

**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 | 罗瑞欣 |
| 学号 | 1170300821 |
| 班号 | 1703008 |
| 电子邮件 | [3102595709@qq.com](mailto:3102595709@qq.com) |
| 手机号码 | 15048260039 |

**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

IDEA, jdk9, guava-jdk5

在这里给出你的GitHub Lab5仓库的URL地址（Lab5-学号）。

git@github.com:ComputerScienceHIT/Lab5-1170300821.git

# 实验过程

## Static Program Analysis

### 人工代码走查（walkthrough）

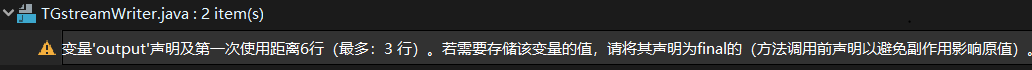
### 使用CheckStyle和SpotBugs进行静态代码分析

1. 缩进问题，例如：





1. 对象声明



1. 缺少javadoc



## 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。

读文件方面

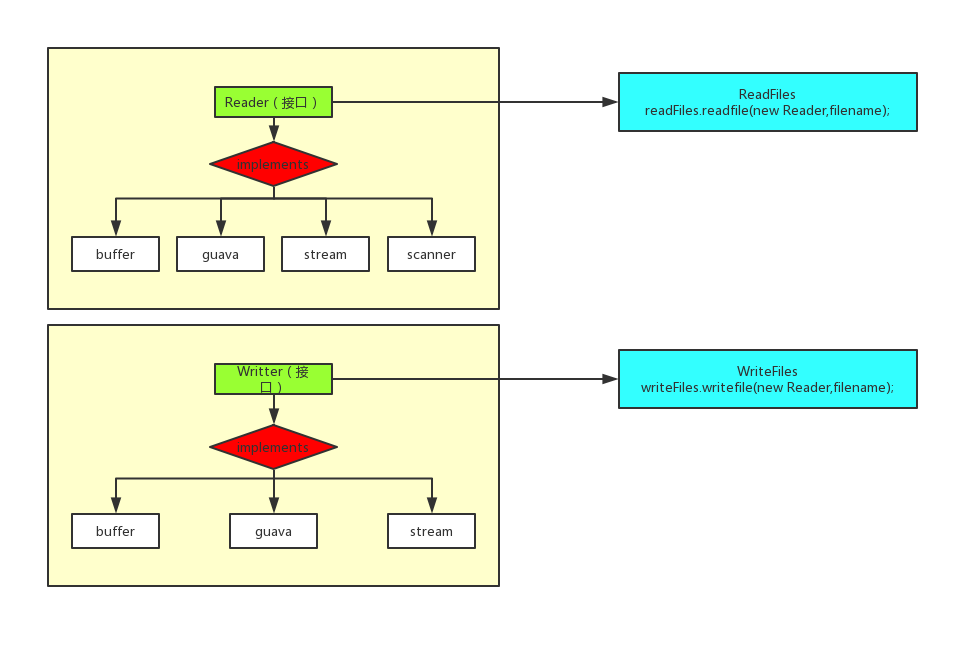
|  |  |
| --- | --- |
| BufferReader | new BufferedReader(new FileReader(filename))  按行读入 |
| Stream | FileInputStream inputStream = new FileInputStream(filename);  BufferedReader bfReader = new BufferedReader(new InputStreamReader(inputStream));  按行读入 |
| Scanner | String encoding = "utf-8";  Scanner scanner = new Scanner(file, encoding);  scanner.hasNextLine()  按行读入 |
| Guava | Files.asCharSource(new File(filename), Charset.defaultCharset())  .readLines(new LineProcessor<String>() {public boolean processLine(String line) throws IOException {if (line != null){    按行读入 |

写文件

|  |  |
| --- | --- |
| BufferWritter | BufferedWriter bfWriter = *new* BufferedWriter(*new* FileWriter(filename));  bfWriter.write  bfWriter.newLine();  bfWriter.flush(); bfWriter.close(); |
| Stream | FileOutputStream output = *new* FileOutputStream(filename);  String newLine = System.getProperty("line.separator");//用于换行  stringBuilder.append()  output.write(stringBuilder.toString().getBytes()); output.close(); |
| Guava | *final* File newFile = *new* File(filename);  stringBuilder.append()  String newLine = System.*getProperty*("line.separator");*//用于换行*  Files.*write*(stringBuilder, newFile, Charsets.*UTF\_8*); |

如何用strategy设计模式实现在多种I/O策略之间的切换。

Package iostrategy中：



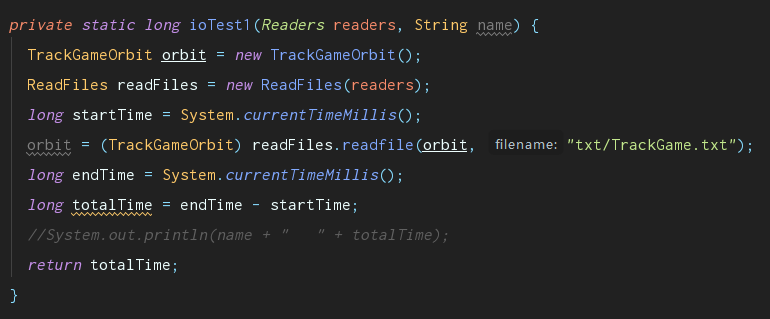
### 多种I/O实现方式的效率对比分析

如何收集你的程序I/O语法文件的时间：

在iostrategy.IoTest中，使用静态方法，分别调用十三个读写方法各十次，将测试平均结果计入文件txt/test.txt中







表格方式对比不同I/O的性能（单位：毫秒）：

TrackGame的读：

|  |  |
| --- | --- |
| BufferReader | 857 |
| Guava | 827 |
| Scanner | 1028 |
| Stream | 797 |

TrackGame的写：

|  |  |
| --- | --- |
| BufferWritter | 463 |
| Stream | 445 |
| Guava | 328 |

SocialNetworkCircle的读：

|  |  |
| --- | --- |
| BufferReader | 3877 |
| Guava | 3132 |
| Stream | 2853 |

SocialNetworkCircle的写：

|  |  |
| --- | --- |
| BufferWritter | 558 |
| Stream | 603 |
| Guava | 640 |

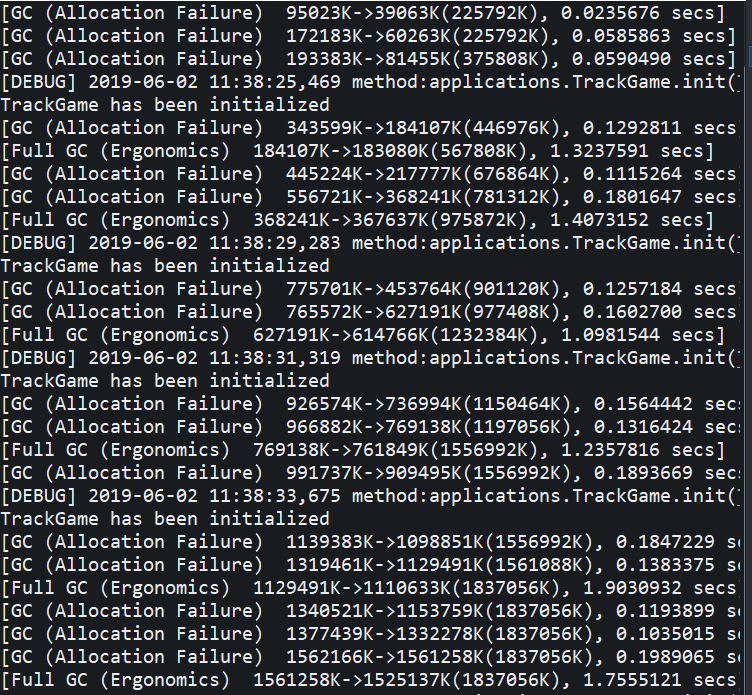
图形对比不同I/O的性能（单位：毫秒）。

## Java Memory Management and Garbage Collection (GC)

### 使用-verbose:gc参数

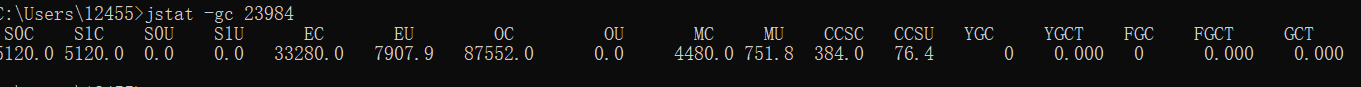
在虚拟机发生内存回收时在输出设备显示信息，这里设置成反复读文件，以便观察。

箭头前后的数据95023K和39063K分别表示垃圾收集GC前后所有存活对象使用的内存容量，说明有95023K-39063K=55960K的对象容量被回收，括号内的数据124096K为堆内存的总容量，收集所需要的时间是0.0235676秒（这个时间在每次执行的时候会有所不同）

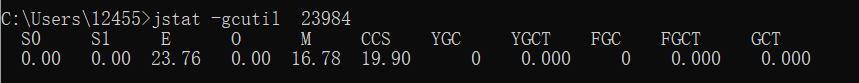


### 用jstat命令行工具的-gc和-gcutil参数

监视Java堆状况，包括Eden区、2个Survivor区、老年代、永久代等的容量



监视内容与-gc基本相同，但输出主要关注已使用空间占总空间的百分比



### 使用jmap -heap命令行工具

打印heap的概要信息，GC使用的算法，heap的配置及wise heap的使用情况.

using parallel threads in the new generation. ##新生代采用的是并行线程处理方式

using thread-local object allocation.

Concurrent Mark-Sweep GC ##同步并行垃圾回收

Heap Configuration: ##堆配置情况

MinHeapFreeRatio = 0 ##最小堆使用比例

MaxHeapFreeRatio = 100 ##最大堆可用比例

MaxHeapSize = 2147483648 (2048.0MB) ##最大堆空间大小

NewSize = 268435456 (256.0MB) ##新生代分配大小

MaxNewSize = 268435456 (256.0MB) ##最大可新生代分配大小

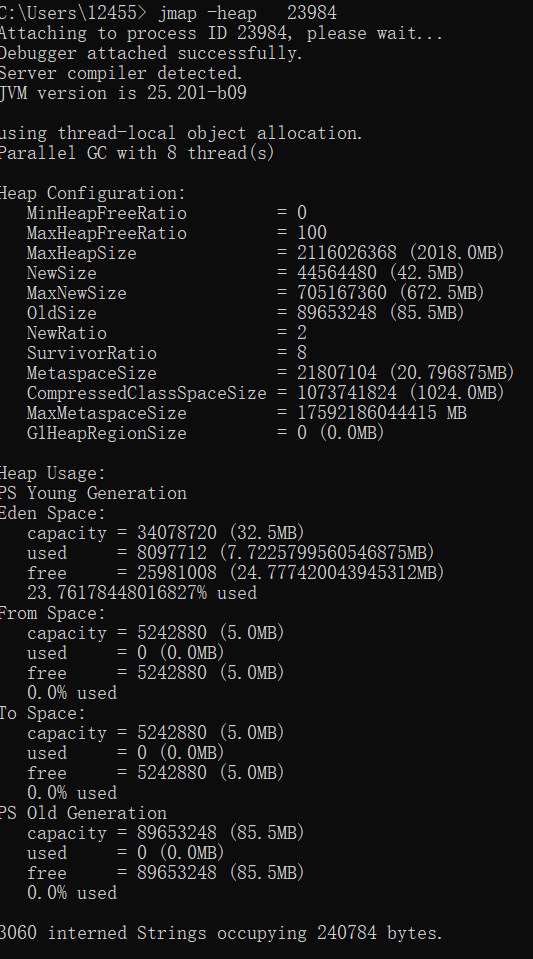
OldSize = 5439488 (5.1875MB) ##老生代大小

NewRatio = 2 ##新生代比例

SurvivorRatio = 8 ##新生代与suvivor的比例

PermSize = 134217728 (128.0MB) ##perm区大小

MaxPermSize = 134217728 (128.0MB) ##最大可分配perm区大小



Heap Usage: ##堆使用情况

New Generation (Eden + 1 Survivor Space):##新生代

capacity = 241631232 (230.4375MB) ##伊甸区容量

used = 77776272 (74.17323303222656MB) ##已经使用大小

free = 163854960 (156.26426696777344MB) ##剩余容量

32.188004570534986% used ##使用比例

Eden Space: ##伊甸区

capacity = 214827008 (204.875MB) ##伊甸区容量

used = 74442288 (70.99369812011719MB) ##伊甸区使用

free = 140384720 (133.8813018798828MB) ##伊甸区当前剩余容量

34.65220164496263% used ##伊甸区使用情况

From Space: ##survior1区

capacity = 26804224 (25.5625MB) ##survior1区容量

used = 3333984 (3.179534912109375MB) ##surviror1区已使用情况

free = 23470240 (22.382965087890625MB) ##surviror1区剩余容量

12.43827838477995% used ##survior1区使用比例

To Space: ##survior2 区

capacity = 26804224 (25.5625MB) ##survior2区容量

used = 0 (0.0MB) ##survior2区已使用情况

free = 26804224 (25.5625MB) ##survior2区剩余容量

0.0% used ## survior2区使用比例

concurrent mark-sweep generation: ##老生代使用情况

capacity = 1879048192 (1792.0MB) ##老生代容量

used = 30847928 (29.41887664794922MB) ##老生代已使用容量

free = 1848200264 (1762.5811233520508MB) ##老生代剩余容量

1.6416783843721663% used ##老生代使用比例

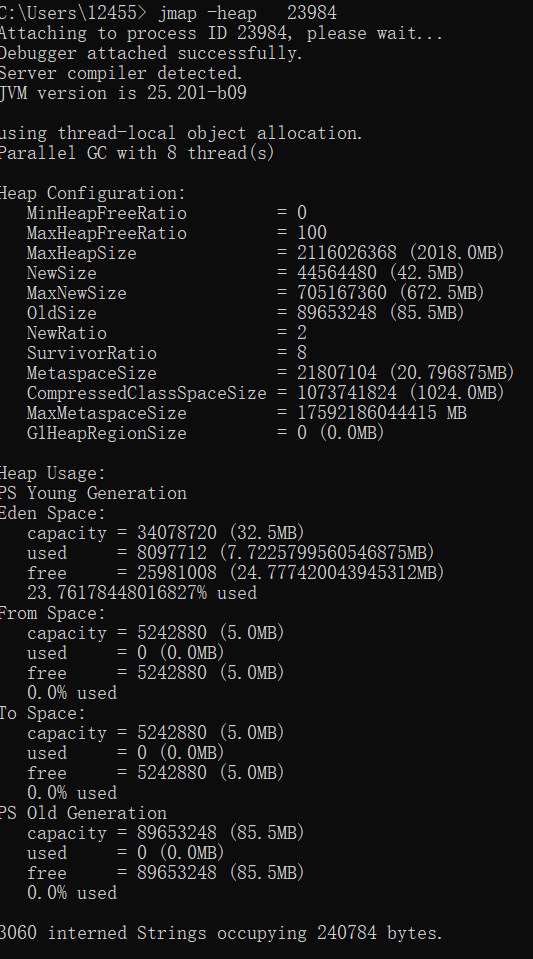
Perm Generation: ##perm区使用情况

capacity = 134217728 (128.0MB) ##perm区容量

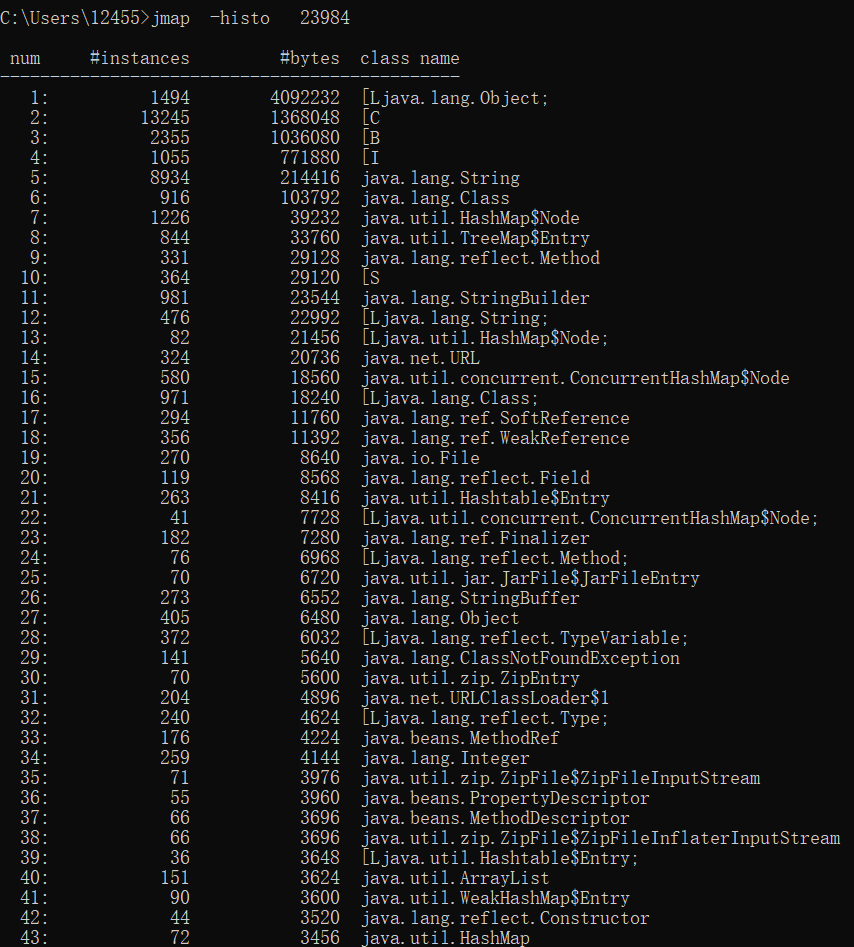
used = 47303016 (45.111671447753906MB) ##perm区已使用容量

free = 86914712 (82.8883285522461MB) ##perm区剩余容量

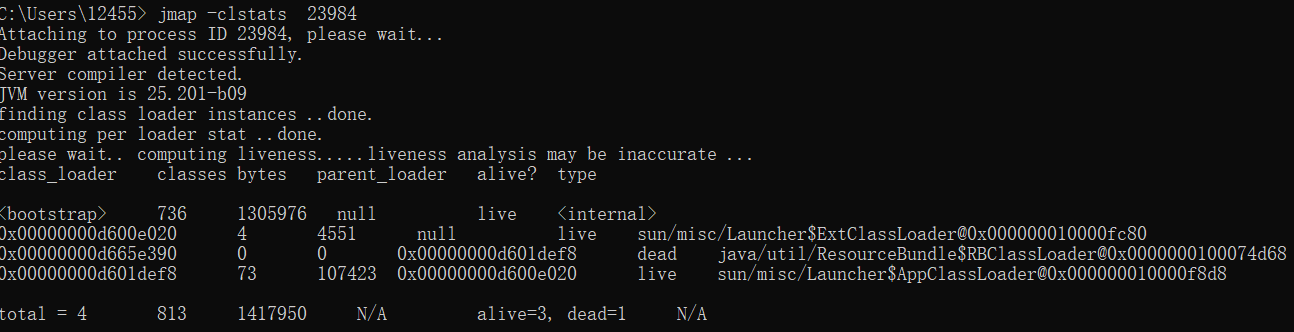
35.24349331855774% used ##perm区使用比例



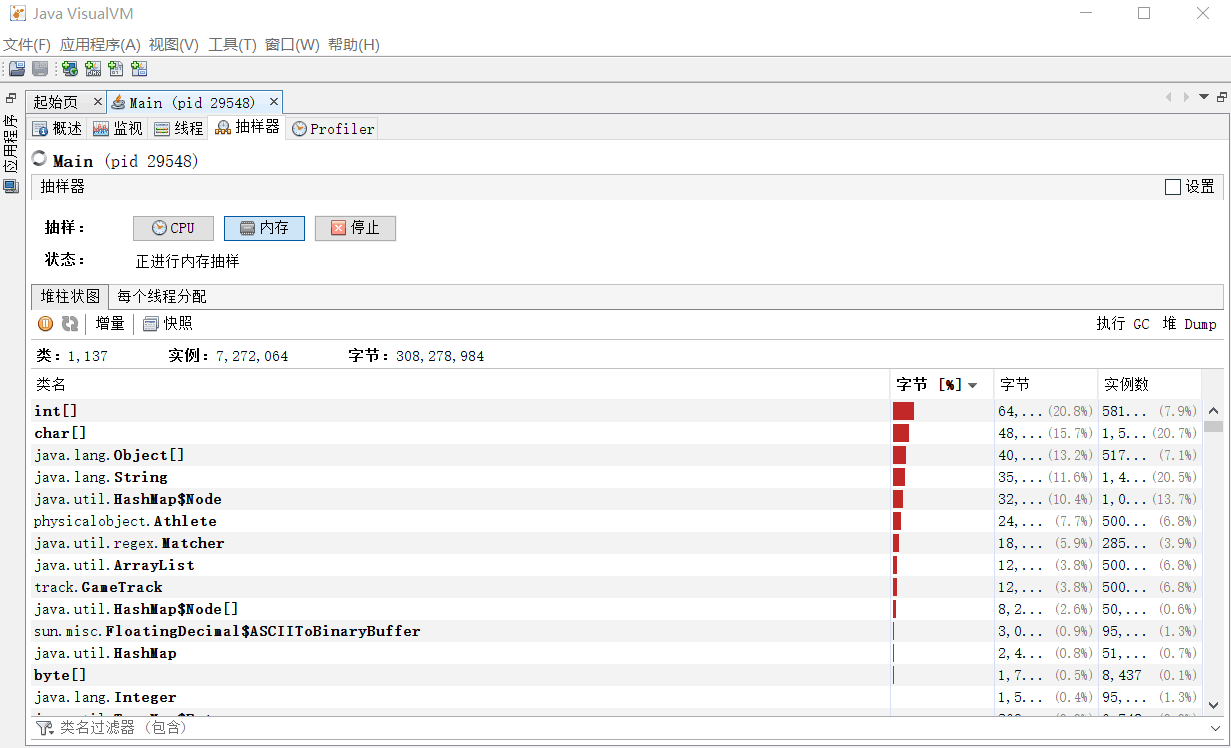
打印每个class的实例数目,内存占用,类全名信息. VM的内部类名字开头会加上前缀”\*”. 如果live子参数加上后,只统计活的对象数量.

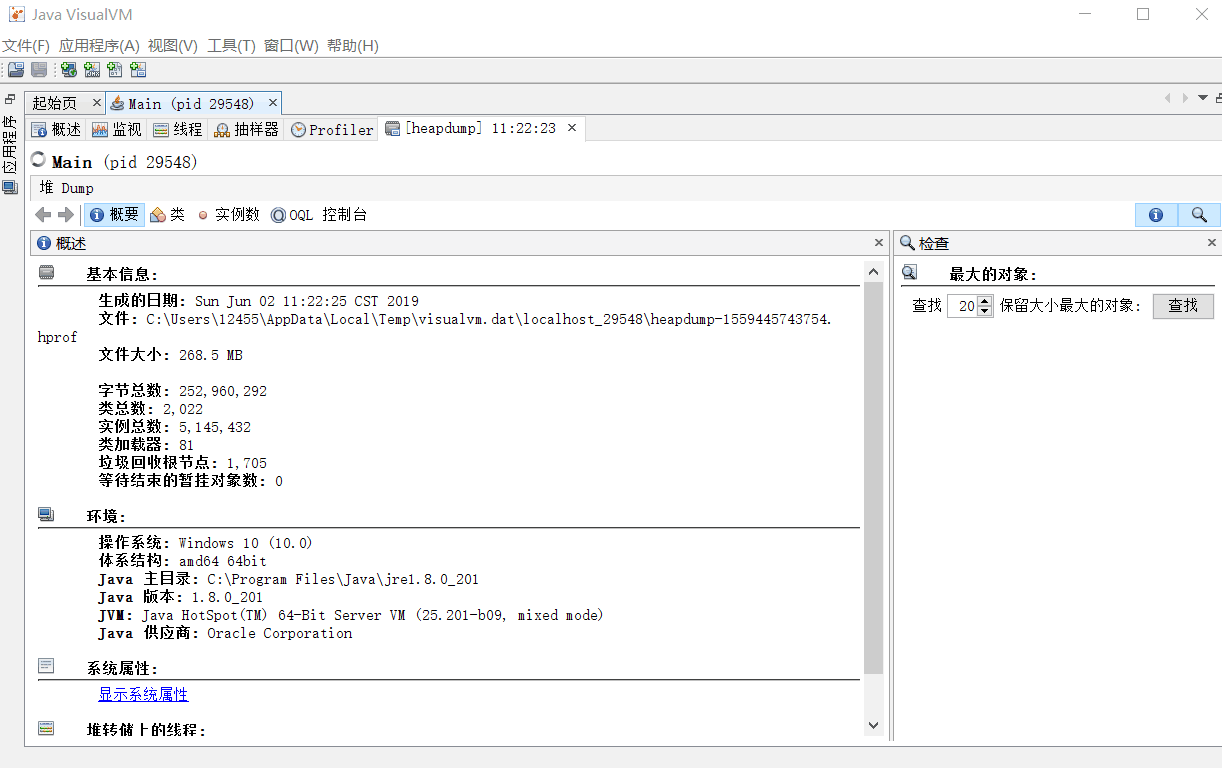


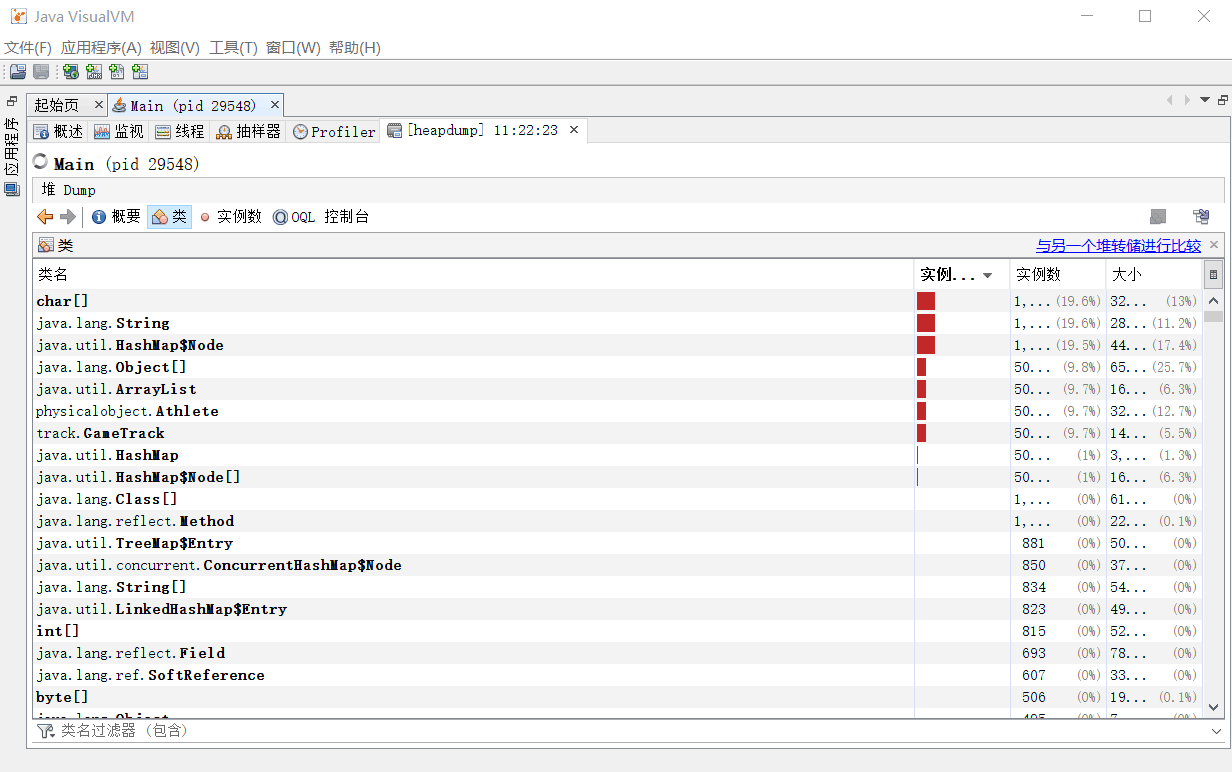
### 使用jmap -clstats命令行工具

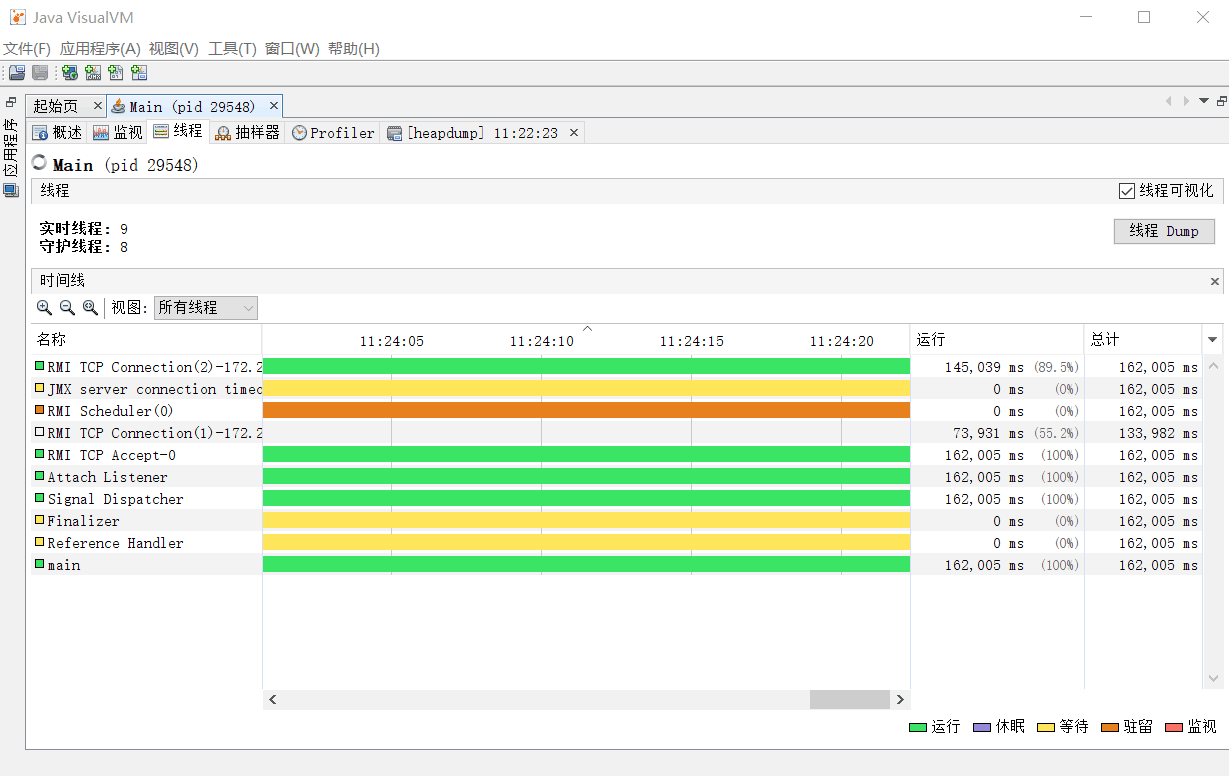


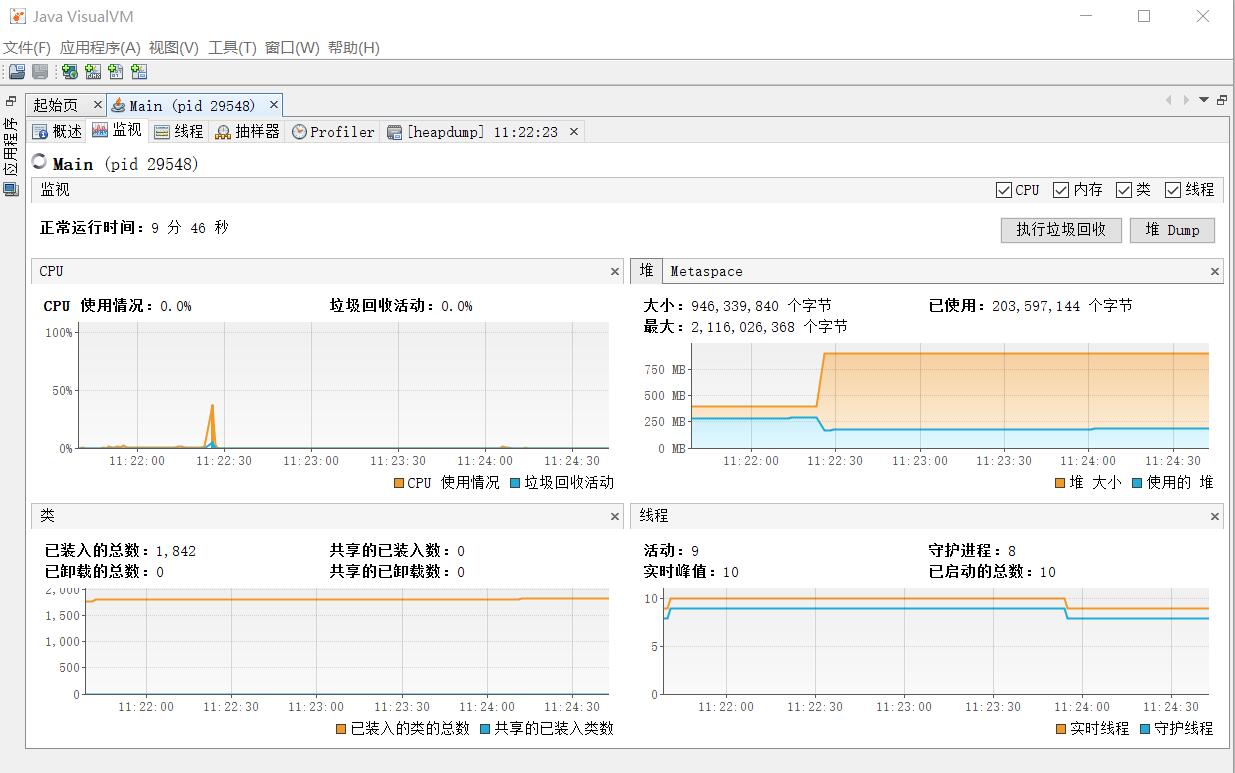
### 使用JMC/JFR、jconsole或VisualVM工具











### 分析垃圾回收过程

JAVA通过可达性分析算法判定对象是否存活的，基本思想是定义一个GC Root对象,以GC Root为起点，向下搜索，走过的路径称作引用链，当一个对象到GC Root之间没有任何的引用链可以连接时，表示这个对象可以被回收了。

在JAVA中，可以作为GC Root的对象包括以下几种：

1.虚拟机栈中引用的对象。

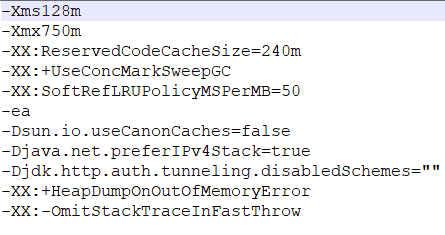
2.方法区中类静态属性引用的对象。

3.方法区中常量引用的对象。

4.本地方法栈中（Native方法）引用的对象。

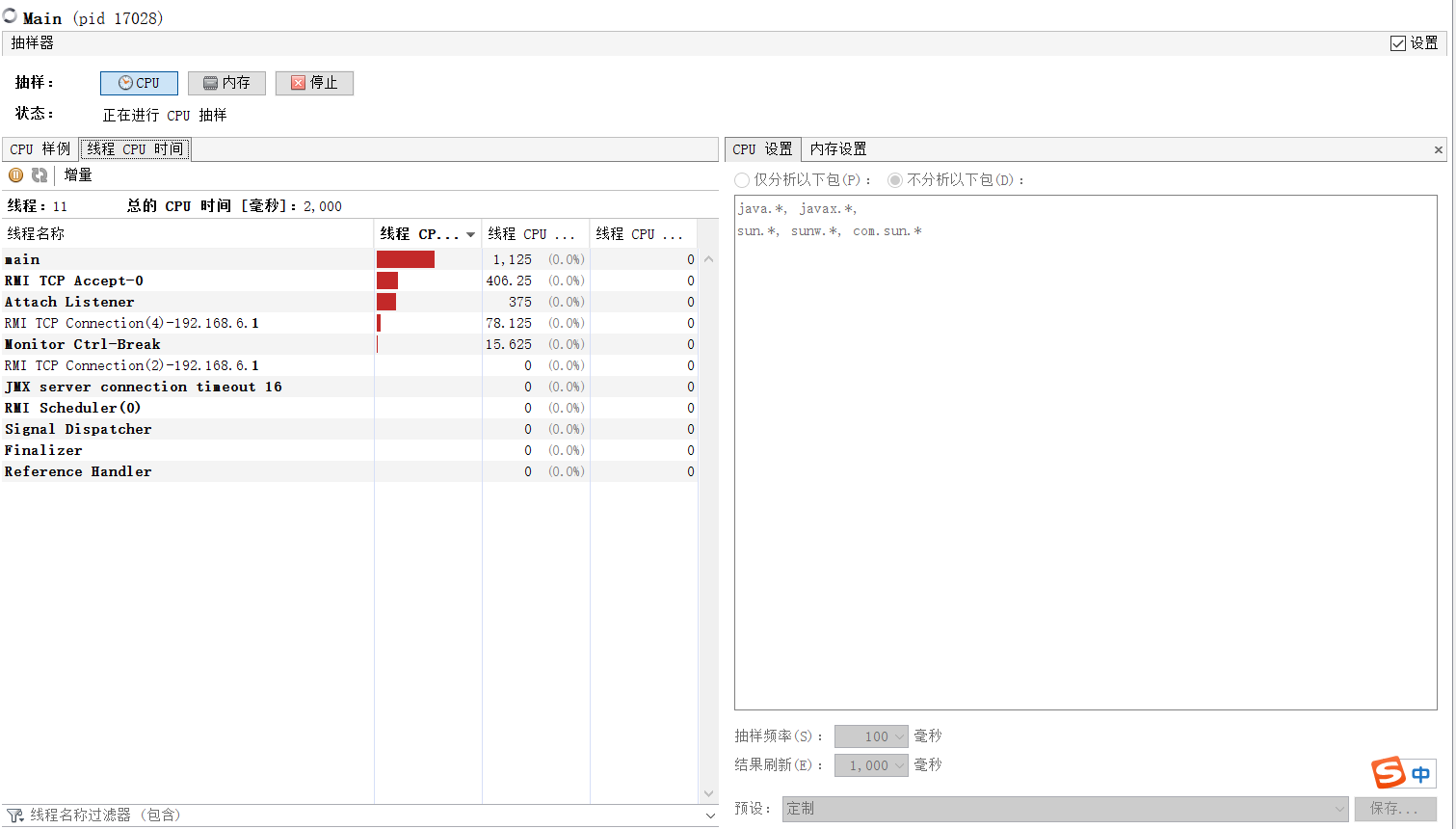
在虚拟机通过可达性分析算法进行内存”死亡“状态判断时，就可以避免上面提到过的循环引用的问题了，所以主流的虚拟机都会采用这种方式来进行垃圾回收的第一个步骤。

### 配置JVM参数并发现优化的参数配置

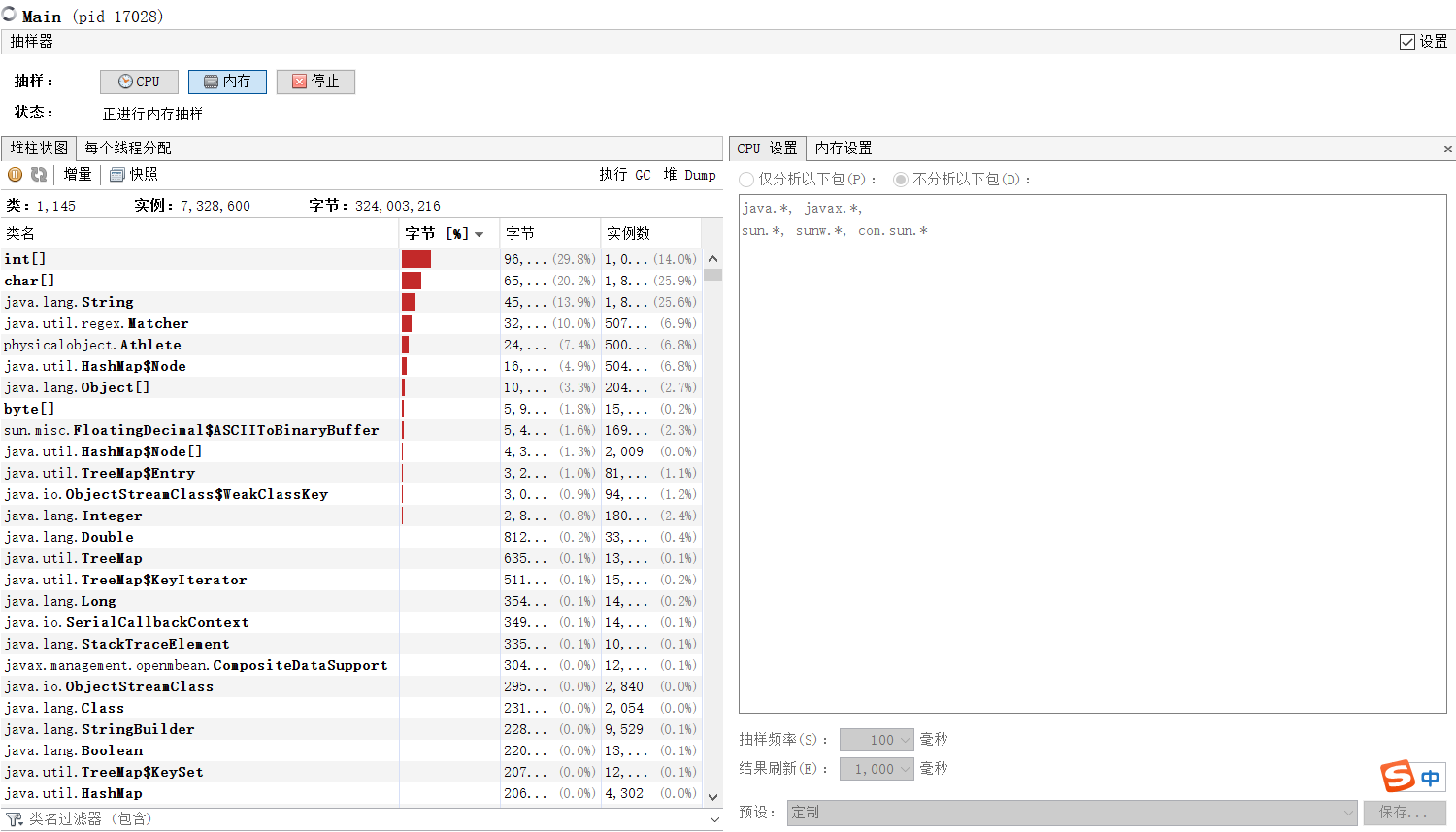


## Dynamic Program Profiling

### 使用JMC或VisualVM进行CPU Profiling

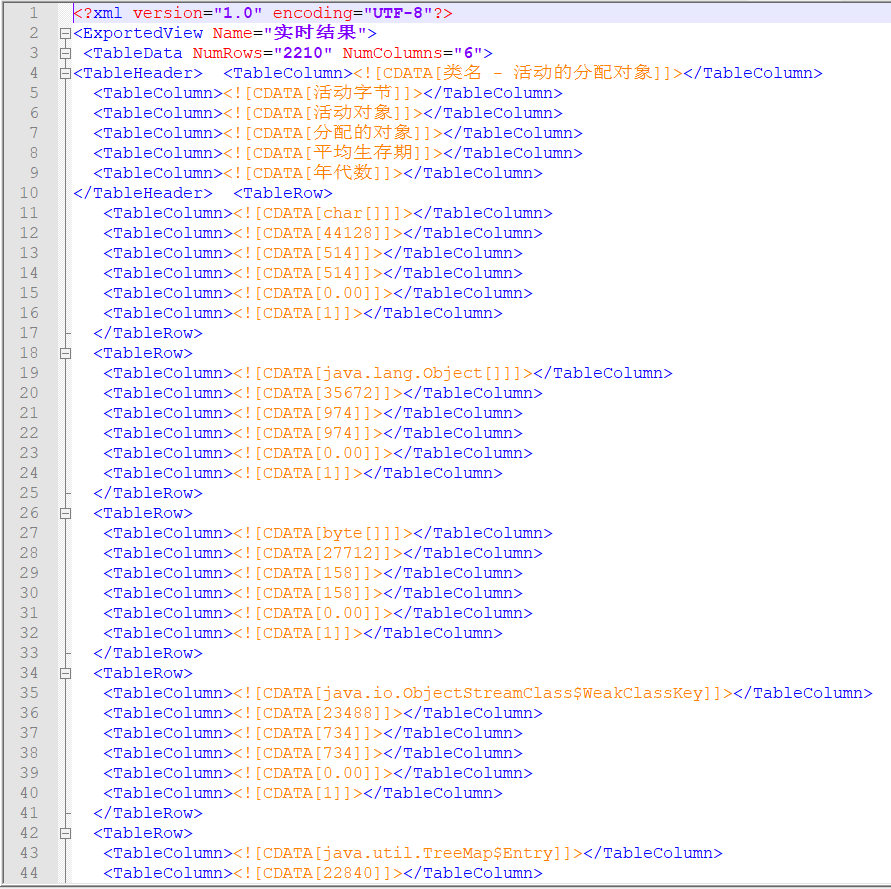


### 使用VisualVM进行Memory profiling

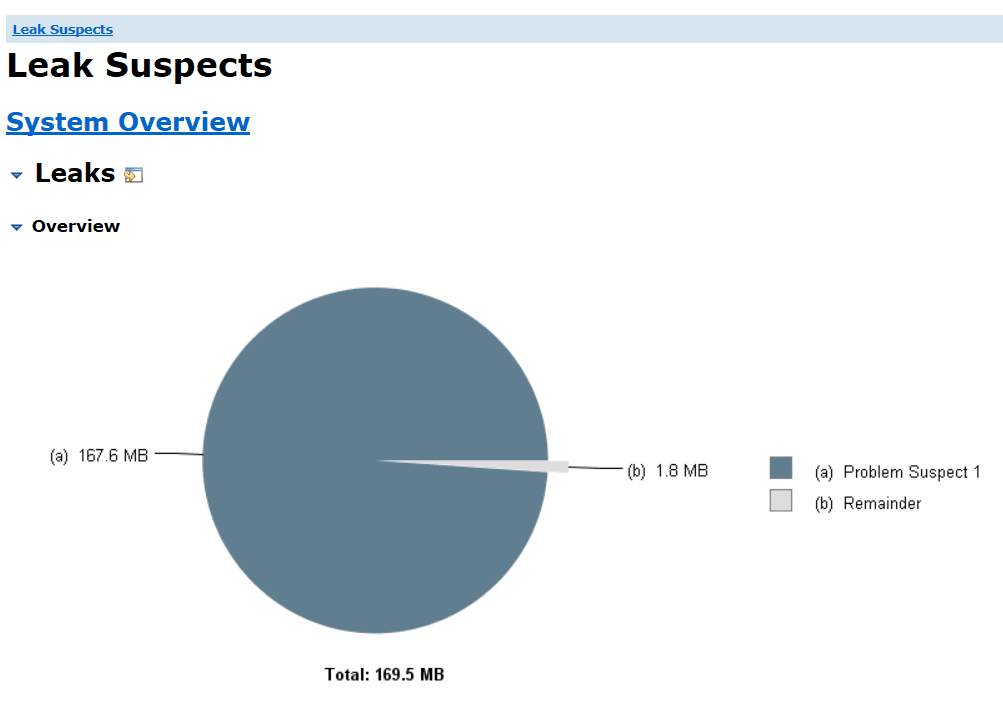


## Memory Dump Analysis and Performance Optimization

### 内存导出



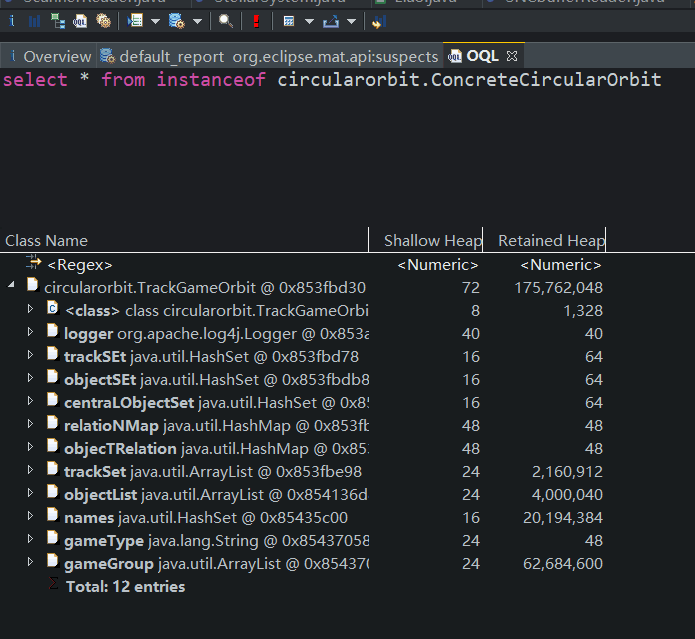
### 使用MAT分析内存导出文件



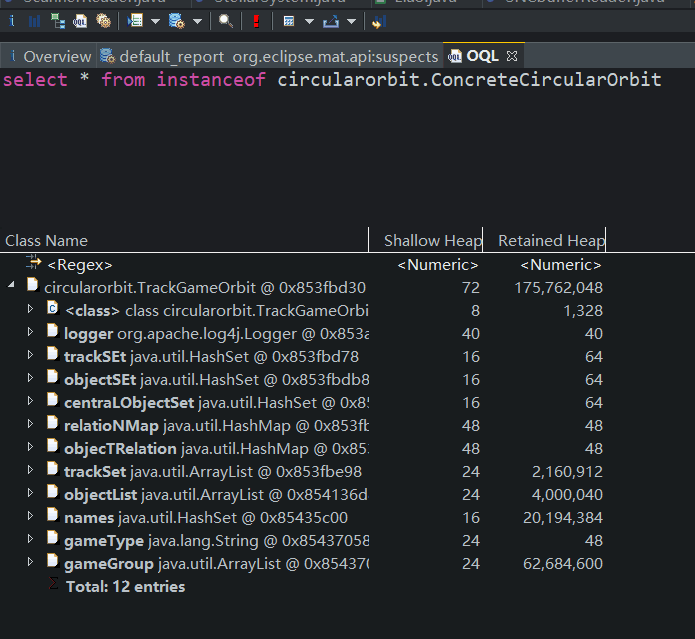
### 发现热点/瓶颈并改进、改进前后的性能对比分析

### 在MAT内使用OQL查询内存导出

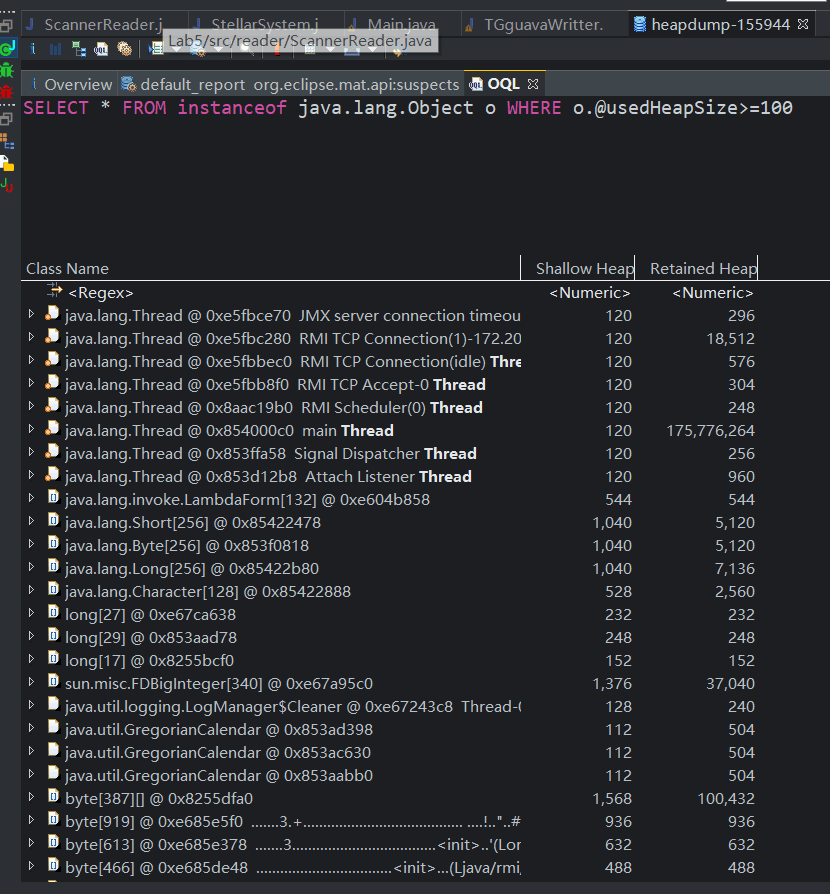
CircularOrbit 的所有对象实例



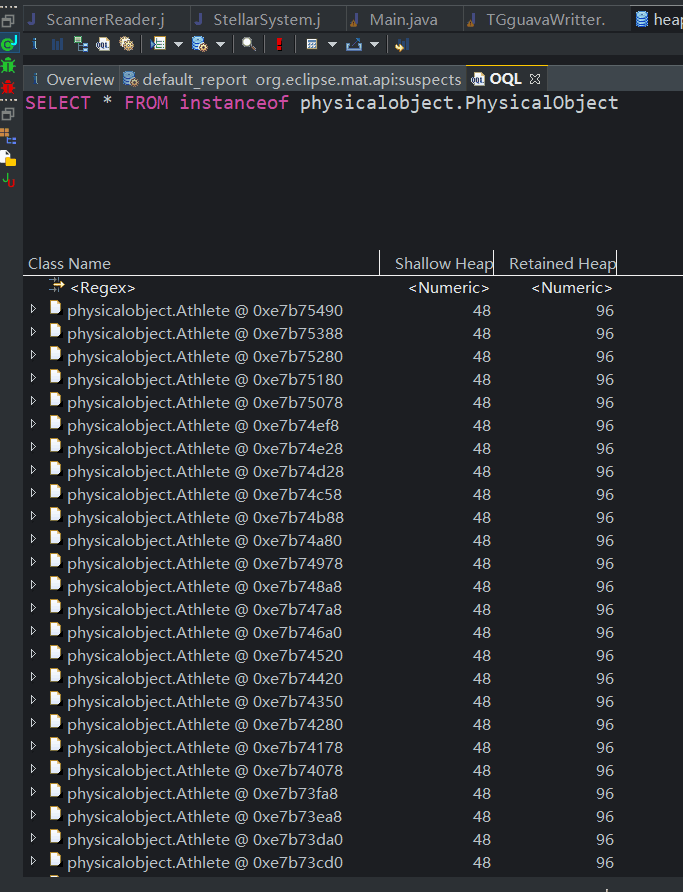
大于特定长度 n 的 String 对象



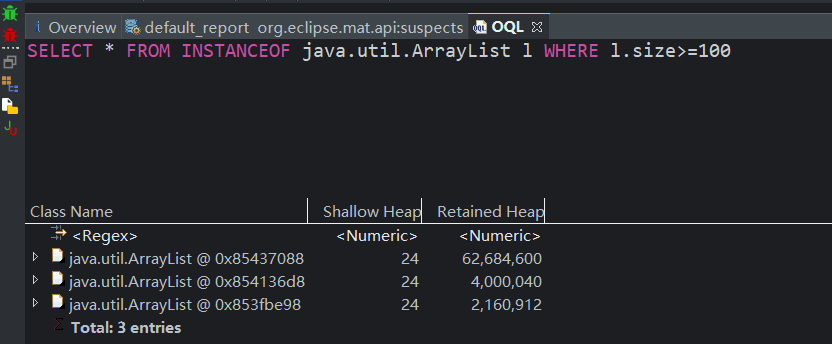
大于特定大小的任意类型对象实例



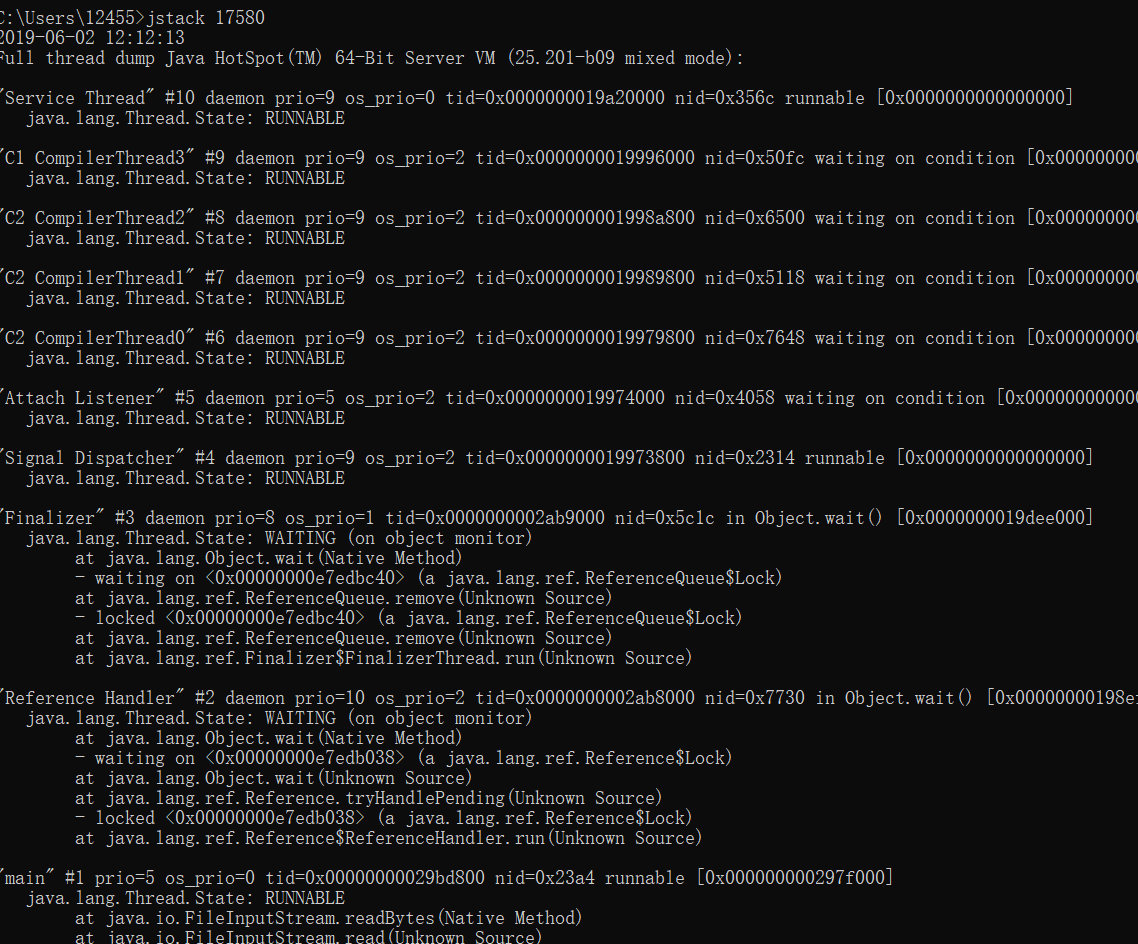
PhysicalObject（及其子类）的对象实例的数量和总占用内存大小



所有包含元素数量大于 100 的 Collections 实例



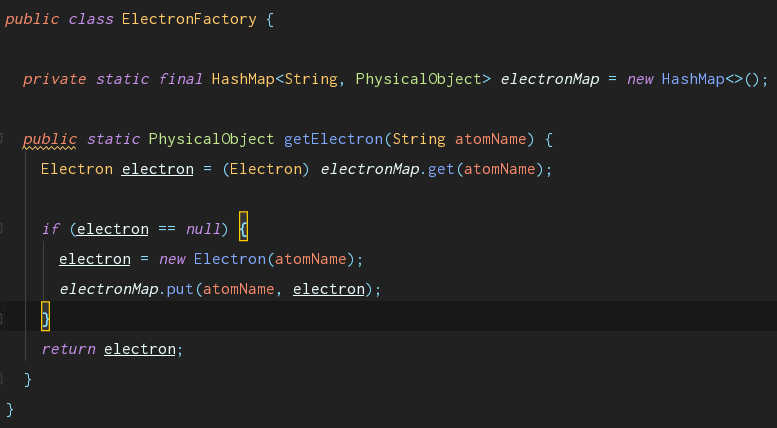
### 观察jstack/jcmd导出程序运行时的调用栈



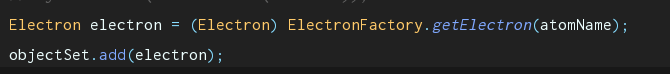
### 使用设计模式进行代码性能优化

1. 使用Flyweight设计模式优化atom系统的电子：

（1）使用工厂ElectronFactory，电子以原子名为标记，记录在map中；当调用getElectron时，若map中存在对应电子，则返回该电子，若不存在，就新建一个电子对应的，并put进map中

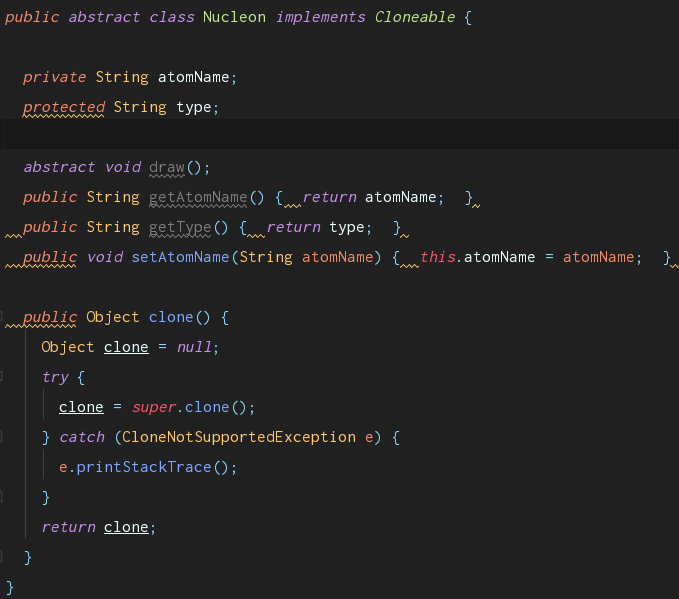


（2）在AtomStructureOrbit类中，每需要新建一个电子，就调用工厂方法getElectron。

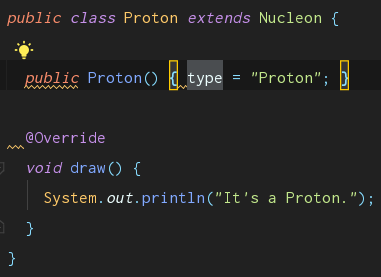
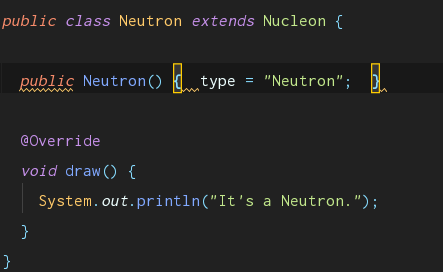


1. 使用Prototype设计模式，对AtomStructure进行拓展，支持原子核多个质子中子。

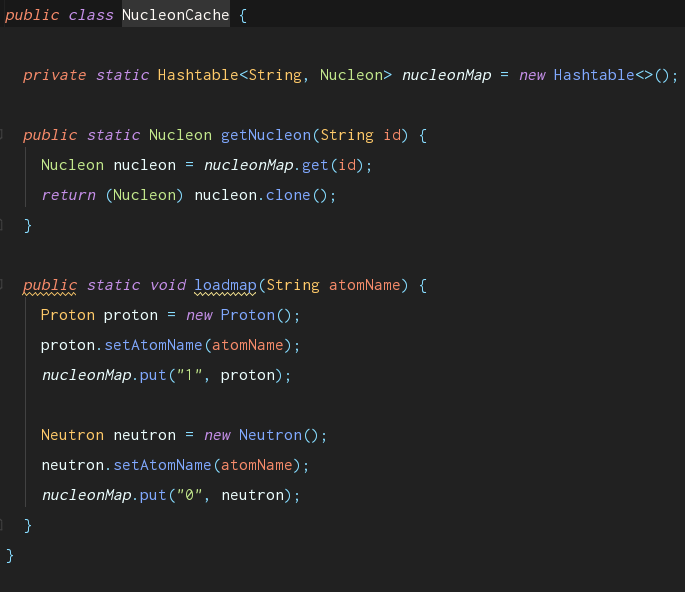
（1）设计核子Nucleon类，作为质子Proton类和中子类Neutron的父类，继承Clonable类



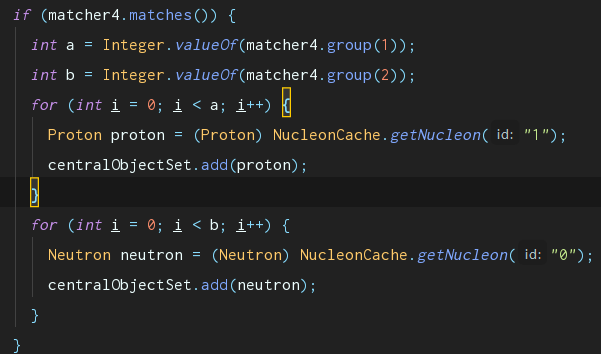
（2）设计质子和中子类



（3）设计工厂类NucleonCache，调用核子

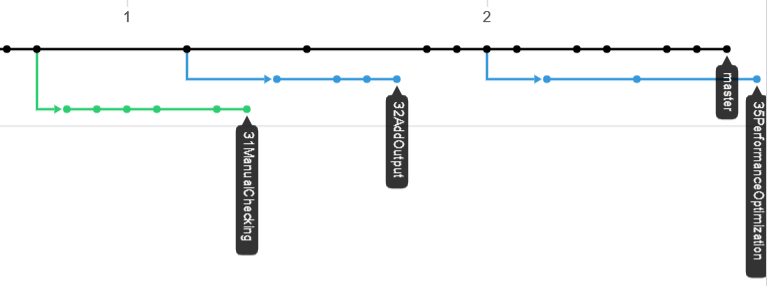


（4）修改atom系统



## Git仓库结构

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



# 实验进度记录

请使用表格方式记录你的进度情况，以超过半小时的连续编程时间为一行。

每次结束编程时，请向该表格中增加一行。不要事后胡乱填写。

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

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

# 实验过程中遇到的困难与解决途径

|  |  |
| --- | --- |
| 遇到的难点 | 解决途径 |
|  |  |
|  |  |
|  |  |

# 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？
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。