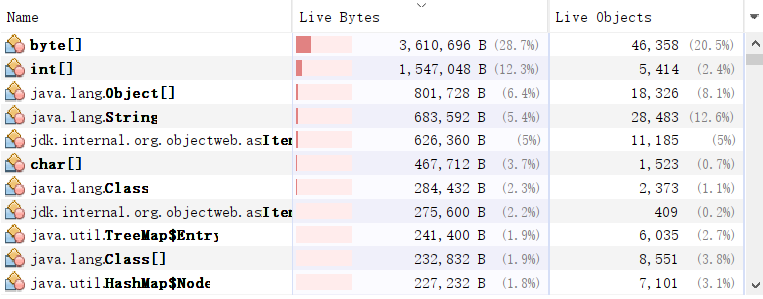
其次，根据histogram 视图（当前时刻内存中存储的各类型的实例数量以及所占用内存的情况）、dominator tree 视图（每个实例之所以未被 GC 的原因，即各实例之间的引用关系，以及与 root 之间的引用路径)、top consumers 视图找出占内存最大部分的内容为程序中的ArrayList，故对其加以改进。



改进前后性能对比：

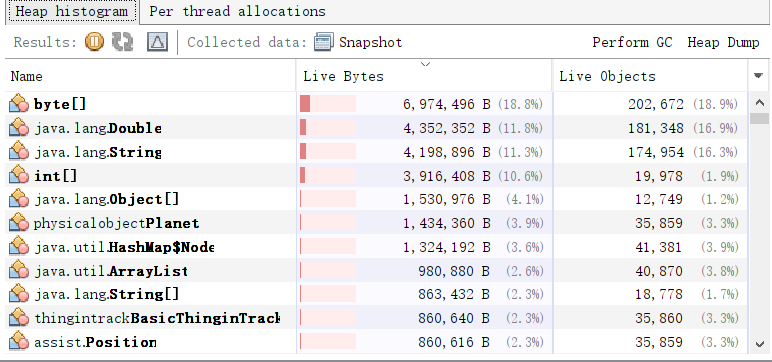
改进前：





改进后：





### 在MAT内使用OQL查询内存导出

在 MAT 中查看内存占用状态（各类实例的分布情况），使用 OQL 查询语言对其进行以下查询：

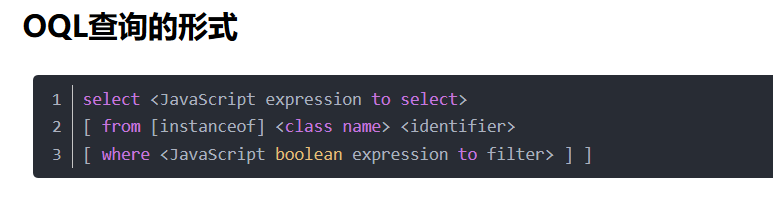
⚫ CircularOrbit 的所有对象实例；

⚫ 大于特定长度 n 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；



解释：

(1)class name是java类的完全限定名，如：java.lang.String, java.util.ArrayList, [C是char数组, [Ljava.io.File是java.io.File[]，依此类推；

(2)类的完全限定名不足以唯一的辨识一个类，因为不同的ClassLoader载入的相同的类，它们在JVM中是不同类型的；

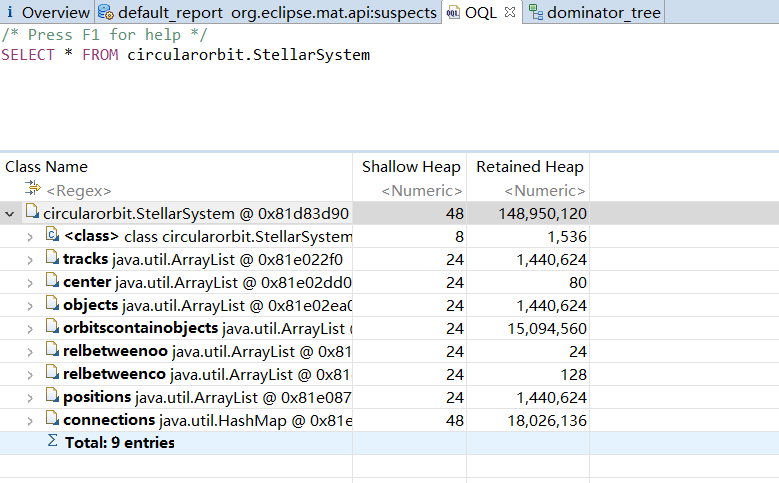
(3)instanceof表示也查询某一个类的子类，如果不明确instanceof，则只精确查询class name指定的类；

(4)from和where子句都是可选的；

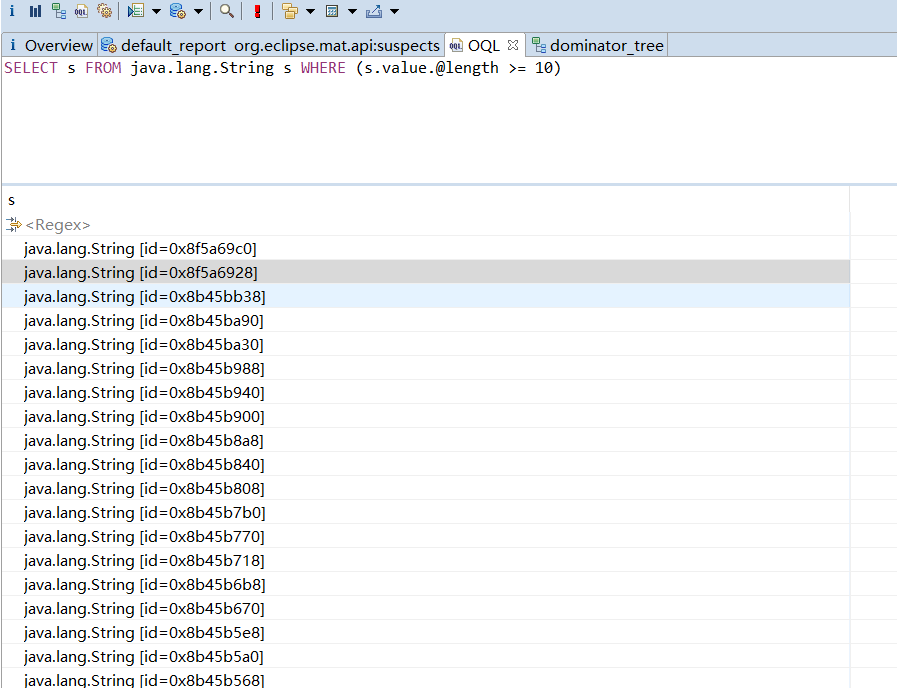
(5)可以使用obj.field\_name语法访问Java字段，并且可以使用array [index]语法访问数组元素。

查询结果：

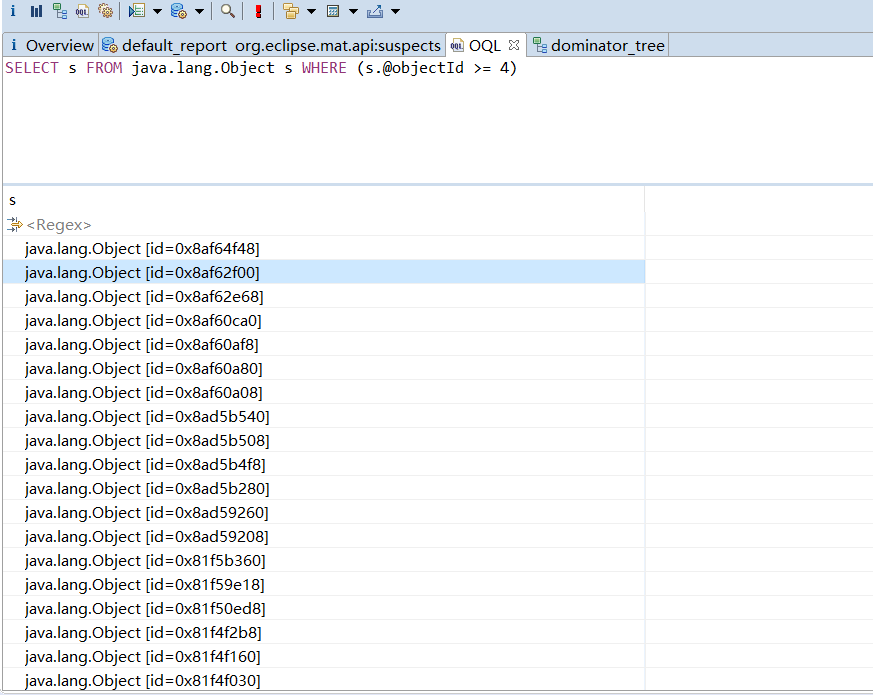
CircularOrbit 的所有对象实例：



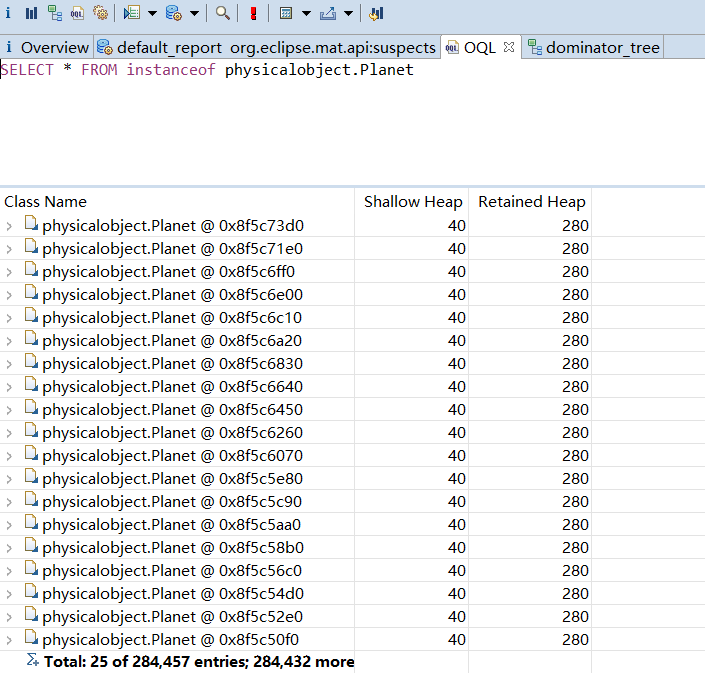
大于特定长度 n 的 String 对象；



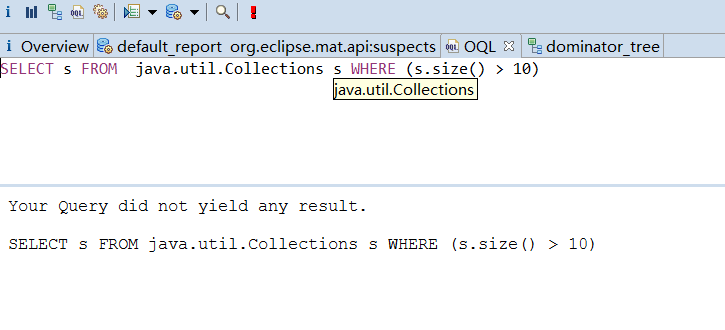
大于特定大小的任意类型对象实例：



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

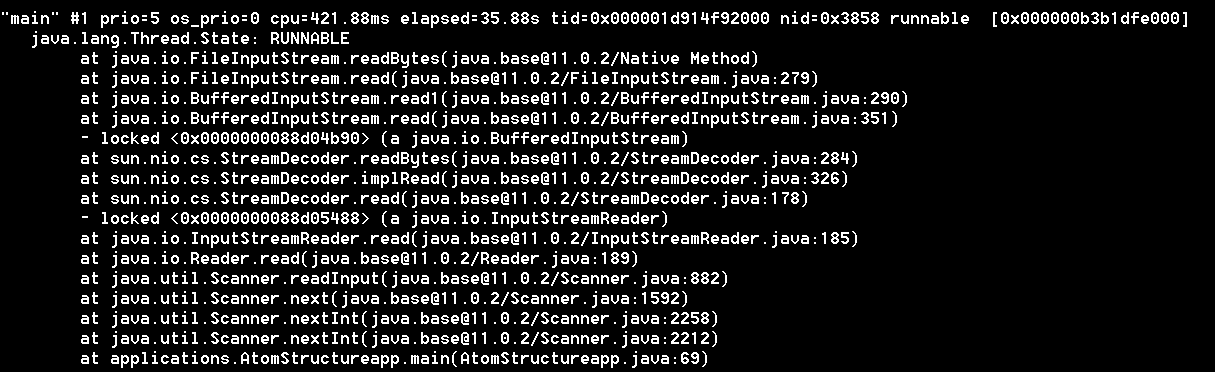
给程序输入若干条操作指令，在每条指令执行时，利用 jstack 或 jcmd导出 java 程序运行时的调用栈（stack trace），观察其中展现出的类和函数调用关系。例如：

⚫ 增加轨道、删除轨道；

⚫ 向特定轨道上增加物体、从特定轨道上删除物体；

⚫ 判断多轨道系统的合法性

读文件：



增加轨道、删除轨道：



向特定轨道上增加物体、从特定轨道上删除物体：



判断多轨道系统的合法性：



### 使用设计模式进行代码性能优化

1.在AtomicStructure应用中，使用Flyweight设计模式实现“电子”：原子核周围的所有电子都是等同的，唯一区别在于其所处的 track；

电子类的设计：只留下电子的基本性质（名称，即字符串“Electronics”）以及相应的实现PhysicalObject接口所需的方法。

/\*\*

\* 一个PhysicalObject接口的具体实现，用于AtomStructure中.

\*

\* **@author** junbaba

\*

\*/

**public** **class** Electronics **implements** PhysicalObject {

**private** **final** String name = "Electronics";

// Abstraction function:

// name对应名称

// Representation invariant:

// name != null

// Safety from rep exposure:

// 通过private使其它类中无法得知本类中的rep

// **TODO** checkRep

**public** **void** checkRep() {

**assert** name != **null**;

}

// **TODO** constructor

**public** Electronics() {

// **TODO** Auto-generated constructor stub

}

@Override

**public** String getname() {

checkRep();

**return** name;

}

@Override

**public** String getsex() {

**throw** **new** UnsupportedOperationException();

}

@Override

**public** Number getages() {

**throw** **new** UnsupportedOperationException();

}

@Override

**public** String getform() {

**throw** **new** UnsupportedOperationException();

}

@Override

**public** String getcolor() {

**throw** **new** UnsupportedOperationException();

}

@Override

**public** Number getplanetr() {

**throw** **new** UnsupportedOperationException();

}

@Override

**public** Number getv() {

**throw** **new** UnsupportedOperationException();

}

@Override

**public** String getdirection() {

**throw** **new** UnsupportedOperationException();

}

@Override

**public** **boolean** equals(Object obj) {

**return** **super**.equals(obj);

}

@Override

**public** String toString() {

StringBuilder str = **new** StringBuilder();

str.append("<" + name + ">");

**return** str.toString();

}

@Override

**public** **int** hashCode() {

// **TODO** Auto-generated method stub

**return** **super**.hashCode();

}

}

工厂方法Efactory：将电子与相应的轨道连接起来：

**package** physicalobject;

**import** thingintrack.ThinginTrack;

**import** track.Track;

**public** **class** Efactory {

/\*\*

\* 将电子与相应轨道连接起来(flyweight模式).

\*/

**public** ThinginTrack<PhysicalObject> gete(PhysicalObject e, Track t) {

ThinginTrack<PhysicalObject> connect = ThinginTrack.*creator*(t, "Basics");

connect.addobject(e);

**return** connect;

}

}

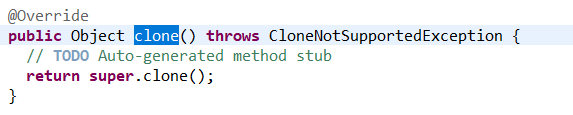
2.基于 Prototype 设计模式的思想改造程序，支持 clone；

改进实现CircularOrbit的接口的类，运用Prototype 设计模式，使其支持clone。（需要注意clone不能是对引用的浅层clone!）

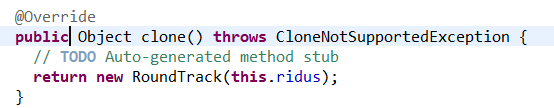




非引用故可直接调用super.clone():



有引用的不可直接调用super.clone():



3.对程序中使用的各种临时性的集合类或其他构造代价高的对象，使用 Singleton设计模式的思想进行改造；

在相应的工厂类中使用Singleton设计模式，使得程序中仅可能存在一个工厂类，从而节省内存空间：

**package** physicalobject;

**import** thingintrack.ThinginTrack;

**import** track.Track;

**public** **class** Efactory {

**private** **static** Efactory *factory*;

**public** **static** Efactory getfactory() {

**if** (*factory* == **null**)

*factory* = **new** Efactory();

**return** *factory*;

}

/\*\*

\* 将电子与相应轨道连接起来(flyweight模式).

\*/

**public** ThinginTrack<PhysicalObject> gete(PhysicalObject e, Track t) {

ThinginTrack<PhysicalObject> connect = ThinginTrack.*creator*(t, "Basics");

connect.addobject(e);

**return** connect;

}

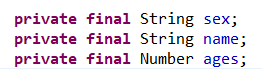
}



4.对程序中使用的各种简单对象，使用 canonicalization 的思想对其进行改造，降低构造对象的数量；

用int取代一些可以替代的常量（ canonicalization 的思想）：

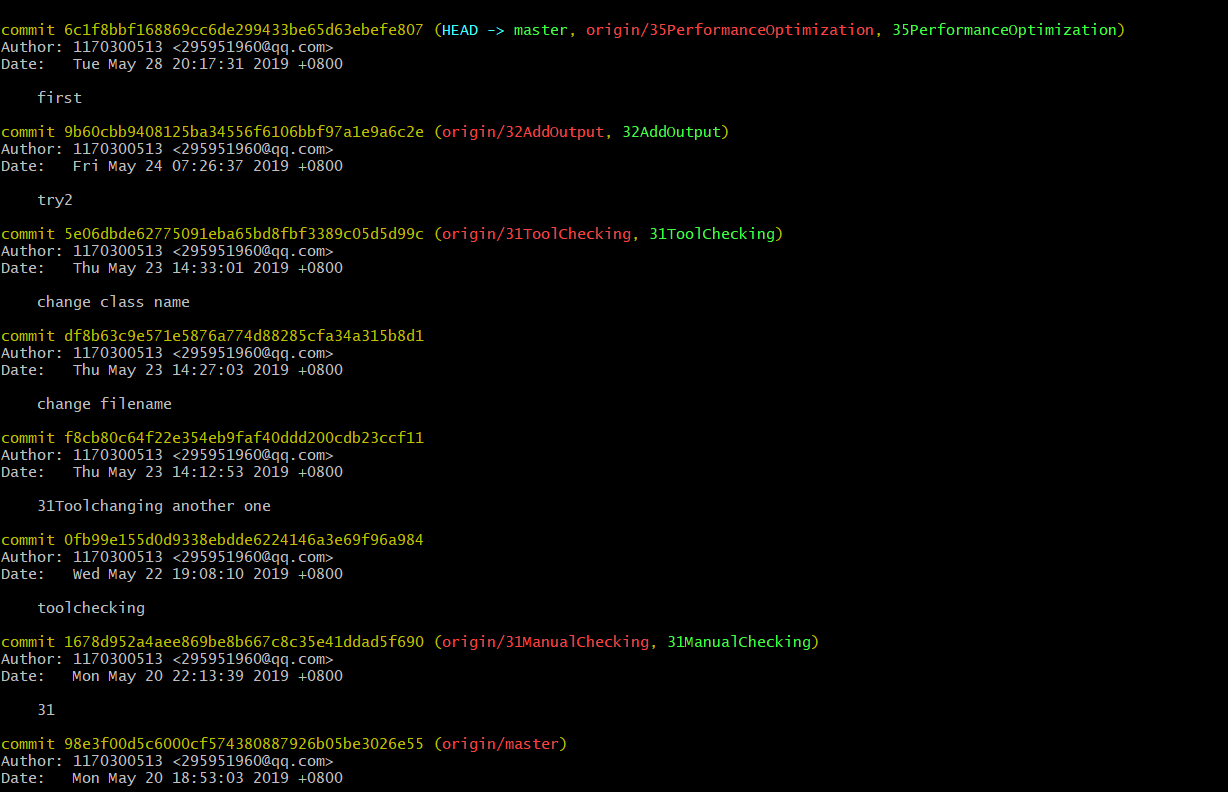
取代前：



取代后：



## 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 |
| 2019-05-20 | 15:45-18:00 | 阅读实验报告要求及Google代码规范 | 按时完成 |
| 2019-05-21 | 13:00-15:00 | 人工修改代码格式 | 按时完成 |
| 2019-05-22 | 18:00-22:00 | 利用checkstyle辅助修改代码格式 | 按时完成 |
| 2019-05-23 | 18:00-20:00 | 写三种I/O策略 | 按时完成 |
| 2019-05-24 | 18:00-22:00 | 修改原来读文件的方式并对I/O策略进行相应的测试 | 按时完成 |
| 2019-05-25 | 15:45-18:00 | 学习jstat、jmap并完成3.3中内容 | 部分未完成 |
| 2019-05-26 | 18:00-23:00 | 完成昨日欠缺并学习3.4中工具并完成3.4 | 按时完成 |
| 2019-05-27 | 18:00-23:00 | 学习使用MAT并完成3.5 | 设计模式未完成 |
| 2019-05-28 | 18:00-19:00 | 完成设计模式 | 按时完成 |

# 实验过程中遇到的困难与解决途径

|  |  |
| --- | --- |
| 遇到的难点 | 解决途径 |
| Checkstyle中包含大量Tab类格式错误，数量过大以至于修改费时 | 网上下载并向Eclipse中导入Google代码规范文件，使得Eclipse每次保存都会自动修改，大大减少了修改的时间。 |
| 不会使用jstat、jmap工具 | 上网查询相应的方法。 |
|  |  |

# 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

1. 不懂得知识要多上CSDN寻找相应的内容，并且大胆进行实验，例如本次的VisualVM的使用，其实并不难，但未知的恐惧还是让我犹豫了太久！
2. 可以尝试各种新工具，比如本次实验中用Google代码规范文件快速修改代码中的格式。

## 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？

取决于自己的目的：如果是要让程序的可读性更强，则要更偏向于“看起来很美”；如果是为了追求运行速度，则更倾向于“运行起来很美”。我更愿意让代码“运行起来很美”（反正除了TA也没其他人读我代码嘿嘿嘿）

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？

SpotBugs能找出潜在的bug，确实能够有效地改进代码质量；CheckStyle仅仅只辅助改进格式，我认为对改善代码质量的作用非常有限。

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？

JDK1.0的时候，所有与输入相关的类都继承于InputStream，所有与输出相关的类都继承于OutputStream；JDK1.1的时候，增加了面向字符的IO类，包括Reader和Writer；JDK1.4的时候，增加了NIO，实际上旧的IO已经使用重新使用NIO实现过了，即使没有显示的使用NIO编码，我们也能从中受益的。

其中InputStreamReader和OutputStreamWriter是比较特殊的类，它们可将字节流转换为字符流，因而称为转换流；BufferedInputStream,BufferedOutputStream的用法虽然和FileInputStream,FileOutputStream的用法一样，但是效率却相差很大，因为内存的效率比IO操作的效率要高得多；ObjectInputStream,ObjectOutputStream则分别是将序列化的对象(即实现了Serializable接口的对象)读取出来/写入到文件中，显然，这其实是利用反射的原理；Reader与InputStream的区别在于一个是字符输入流，一个是字节输入流。FileReader与FileInputStream对应；BufferedReader与BufferedInputStream对应，所以在读取文本文件时最好使用BufferedReader而不要使用FileReader.

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？

内存管理方式大致相同，但JVM添加了自主的垃圾回收机制。这样避免了使用者手动回收垃圾的麻烦，解放了使用者。我认为它有好处也有坏处，好处是降低了java程序员的编写代码的难度与工作量，坏处是使程序的性能有所下降。

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？

JVM自动的垃圾回收机制。这样避免了使用者手动回收垃圾的麻烦，解放了使用者。降低了java程序员的编写代码的难度与工作量，坏处是使程序的性能有所下降。C++的垃圾回收机制，虽然提升程序性能，但增加了程序员编写代码的难度与工作量。

我认为还有改进空间：如何选择更佳的时间点进行垃圾回收，如何更好更快地进行垃圾回收，都是可以继续改进的地方。

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？

是！我的程序中提升了近15%的性能！

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？

能，它们能便利地发现程序热点，针对热点进行优化，使得我们修改代码提升性能是有的放矢，并且能去的较为良好的改进结果。

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？

通过实验的锻炼，我能说：有！

1. 关于本实验的工作量、难度、deadline。

工作量不大，但需要自学运用多门工具，难度较高，deadline还行。

1. 到目前为止，你对《软件构造》课程的意见与建议。

非常严肃：希望能提供一些MAT的使用手册，否则自己通过搜索引擎查询所得到的的结果真的都是糟粕！！