

**2019年春季学期  
计算机学院《软件构造》课程**

**Lab 5实验报告**

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**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

# 2实验环境配置

|  |  |
| --- | --- |
|  | 配置 |
| URL | <https://github.com/ComputerScienceHIT/Lab5-1170300614> |
| 系统 | Win10 |
| JDK版本 | 1.8.0\_16 |
| JAVA\_PATH | C:\Program Files (x86)\Common Files\Oracle\Java\javapath |

# 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）

0．注意使用大小写

1. 不允许使用tab，并且tab以两个空格的大小为单位。
2. 每个方法前都需要写JavaDoc
3. 包的名字首字母要小写。变量定义格式规范，首字母大写，不允许有两个连续的大写字母，不允许有特殊字符。
4. 消除所有的warning，比如没有被使用到的变量造成的warning.
5. 类内的属性是否都已经被初始化
6. 检查代码是否尽量使用if-else，不要一直if去判断，影响效率
7. 检查代码中被返回的变量的值是否可能为null，这可能造成其他地方的程序执行中断

8添加注释时注意格式

### 使用CheckStyle和SpotBugs进行静态代码分析

Checkstyle:

使用Sun check 使用自己的规范文件先将缩进功能都实现。

Spotbugs:

进行检查没有错误

## Java I/O Optimization

### 多种I/O实现方式

工程中采用了Stream，Writer/Reader，buffer来读写文件。

使用了strategy设计模式为读写分别设计了3种方式，并可以自由切换。

分别是FileStream、FileReader/Writer、BufferedReader/Writer。

首先设计一个strategy接口，包含一个读写方法声明；设计一个Read/Write类，包含一个构造器，参数为一个strategy类，还包含一个对应的读写方法，返回具体策略的读写方法；分别设计3个策略的实现即可。

**Stream读：**

使用java.io.FileInputStream中的FileInputStream类创建一个实例inputStream，用文件路径作为参数。然后把 inputStream实例作为参数创建一个InputStreamReader对象，最后传递给BufferedReader的构造方法，创建一个BufferedReader对象，并且从文件中读取，返回一个字符串，传递给相应的工厂类来解析并将相应信息传递入图中，直到文件末尾，读文件建图结束。

**Stream写：**

先利用文件路径创建一个File对象，然后与Stream读相对应的，传进FileOutputStream的构造方法，创建一个对象。然后遍历图中的信息，按行写入目标文件。

**Reader**读**：**

根据文件路径创建一个文件对象，然后用这个对象作为参数创建一个FileReader对象，并且按行读取文件中的信息，调用parse方法解析字符串，将信息加入图中。文件读到末尾后，文件读取和建图同时结束。

**Writer写：**

根据要写入的文件的路径，创建一个File对象，传入FileWriter的构造函数中创建一个FileWriter对象，然后再传入BufferedWriter的构造方法中创建一个BuffeeredWriter对象bwriter，遍历图中的顶点和边，将相应的信息写入目标文件。

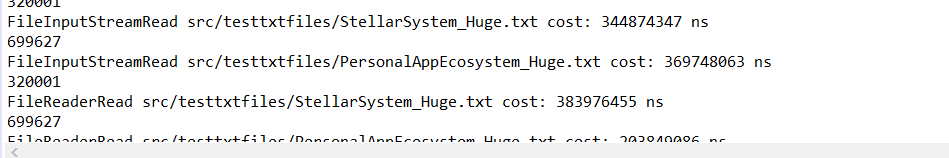
Strategy模式下，各种读写策略的转换方式如下：

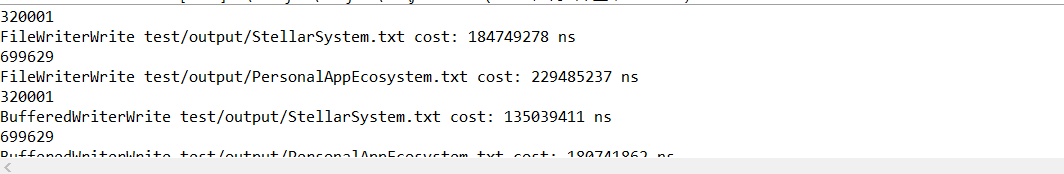
工程内有一个IO包，包内存储的是所有IO相关的类，首先是一个抽象类IO，里面定义了readStrategy和writerstrategy两个方法：

### 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。

使用test进行测试进行文法时间测试





表格方式对比不同I/O的性能。

|  |  |  |  |
| --- | --- | --- | --- |
| **时间(ms)** | **读/写** | StellarSystem.txt | PersonalAppEcosystem.txt |
| **I/O策略1**  **（stream）** | **读文件** | 345 ms | 369 ms |
| **写文件** | 8147 ms | 18919 ms |
| **I/O策略2**  **(writer/reader）** | **读文件** | 383 ms | 203 ms |
| **写文件** | 186 ms | 216 ms |
| **I/O策略3**  **(Buffer)** | **读文件** | 41 ms | 116 ms |
| **写文件** | 188 ms | 247 ms |

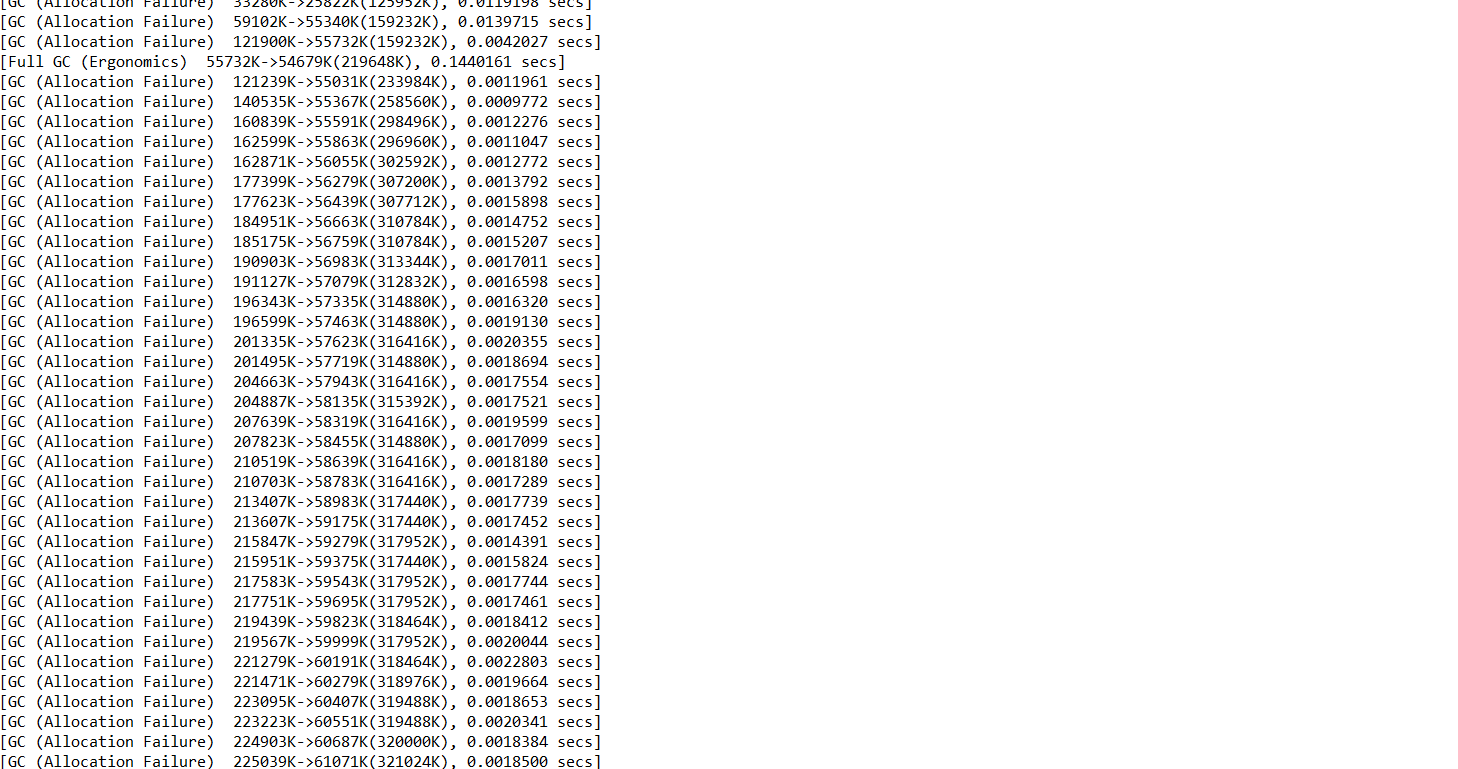
图形对比不同I/O的性能。

StellarSystem.txt

PersonalAppEcosystem.txt

## Java Memory Management and Garbage Collection (GC)

### 使用-verbose:gc参数



由日志内容可知：

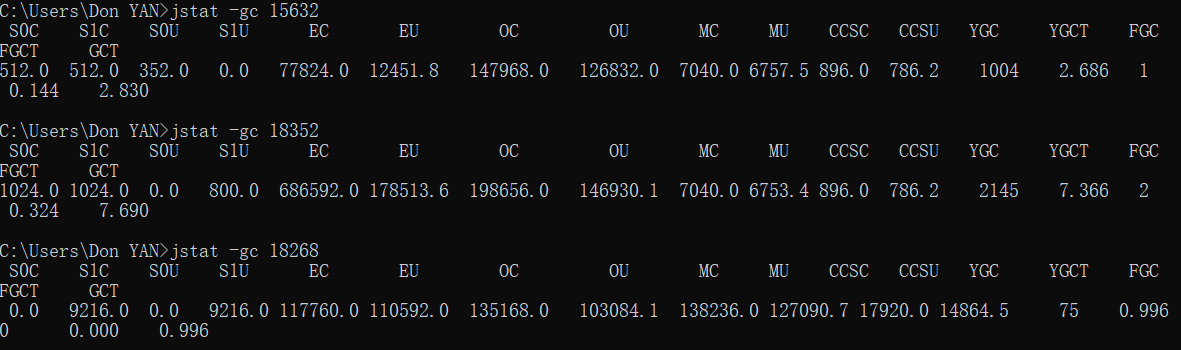
1. Minor GC一共发生了多次，Full GC发生了1次。
2. Full GC一共耗费了0.144秒
3. 每次GC前后heap中各区域占用情况变化：

Minor GC前后只有PSYoungGen区域占用变小，变小了一个数量级，堆上被占用的空间变小，变小一个数量级。

Full GC前后PSYoungGen区域占用清空，ParOldGen区域占用增加，Metaspace区域占用空间不变，整个堆上被占用空间变少。

(5)一共进行多次Minor GC和一次Full GC，而且每次Minor GC之后PSYoungGen区域的占用情况都能被缩小一个数量级，而且时间短，频率高。Full GC每次进行之后可以将PSYoungGen区域上的占用清空，但是时间耗费长，频率低。可以看到，Minor GC是JVM垃圾回收的主要方式，耗费时间短，频率高。当然，Full GC也是JVM不可或缺的垃圾回收方式。

### 用jstat命令行工具的-gc和-gcutil参数



**分析：**

①图中的参数的含义分别如下：

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；

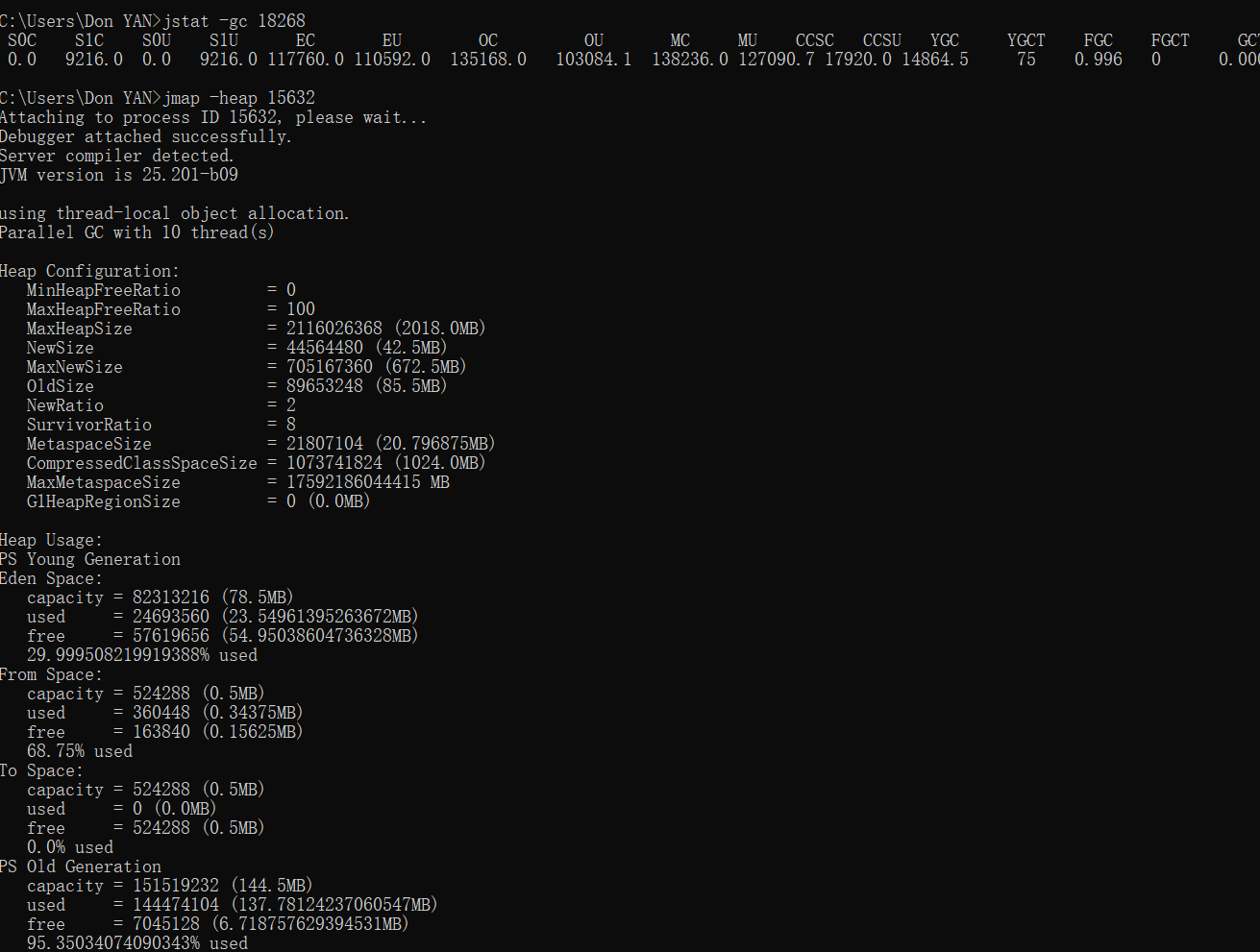
EC：Eden区的大小；EU：Eden区的使用大小

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

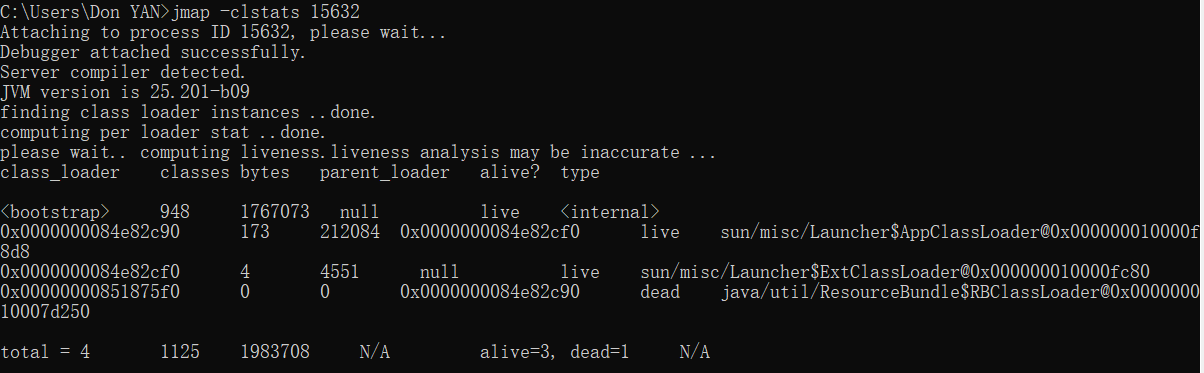
CCSC:压缩类空间大小；CCSU:压缩类空间使用大小；YGC：年轻代垃圾回收次数

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

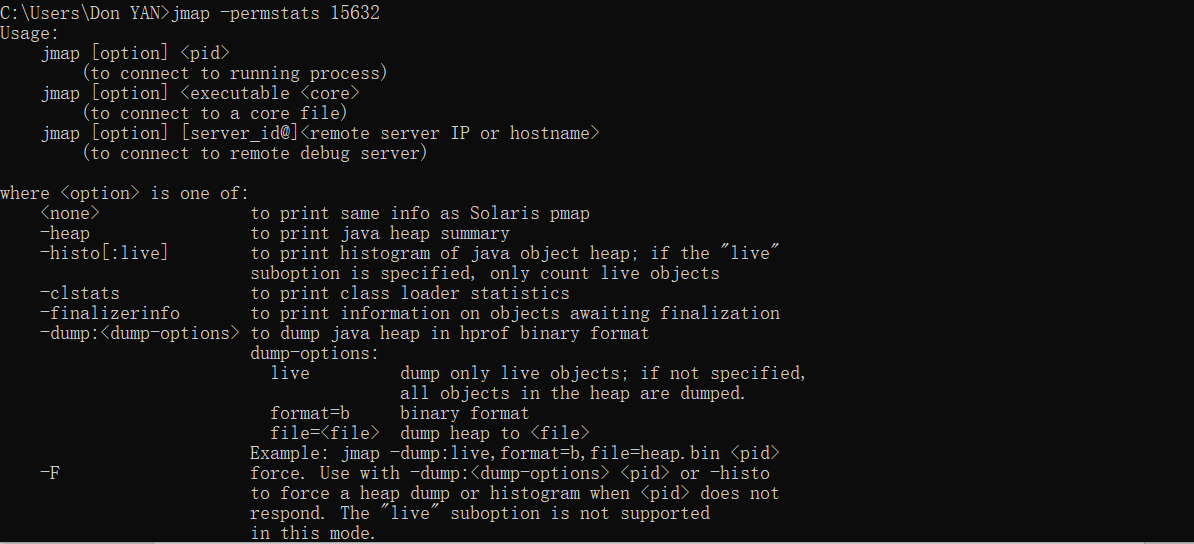
### 使用jmap -heap命令行工具

各种数据及含义显然

### 使用jmap -clstats命令行工具



### 使用jmap -permstat命令行工具



Java1.8之后完全除去了PermGen，所以没有 jmap -permstat Pid指令，取而代之的可以采用jamp -clstats Pid命令。

打印classload和jvm heap长久层的信息,包含每个classloader的名字，活性，地址，父classloader和加载的class数量。

这里显示的是一次读取文件时装载的class和method相关信息。

### 使用JMC/JFR、jconsole或VisualVM工具



使用的堆的大小明显减小的部分是Full GC，其他的是Minor GC，另外堆的大小也在变化。装载的class相关的信息。在程序运行过程中，装载的类的数量基本上没有发生变化。还有实时线程和守护线程的变化情况

### 分析垃圾回收过程

每次在使用堆接近于堆的大小时，堆的大小都会被扩展，同时堆也会展开垃圾清理，维持程序正常工作，在整个过程中，堆扩展了两次以满足程序运行的需要，同时垃圾回收机制维护了程序的正常运行，程序运行结束后，使用的堆的大小和堆的大小趋于稳定，不再有大的变化。从图中可以看到，垃圾回收过程正常。

### 配置JVM参数并发现优化的参数配置

Xmx:JVM最大可用内存。Xms:JVM最小可用内存。Xmn:年轻代大小。

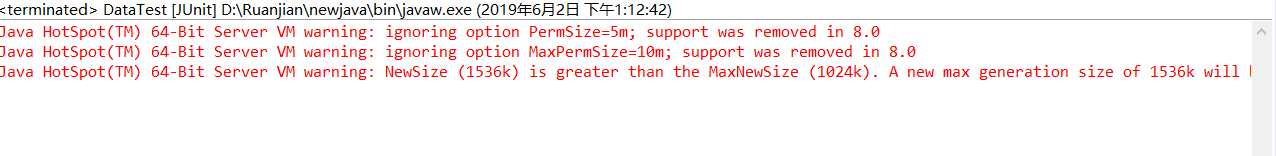
-XX:PermSize：表示非堆区初始内存分配大小。

-XX:MaxPermSize：表示对非堆区分配的内存的最大上限。

-XX:+Use使用的垃圾回收是并行还是串行

**1.测试配置**：-Xmx32m -Xms4m -Xmn1m -XX:PermSize=5m -XX:MaxPermSize=10m -XX:+UseSerialGC

测试结果：发生了异常



**2.测试配置：**-Xmx500m -Xms500m -Xmn320m -XX:+UseSerialGC

测试结果：从日志来看，没有了Full GC，Minor GC次数也变少了。用时较少.

**3.测试配置：**-Xmx1002m -Xms1002m -Xmn160m -XX:+UseSerialGC

测试结果：用时与2差不多，每次消耗时间与2相差不多。总用时也与2差不多。

综上，最优的策略应该是：-Xmx1002m -Xms1002m -Xmn320m -XX:+UseSerialGC

## Dynamic Program Profiling

### 使用JMC或VisualVM进行CPU Profiling

设置：



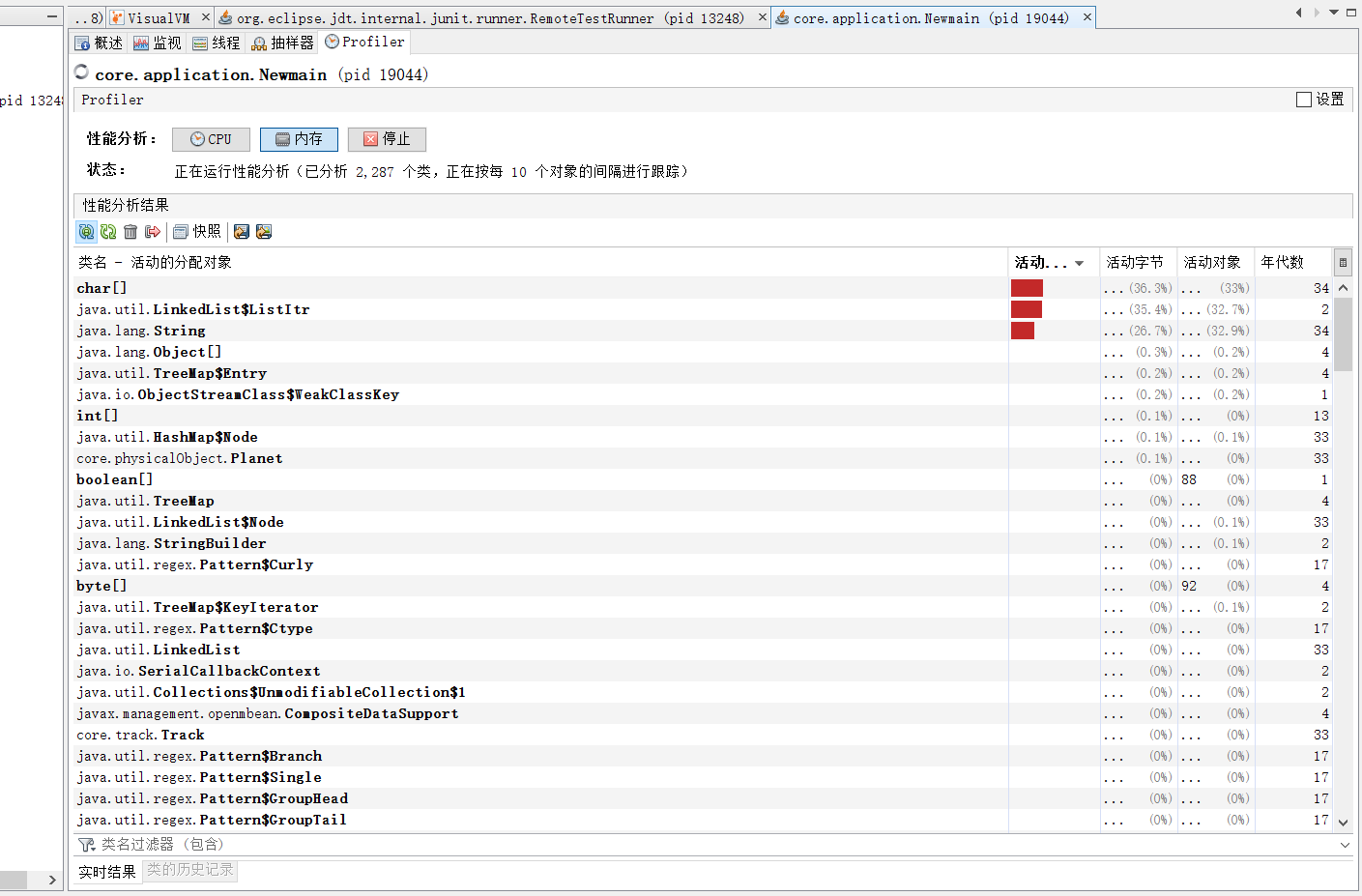
读操作：

src.stellarsystem()方法占据的时间最多，占据了60%.其次是src.application.main()方法，占据了26%，剩下依次是Strea()方法，从获取的信息分析来看，结合工程的特点，出现这个结果是合理的。在buildsystem()方法中存在大量的正则匹配和字符串处理，以及很多的条件判断和一小部分的异常处理，操作比较复杂，占据时间比较多是合理的。其次是main()，其中有循环操作，很多函数调用和输出，占据时间也合理。剩下的方法的操作操作简单，都是常数级的操作，占用时间较少，合理。

写操作：

main()方法占据的时间最多，因为里面操作复杂，并且在读写工程中都在工作，所以合理。另外可以看到write()方法也占用了较多时间，合理，因为要向文件中写入信息。

### 使用VisualVM进行Memory profiling



从图中可以知道int[]最多，。其次是char[]和Object[]，这些耗费内存是合理的。

写操作：

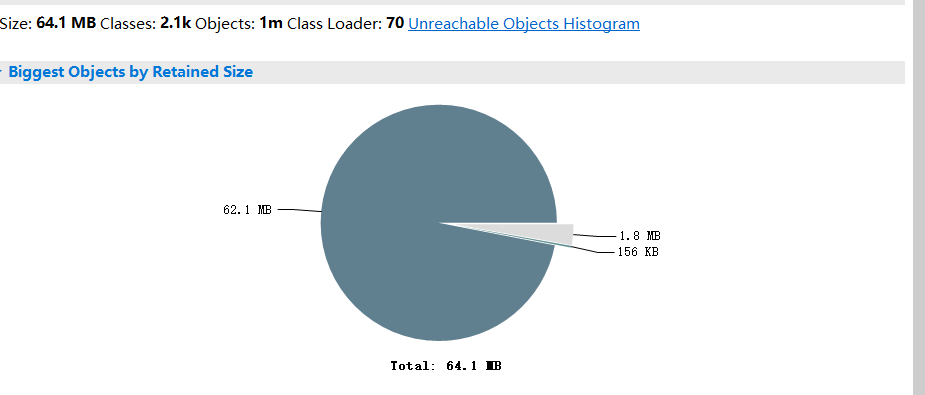
char[]最多，其次是byte[]和Object[]，合理，因为在写操作时，需要生成很多字符串，并且写入文件。

## Memory Dump Analysis and Performance Optimization

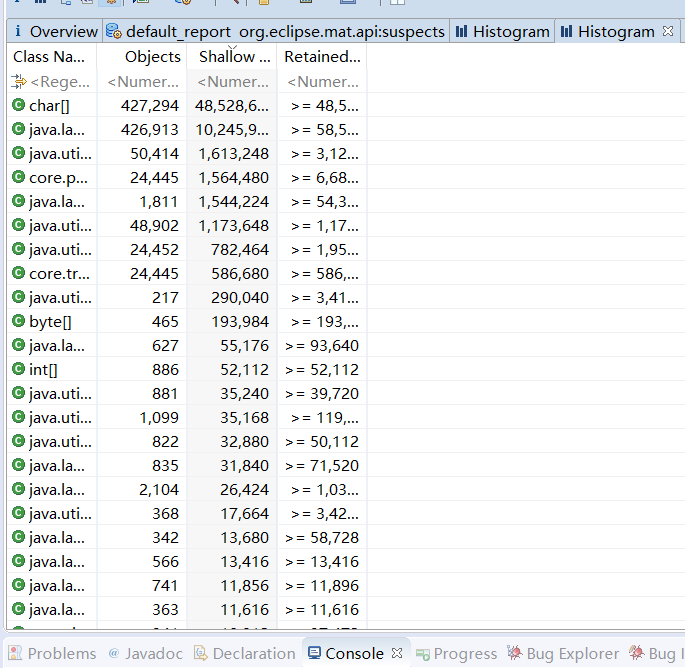
### 内存导出

在读取完文件并生成Graph对象后，在Visual VM中点击生成堆dump，就会在本地自动生成一个.hprof文件。

### 使用MAT分析内存导出文件

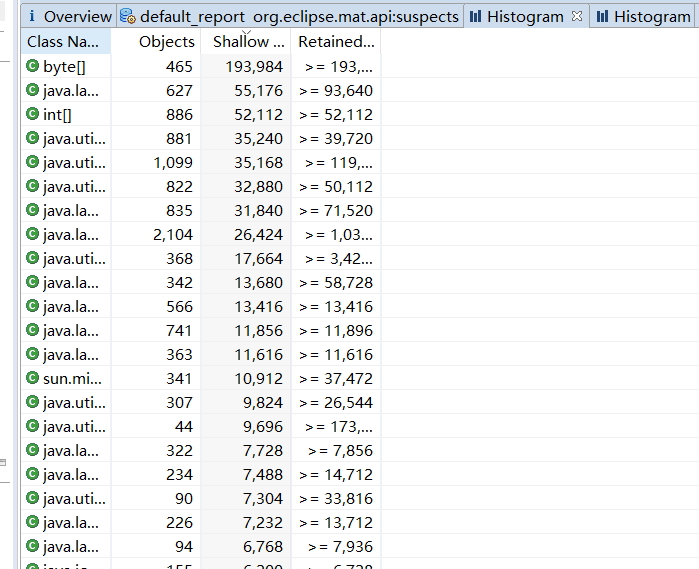
1.overview界面以饼图的方式显示当前消耗内存最多的几类对象，可以使我们对当前内存消耗有一个直观的印象。

2.Histogram: Lists number of instances per class(列出每个类的实例的数目来分析)

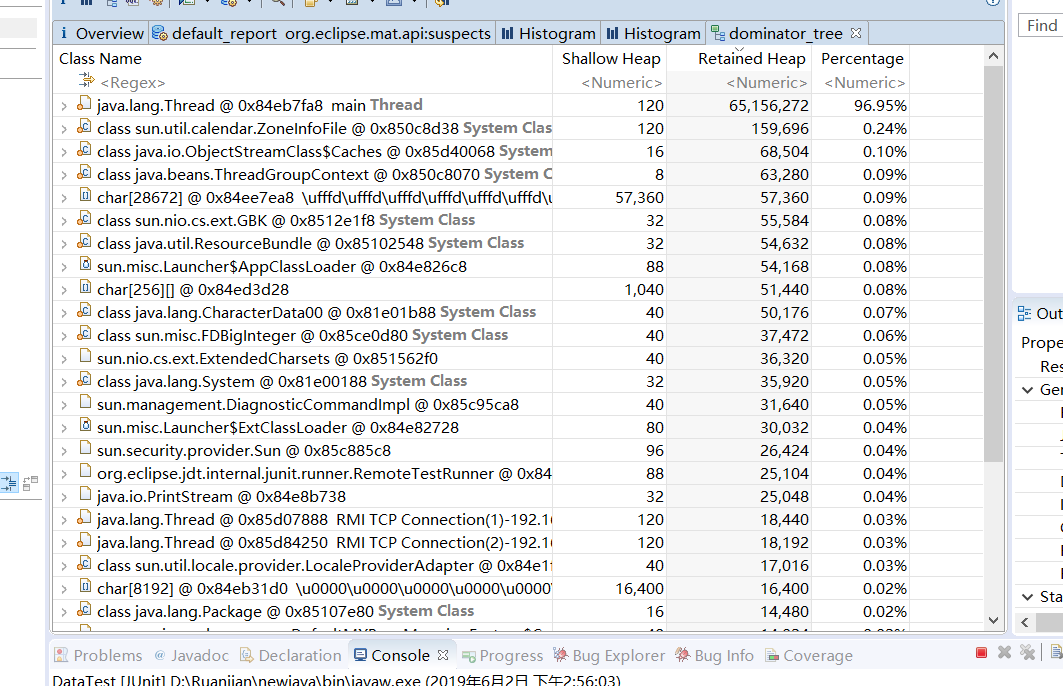


**分析：**

可以看到文件中轨道的个数和各类物体的个数。通过上述数据可以查到，在图中也有标注。它以图表的方式来显示当前内存使用情况，更加适合较为复杂的内存泄漏分析，它默认直接显示当前内存中各种类型对象的数量及这些对象的shallow heap和retained heap。



3. Dominator tree：List the biggest objects and what they keep alive.

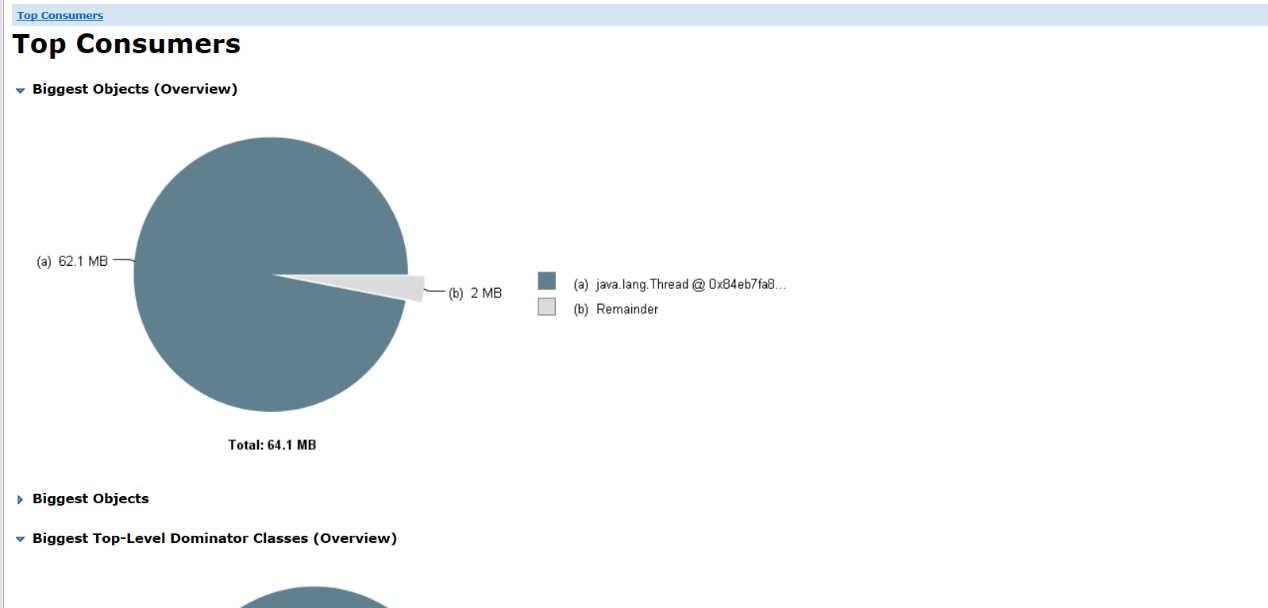


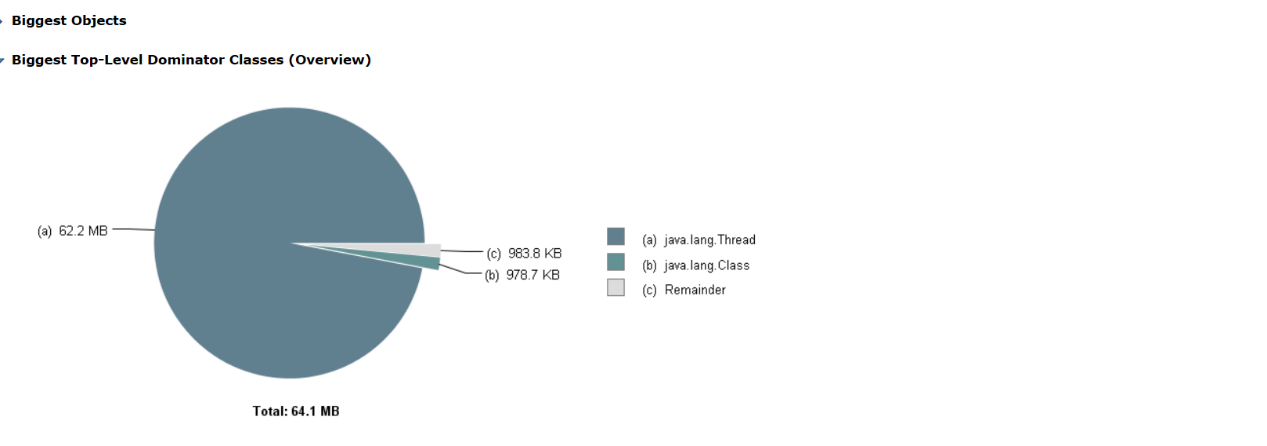
**分析：**

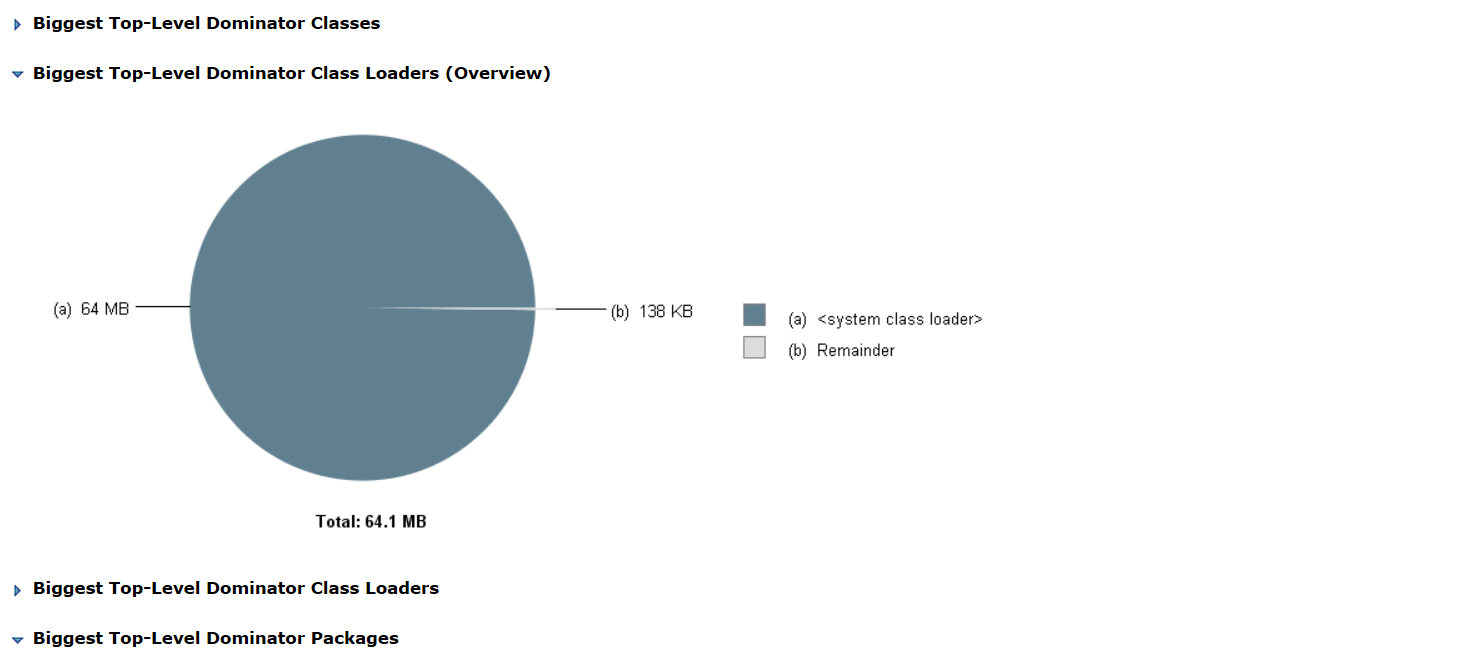
通过MAT提供的Dominator Tree，可以很清晰的得到一个对象的直接支配对象，如果直接支配对象中出现了不该有的对象，说明发生了内存泄漏。

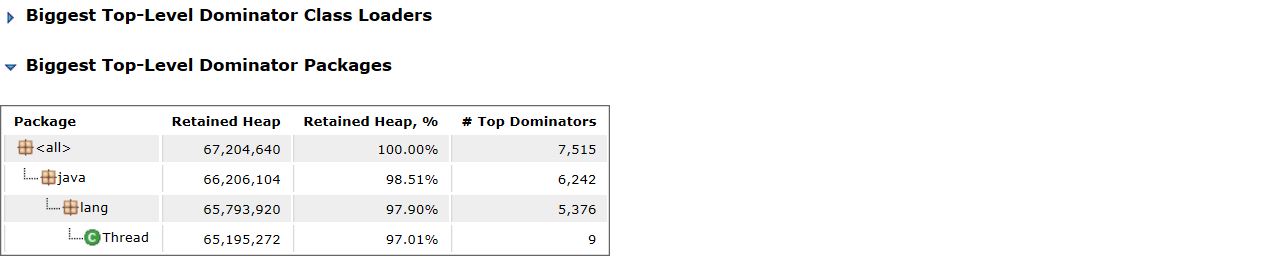
4.Top Components: list reports for components bigger than 1 percent of the total heap.

通过图形列出最大的内存对象有哪些，它们对应的类有哪些，类加载器classLoader是哪些。

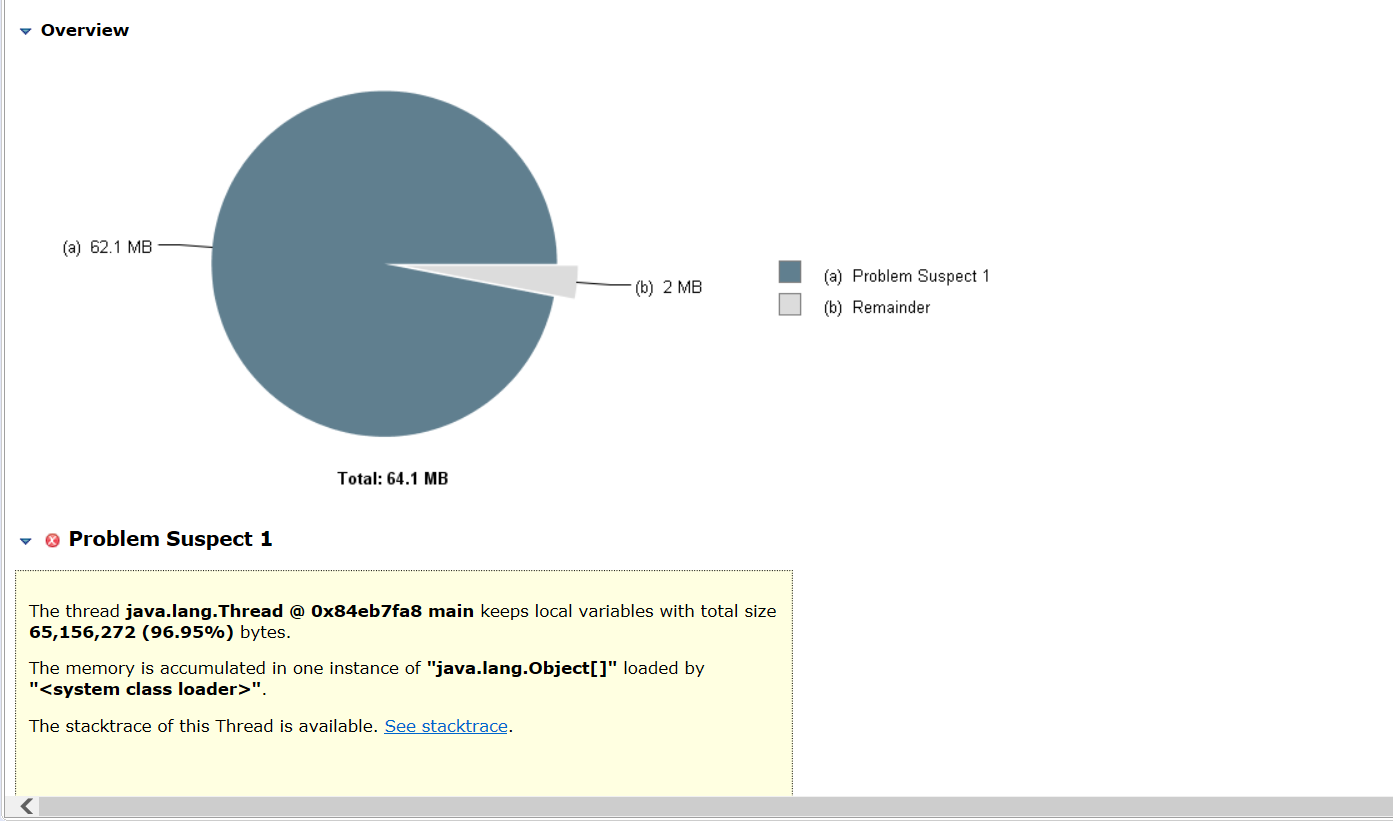




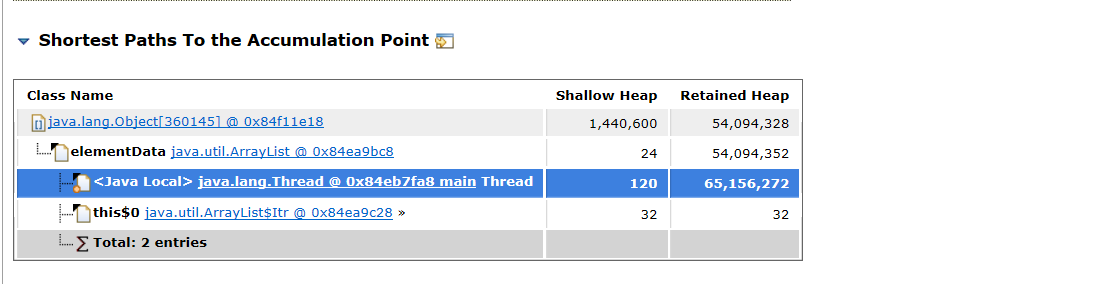


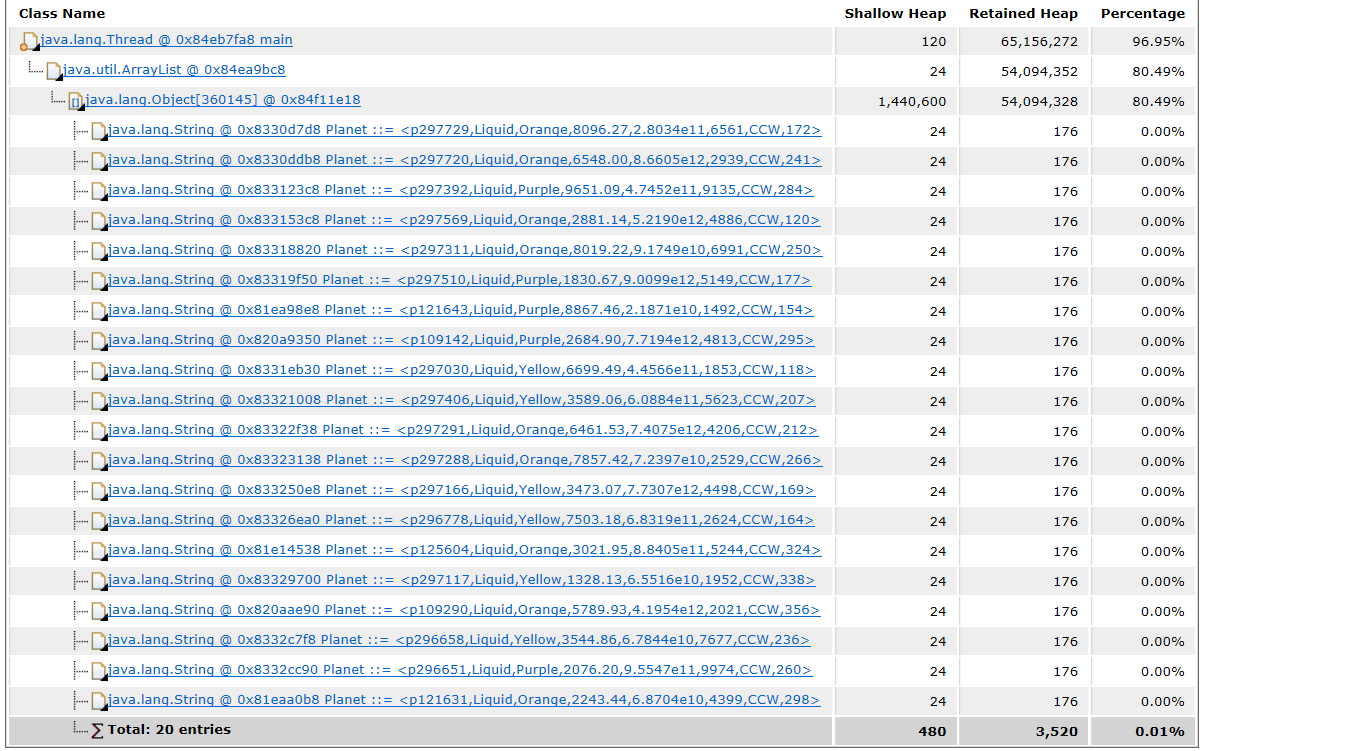


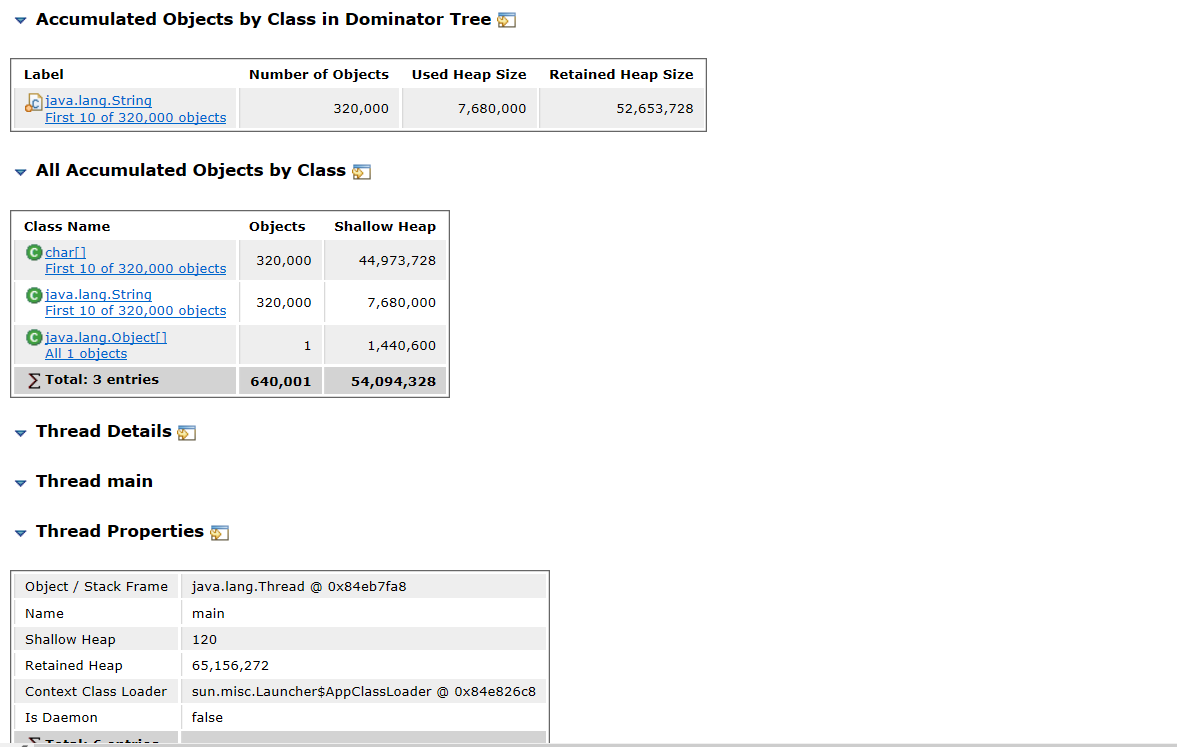
5. Leak suspects report：



(a)区域表示可能发生内存泄漏，点击内存泄漏猜想的Details可以看到更深入的分析清理情况。







### 发现热点/瓶颈并改进、改进前后的性能对比分析

①改进前，读文档遇到边的端点的标签，要把相应的顶点加入边的list属性是通过遍历一遍所有的顶点，匹配相应的顶点，加入边的list中，这样太耗费时间，每次向边中加入端点耗费的时间都是O(n)的操作。改进策略：在图中加入一个map属性，map的key是顶点的label，map的value是顶点对象。在读文档遇到边的端点的label时，可以直接根据map中的映射利用label得到结点，花费时间是O(1)的，大大节省了时间。

②在向图中加入边时，需要判断边的顶点是否在图中已经存在，之前采用的方法是遍历一遍所有的顶点看边的顶点是否已经存在，消耗是时间是O(n)的。改进方法：将顶点提前加入一个set中，要判断的时候直接调用set.contains()方法，因为set是存储特点，所以查找消耗时间的复杂度是O(1)，大大节省了时间。

③在对图中的list操作时，做了防御性拷贝/深拷贝，导致消耗了90%的时间，由于在整个过程中不会对ADT中的属性做违法修改，所以直接返回部分可变属性，节省了90%的时间。

### 在MAT内使用OQL查询内存导出

①查询长度大于等于120的字符串: select s from java.lang.String s where s.value.length >= 120

②Vertex（及其每个子类）的对象实例的数量和总占用内存大小

对象实例的数量: select count(map(heap.objects('src.vertex.Vertex')))

③对象实例的总大小: sum(map(heap.objects('src.vertex.Vertex'), 'sizeof(it)'))

### 观察jstack/jcmd导出程序运行时的调用栈

1.删除轨道；

先调用application.stellarsystem()方法解析命令并且执行命令，然后调用removetrack()方法删除边，最后是为了抓取此时的调用栈轨迹而特意调用的sleep()方法。

2.增加物体

可以看到先调用了application.stellarsystem()方法解析命令并且执行命令,然后调用图中addobjects()方法来向图中添加物体，最后是为了抓取调用栈轨迹调用的sleep()方法

### 使用设计模式进行代码性能优化

## 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 |
| 2019.5.20 | Free days | 完成代码的手动检查 | 基本实现 |
| 2019.5.21 | Free day | 实现机器代码检查 | 基本实现 |
| 2019.5.22 | ALLdays | 实现各种I/O策略 | 未实现 |
| 2019.5.25 | Fre day | 实现读写 | 基本实现 |

# 实验过程中遇到的困难与解决途径

|  |  |
| --- | --- |
| 遇到的难点 | 解决途径 |
| 不能很流畅地使用各种cmd指令 | 网上学习 |
| 选择I/O策略 | 都实现 |
|  |  |

# 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？

答：一个要求效率，一个要求美观。两者都很重要。后者

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？

答：是的。有很大的作用。

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？

答：I/O策略的多样性体现了java的强大之处，暂时能够

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？

答：差异较多，jvm的内存管理更加的靠近编程。我认为JVM已经很好了

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？

答：两者各有优劣。还有较大的改进空间。

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？

答：是有足够的回报的。

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？

答：是的。

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？

答：是的。

1. 关于本实验的工作量、难度、deadline。

答：一般。

1. 到目前为止，你对《软件构造》课程的意见与建议。

答：课程难度较大