

**2018年春季学期  
计算机学院大二软件构造课程**

**Lab 5实验报告**

|  |  |
| --- | --- |
| 姓名 | 朱明彦 |
| 学号 | 1160300314 |
| 班号 | 1603003 |
| 电子邮件 | [1160300314@stu.hit.edu.cn](mailto:1160300314@stu.hit.edu.cn) |
| 手机号码 | 18846082306 |

**目录**

[1 实验目标概述 1](#_Toc515298177)

[2 实验环境配置 1](#_Toc515298178)

[3 实验过程 1](#_Toc515298179)

[3.1 Static Program Analysis 1](#_Toc515298180)

[3.1.1 人工代码走查（walkthrough） 1](#_Toc515298181)

[3.1.2 使用CheckStyle和FindBugs进行静态代码分析 2](#_Toc515298182)

[3.1.2.1 checkStyle发现问题 2](#_Toc515298183)

[3.1.2.2 findbugs发现的问题 4](#_Toc515298184)

[3.1.2.3 checkstyle和findbugs的比较 5](#_Toc515298185)

[3.2 Java I/O 5](#_Toc515298186)

[3.2.1 多种I/O实现方式 5](#_Toc515298187)

[3.2.2 多种I/O实现方式的效率对比分析 8](#_Toc515298188)

[3.3 Java Memory Management and Garbage Collection (GC) 11](#_Toc515298189)

[3.3.1 使用-verbose:gc参数 11](#_Toc515298190)

[3.3.2 用jstat命令行工具的-gc和-gcutil参数 12](#_Toc515298191)

[3.3.3 使用jmap -heap命令行工具 13](#_Toc515298192)

[3.3.4 使用jmap -histo命令行工具（可选） 13](#_Toc515298193)

[3.3.5 使用jmap -permstat命令行工具（可选） 14](#_Toc515298194)

[3.3.6 使用jconsole或VisualVM工具 14](#_Toc515298195)

[3.3.7 分析垃圾回收过程是否正常、异常 16](#_Toc515298196)

[3.3.8 配置JVM参数并发现最优参数配置 17](#_Toc515298197)

[3.4 Dynamic Program Profiling 21](#_Toc515298198)

[3.4.1 使用Visual VM进行CPU Profiling 21](#_Toc515298199)

[3.4.2 使用Visual VM进行Memory profiling 22](#_Toc515298200)

[3.5 Memory Dump Analysis and Performance Optimization 24](#_Toc515298201)

[3.5.1 内存导出(memory dump) 24](#_Toc515298202)

[3.5.2 使用MAT分析内存导出文件 24](#_Toc515298203)

[3.5.3 发现热点/瓶颈并改进、改进前后的性能对比分析 27](#_Toc515298204)

[3.5.4 jhat和OQL查询内存导出（可选） 29](#_Toc515298205)

[3.5.5 jstack导出java程序运行时的调用栈（可选） 33](#_Toc515298206)

[4 实验进度记录 34](#_Toc515298207)

[5 实验过程中遇到的困难与解决途径 39](#_Toc515298208)

[6 实验过程中收获的经验、教训、感想 39](#_Toc515298209)

# 实验目标概述

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”。

# 实验环境配置

由于使用IDEA作为开发使用的IDE，在其中没有Memory Analyzer Tool分析工具，所以在实验的后期需要将文件导出至Eclipse中继续测试。对于其余的测试工具如jmap、jstack等均可以直接使用，visual VM安装后也可以正常使用。

在这里给出你的GitHub Lab5仓库的URL地址（Lab5-学号）。

https://github.com/ComputerScienceHIT/Lab5-1160300314

# 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）

列出你所发现的问题和所做的修改。每种类型的问题只需列出一个示例即可。

发现的问题：单行的代码过长，超过了100个字符，具体的代码如下：

|  |
| --- |
|  |

（由于在word中一行的代码显示问题此处不能显示出长度，此处的长度为127字符组成）

由于过长的代码对于程序员直接去看会非常累，所以我们采用了合理划分长度的办法，在等号之后划分，最终的效果如下：

|  |
| --- |
|  |

### 使用CheckStyle和FindBugs进行静态代码分析

完成相关要求的commit ID为：

（1）完成相关findbugs的工作，见如下

<https://github.com/ComputerScienceHIT/Lab5-1160300314/commit/a8328c6fdce2d83dc56ecbb58a574768234de930>

（2）完成checkstyle相关工作，见如下：

<https://github.com/ComputerScienceHIT/Lab5-1160300314/commit/295515a61a35c17f3b522f0603a8dc468e07a6f7>

#### checkStyle发现问题

checkStyle此处我使用的是Google Style作为代码的规范的格式进行要求，所以在运行checkstyle后主要有以下几个问题。

（1）行前空格。在Google Style的要求中，必须使用两个空格代替tab键作为行前缩进。此处的遵循这种代码规范的原因，是因为Google公司在使用Java的过程中发现，Java是一个多层缩进的代码，举个例子

这是Java的很常见的一种格式，在真正的函数体的前方的缩进已有8个空格，另外最长的代码一行最好不要超过100个字符，这样一下就减少了近十分之一。

|  |
| --- |
|  |

所以谷歌就建议使用两个空格进行缩进，如下：

|  |
| --- |
|  |

（2）Javadoc的书写规范

在Google中要求Javadoc的第一个描述的最后必须有结束的符号，如下所示

|  |
| --- |
|  |

（3）注释的缩进应与最近的代码缩进类似，即不能出现注释的双斜线在正常代码的前面，比如下面这种情况：

|  |
| --- |
|  |

而应该将注释的位置向后移动，转为如下形式：

|  |
| --- |
|  |

（4）import 库的时候，google不允许使用类似这种\*通配符来导入多个类文件。如下：

|  |
| --- |
|  |

如果我使用了相应的类应该仅仅导入我所需要的类文件，比如我使用了assertEquals函数，那么

|  |
| --- |
|  |

#### findbugs发现的问题

findbugs主要发现的问题就是赘余的代码和字符编码的问题。

（1）代码赘余的问题，就是所有定义过的变量均需要有地方使用，也就是我们不能有的变量定义了但是却没有使用。解决的办法，就是将所有的不必要的变量没有用到的变量均删除。

（2）IO依赖于平台的编码，也就是我们IO策略没有考虑编码的问题，在findbugs中表现出的问题“Found reliance on default encoding”。解决办法如下：

|  |
| --- |
|  |

对于上述的代码，其实在使用的时候就默认使用了平台代码，其实非常简单，只需要在new对象的时候传入另一个参数作为默认使用的编码形式如下

|  |
| --- |
|  |

就可以避免这种潜在bug的引入。

#### checkstyle和findbugs的比较

在checkstyle中关注的就是格式问题，比如行前缩进应该有几个空格、Javadoc应该怎么写，这些的内容不是必须的。而是像第4章中说的可理解性，即使我们的代码写的“奇丑无比”，也是可用的。但是在与同行之间的交流就会显得非常困难，因为缩进没有一点也不美的代码会给人极大的厌恶感。

findbugs更关注的是代码较深的层次里的问题，我这样写代码是不是有引入潜在bug的风险，比如我依赖于平台的编码方式，那么我的可移植性是不是受到了极大的阻碍，并且在移植后的正确性是不是还可以保证，这些问题是findbugs所关注的。

## Java I/O

### 多种I/O实现方式

DataInputStream、InputStreamReader和BufferedReader进行读入的策略。其中所有的读入策略都是使用readline的方式逐行读入并且进行建图。

所有的输入的策略的都是实现了以下接口：

|  |
| --- |
|  |

DataOutputStream、OutputStreamWriter和BufferedWriter方式进行写出的策略。其中所有的输出策略都是基于String输出的方式。

所有的输出策略都是基于以下接口实现：

|  |
| --- |
|  |

对于每个IO策略，在读入的时候都是基于readLine的策略进行的，也就是我们每读一行就处理一行；而在输出的策略都是将已有的图转换为String之后才利用输出的策略进行输出。

如何用strategy设计模式实现在多种I/O策略之间的切换。

在以往使用的GraphFactory.createGraph之外建立了新的Adaptor作为可以选择Input策略的新的创建图的工厂，具体来看对比。在不选择输入策略，默认使用带有buffer的BufferStrategy。

|  |
| --- |
|  |

对于输出图的函数在以往的OutputGraph.outputGraph之外增加了一层Adaptor作为可以选择输出策略的函数。如果不想选择输出策略的函数，可以默认使用OutputGraph.outputGraph，会默认选择Buffer作为输出策略。

对比图如下：

|  |
| --- |
|  |

### 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。

表格方式对比不同I/O的性能。

图形对比不同I/O的性能（可选）。

为了降低单次读入文件建图的时间，所以之前将所有需要读入的文件（file1-file4）均做过处理，保证其中所有的顶点和边都是符合语法并且符合相应的不变量要求。**仅仅将读图建图的时间作为所测量的时间，在此期间不会做对于文件内容合法性的检测。**

测试所有的类文件如下图所示。

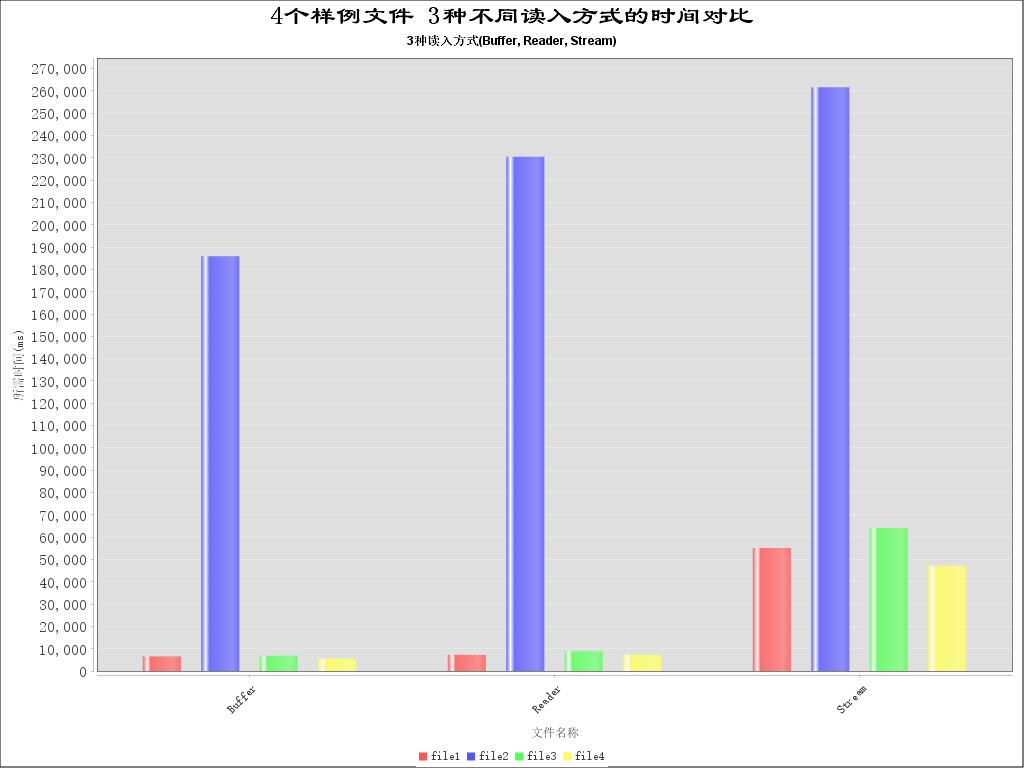
|  |
| --- |
|  |

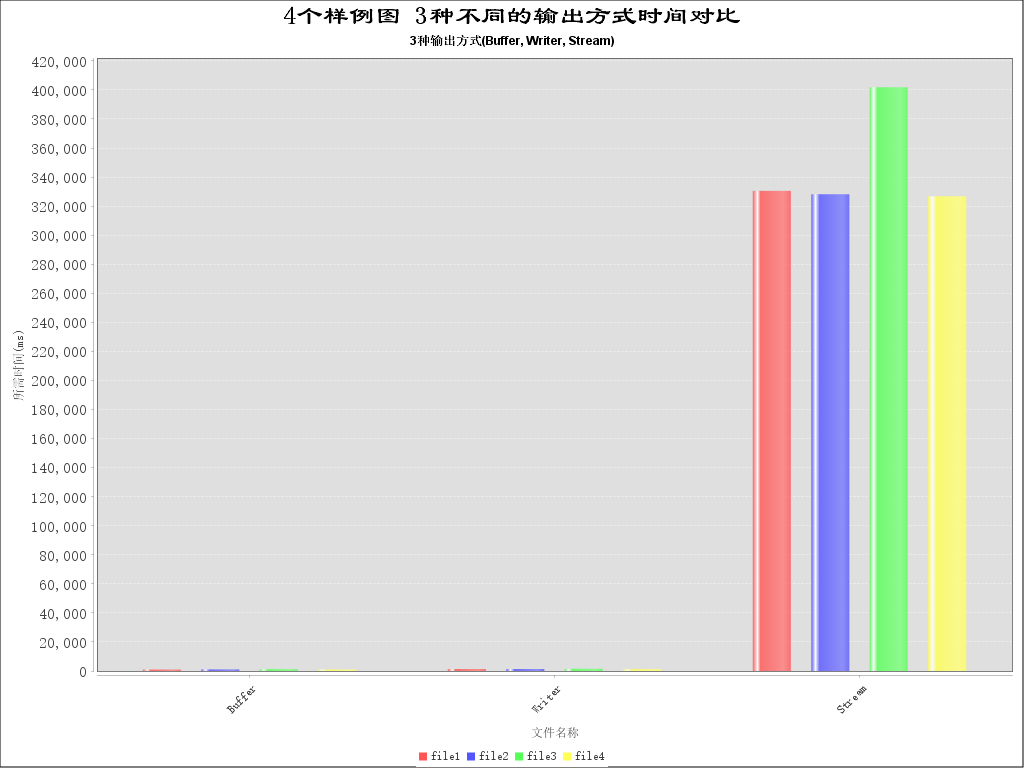
为了测试的准确性所有的数据均交错测试（避免cache对于多次测试结果的影响）原始数据见附件（./doc/testTime.xlsx）。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 单位(ms) | file1.txt | file2.txt | file3.txt | file4.txt |
| Buffer Input | 6554 | 185878.7 | 6797.333 | 5615.667 |
| Buffer Output | 1033 | 1111.666667 | 1275.666667 | 1016 |
| Stream Input | 55142 | 261664 | 64120 | 47191 |
| Stream Output | 330542 | 328183 | 401885 | 326776 |
| Reader Input | 7252.333333 | 230462.6667 | 8868.333333 | 7349 |
| Writer Output | 1354 | 1221 | 1559.33333 | 1286.33333 |

（对于Stream 输入和输出的，由于显然可以看到其运行的时间都远超其他的各种方法，且运行时间较长，此处没有再进行多次试验，仅以一次实验的结果进行展示）

**另外利用JFreechart将以上的数据进行可视化，可以得到以下结果：**

****

****

**对于输出方式，由于Stream的时间过长，可能表示效果不佳，详细的结果可以看原始数据文件。**

## Java Memory Management and Garbage Collection (GC)

### 使用-verbose:gc参数

在测试类中连续读100次file1.txt，并使用-verbose:gc参数将GC的情况输出至log文本文件中，经过对于log文件的分析可以得到如下的信息。

所有使用的命令行参数如下：

|  |
| --- |
|  |

（a）在全过程中一共进行了757 次Minor GC，平均每两次进行Minor GC的时间间隔是 0.5853725231175694s，每进行一次Minor GC使用的平均用时 0.045822134478203394s

（b）在全过程中一共进行了21次 Full GC，平均每两次进行Full GC的时间间隔是20.24266666666667s，每进行一次Full GC的平均用时 0.5618594285714285s

（c）在每次Minor GC前，年轻代的内存区域平均会占到最大年轻代的95.48%，在经过一次Minor GC之后，就会降到3.9%左右，与此对应的是每一次MinorGC之后通过对比Heap的变化可以平均有28M的年轻代会被拷贝到Old generation的区域。

（d）在每次进行Full GC之前，Old Gen占比都可以达到最大Old Gen的94%以上，然后每一次Full GC之后的就会将其占比降到11%左右。

（e）综合数据可以看到，Old Gen和Young Gen之间的内存分配比例是65%对比35%。

### 用jstat命令行工具的-gc和-gcutil参数

根据jstat -gcutil可以看到（详细的数据文件见./doc/gc分析jstat -gcutil.xlsx或./doc/gc分析jstat -gc.xlsx）下面只摘取前两次的Minor GC数据进行分析。

|  |
| --- |
|  |

采用的采样的时间间隔为250ms。

可以看到在进行第40次Minor GC所用的时间为0.044s（1.706-1.662），并且以后的每进行一次Minor GC的时间也不会超过0.1s，也就是进行Minor GC的时间适中，Minor GC回收时间正常。Eden 区域的占用率经过一个 Minor GC后从89.99%下降到22%。

对于Full GC的分析，针对以下数据进行。

|  |
| --- |
|  |

可以看到在进行一次Full GC的时间所用为0.288s（0.61-0.322由于取样的间隔为250ms， 所以两次取样才得到一次Full GC的时间），进行一次Full GC的时间也合理。并且经过一次，可以回收相当部分的垃圾，比如从这里Old Gen的占用比例的93.48%下降到8.94%。

并且Metaspace的空间利用率和压缩类的空间利用率都是95%左右，表示JVM的垃圾回收正常。

### 使用jmap -heap命令行工具

使用jmap -heap输出的信息可以看到（具体的输出文件见./doc/jmap\_-heap.txt）：

当前的虚拟机使用的是并行的垃圾回收策略（使用4个线程进行垃圾回收）。

最大堆的规模为2G。

新生代的当前区域为341.0M，最大规模为682.5M。

Old Gen的当前区域为683.0M。

当前的Young Gen的各个区域的使用率：

Eden的使用率为89.9%；S0的使用率为31.9%；S1的使用率为0.0%。

Old Gen的使用率为68%。

### 使用jmap -histo命令行工具（可选）

使用jmap -histo命令行可以看到当前装载进内存区域的各类的实例数目和占用的内存的情况，具体的情况如下，仅列出前十位大小的类和实例数表格如下：



其中[C表示char类型；[I表示Integer类型；由于使用了过多的HashSet导致在内存中有关HashMap Node的使用数量过多，这将是后面进行优化的时候的内存瓶颈所在。

### 使用jmap -permstat命令行工具（可选）



使用jmap -clstats可以看到以上的结果。

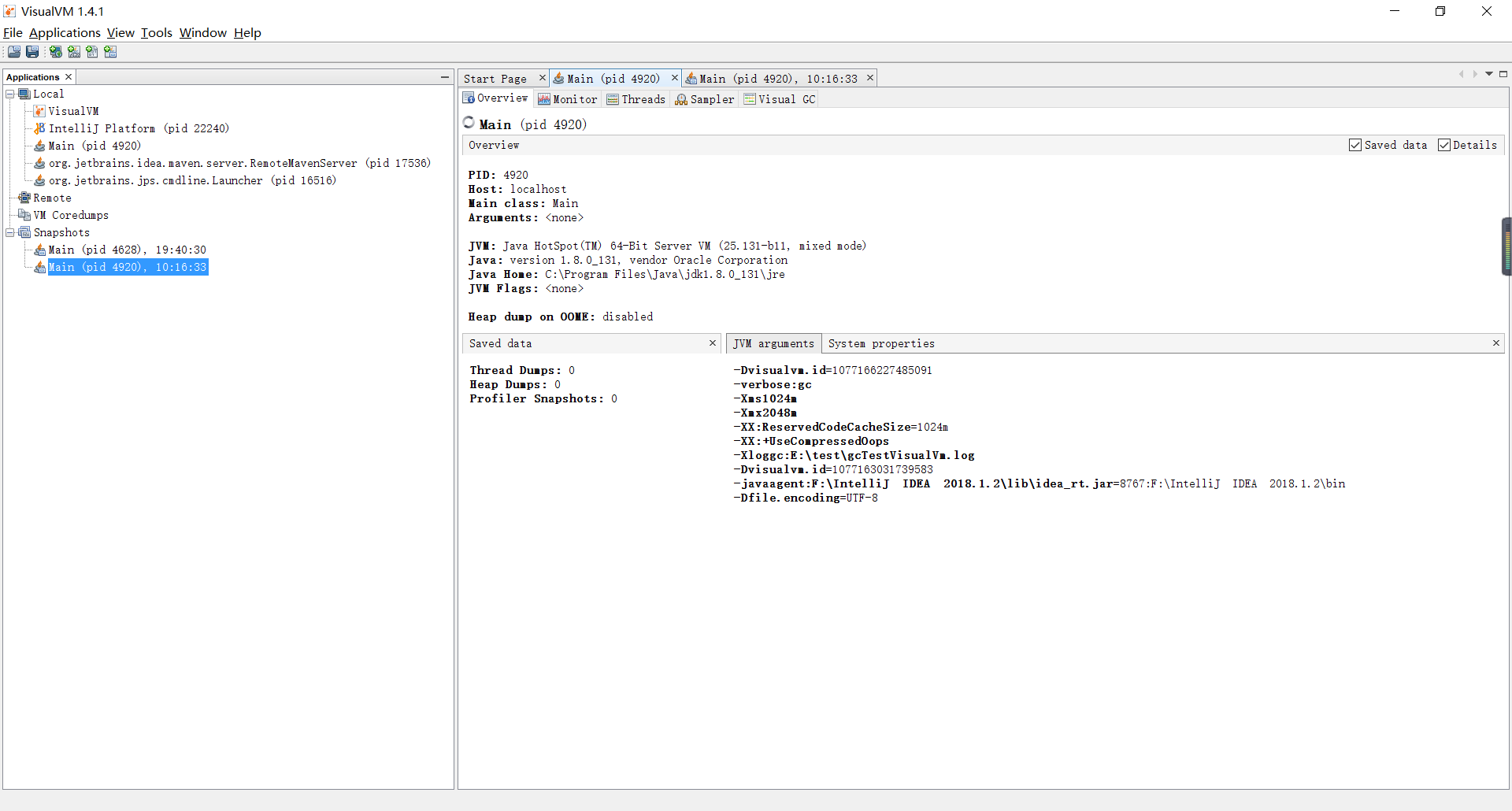
### 使用jconsole或VisualVM工具

使用Visual VM可以得到以下图例



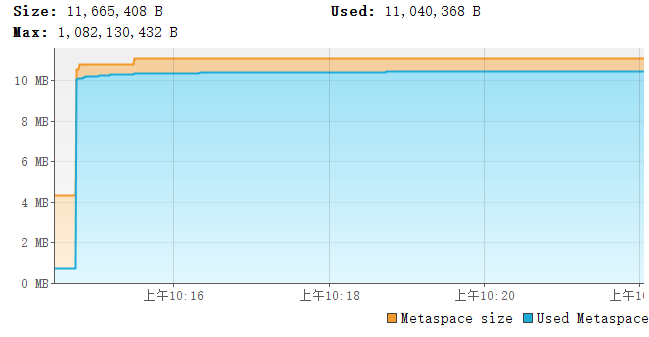


可以看到在使用当前参数的情况下：



Old Gen进行Full GC的次数过多，占用的时间也比较多，证明在Old Gen区域的大小偏小，可能需要增大。

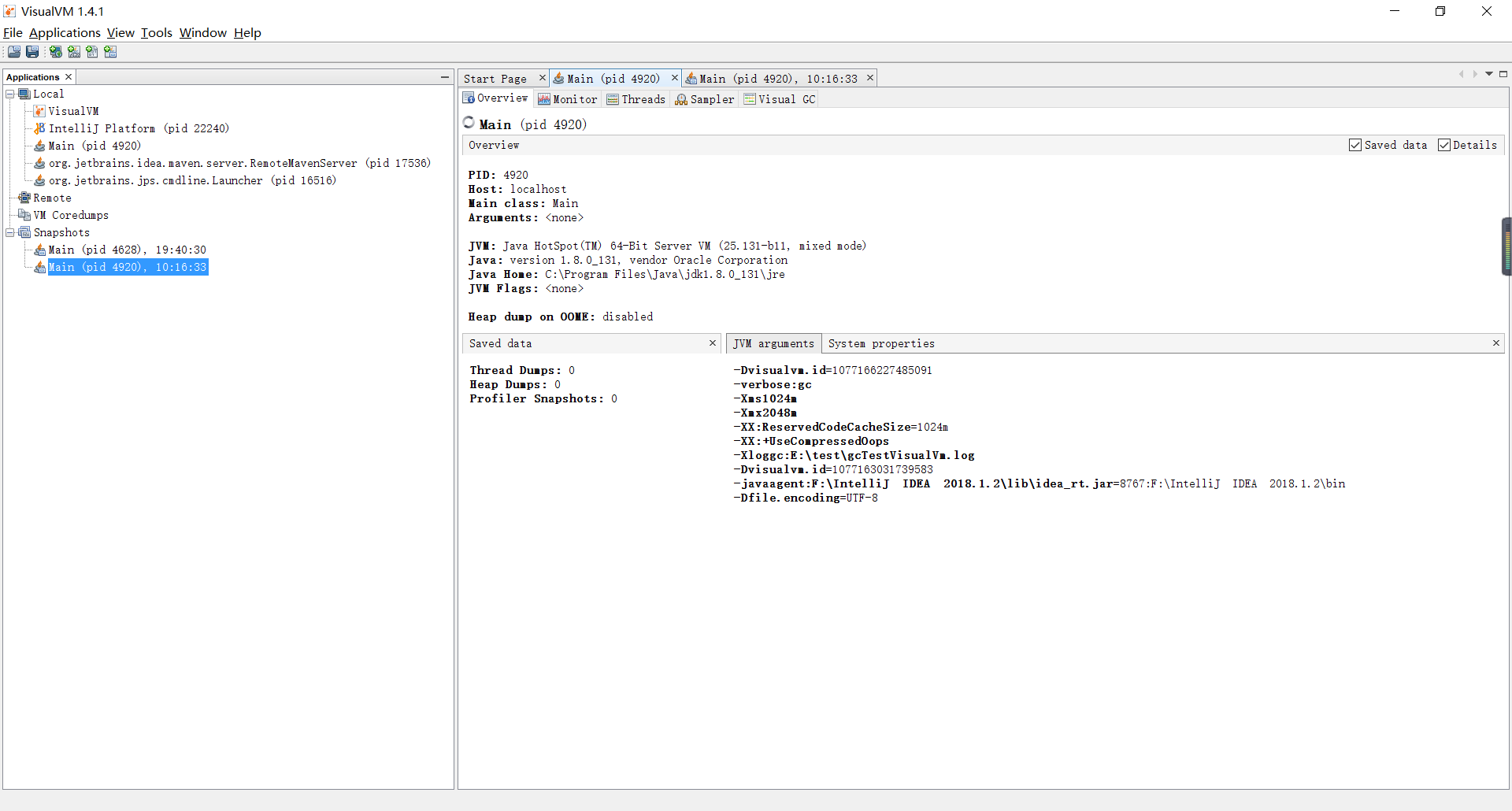
另外对于Metaspace的区域在刚刚开始运行的时候有一个明显的升高，证明对于元数据区域的大小也偏小，这里也是一个优化的地方。



最终连续读入100次file1.txt并构造图所用时间为7min40s所以根据以上的信息进行重新配置JVM参数进行优化。

### 分析垃圾回收过程是否正常、异常

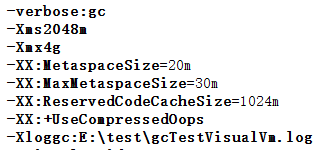
在以下参数的时候，仅仅运行了读入不到20次就已经进行了十几次的Full GC，说明垃圾回收不正常，Full GC的频率过高，这与新生代的初始化内存过小有着密切关系。



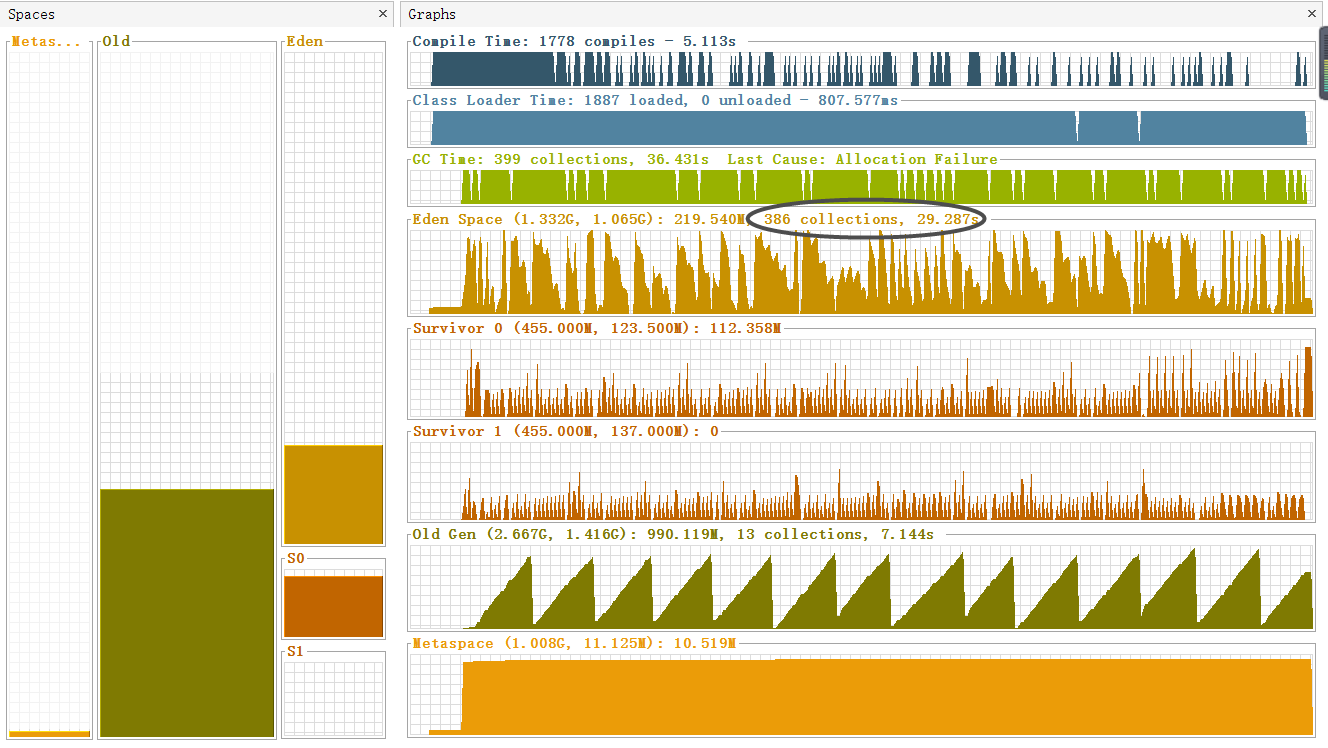
针对以上问题在3.3.8节进行优化。

### 配置JVM参数并发现最优参数配置

（1）调整参数为以下:

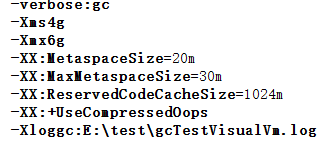


最终的运行完连续读入100次file1之后，所用时间为7min23s相比之前的运行时间有所进步，但是根据

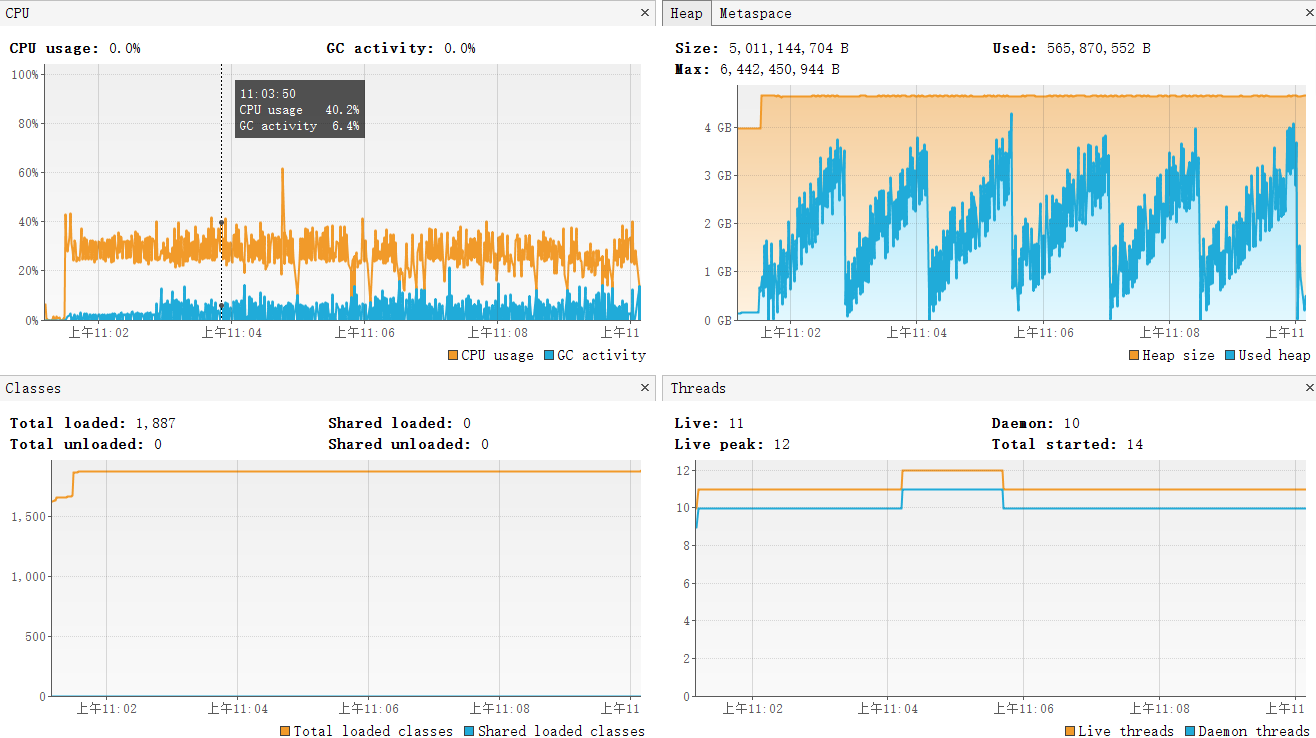


可以看到Young Gen的Minor GC进行的仍然过于频繁，基于此可以推断new Size过小，因此修改参数进一步进行试验。

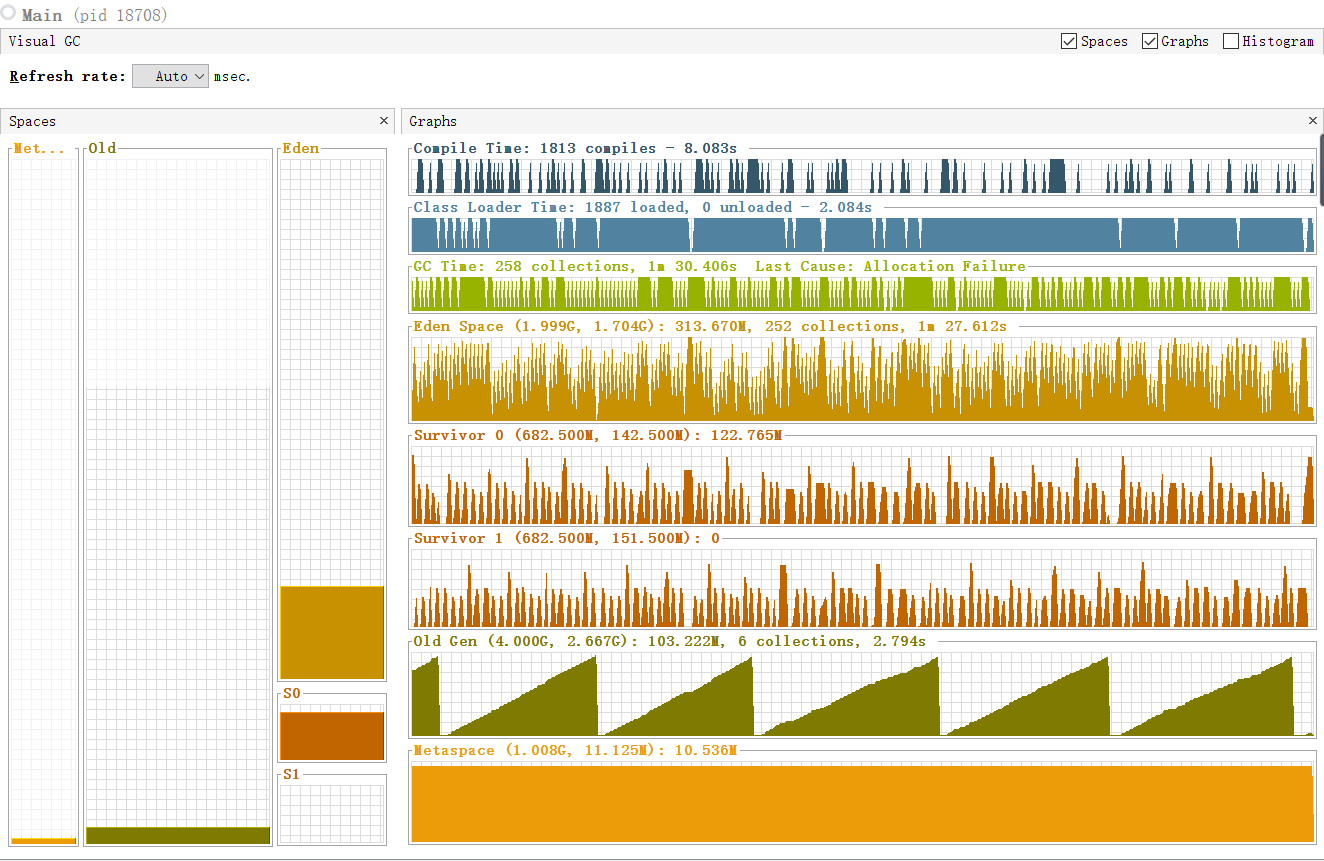
（2）调整参数，增大Young Gen的规模，具体参数如下：



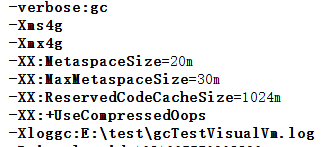
然后时间运行的更加慢了，具体的原因可以看到：



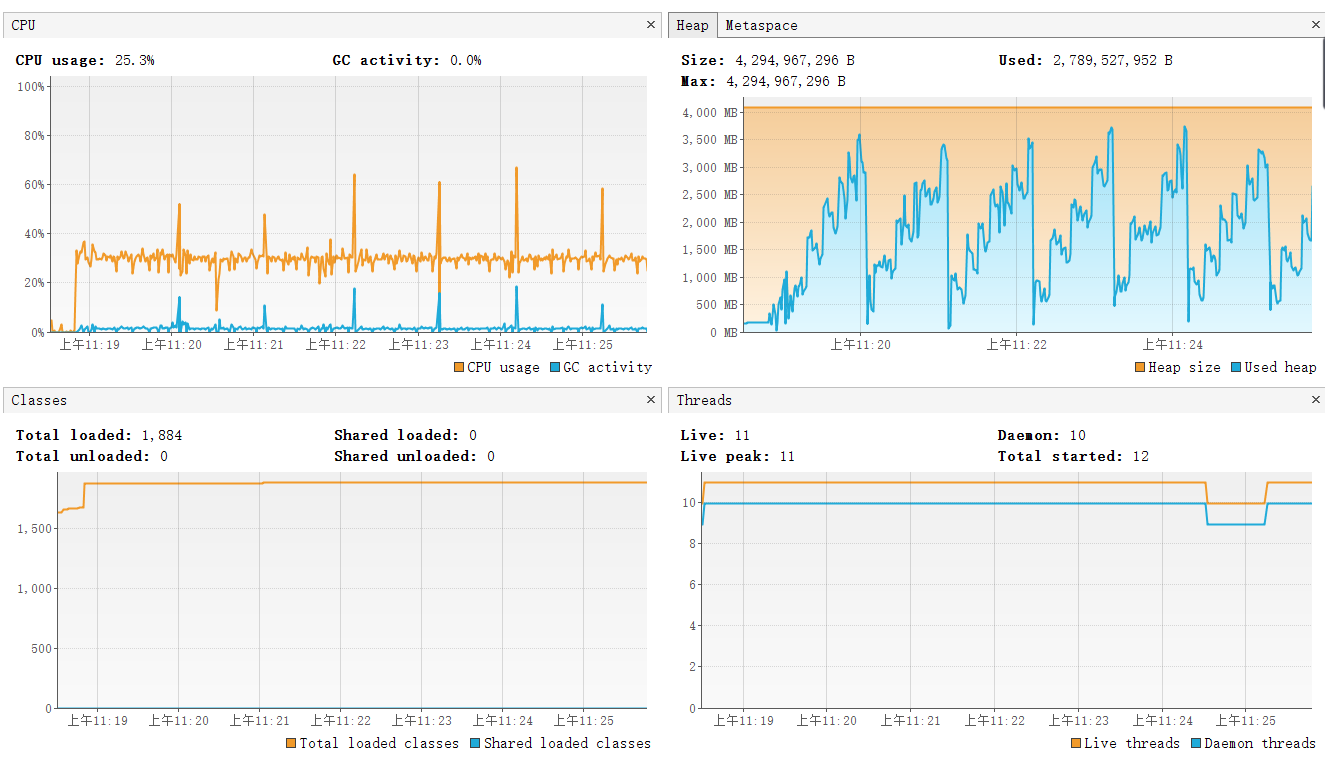
由于每一次Full GC耗费的时间和CPU过长，导致整体的运行时间不但没有相比增大之前下降。GC时间如下：

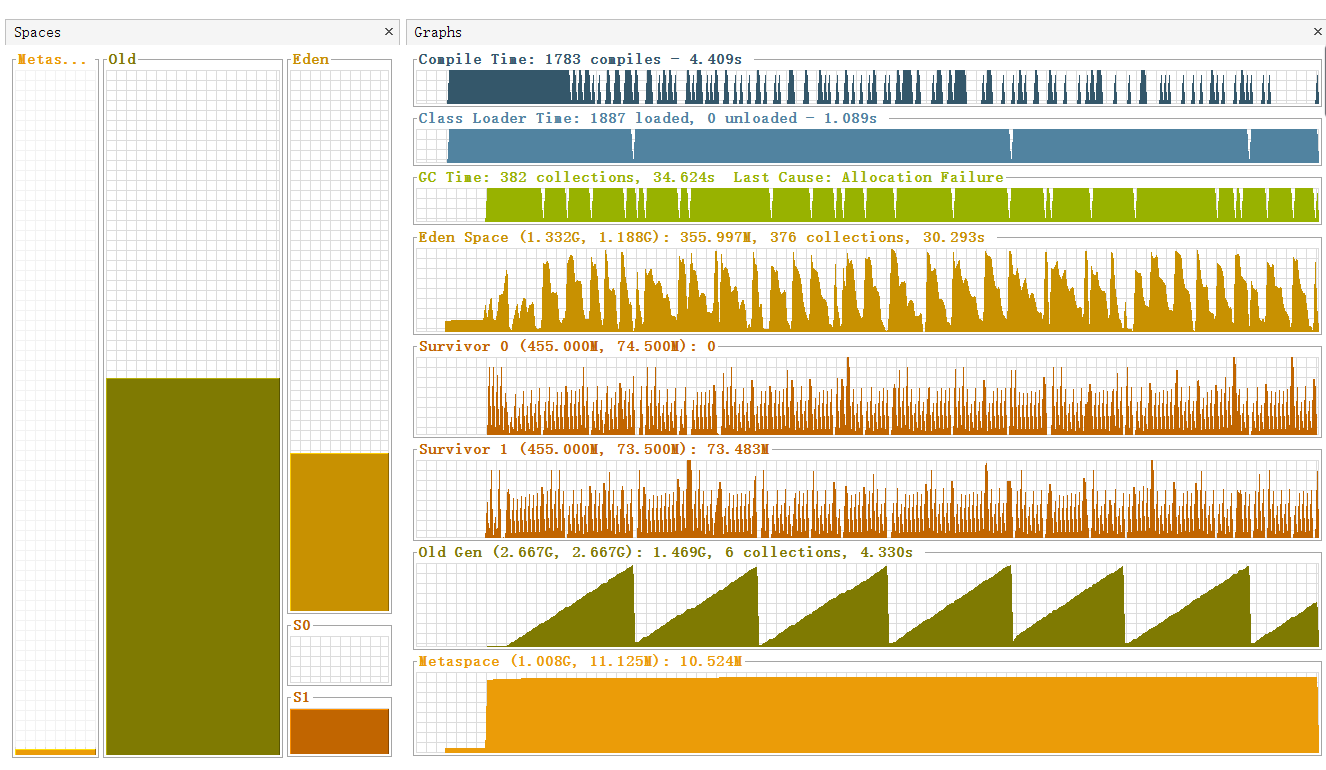


（3）调整参数至如下：



将Young Gen的比例调高，重新进行实验后的效果相比之前的略有提高。



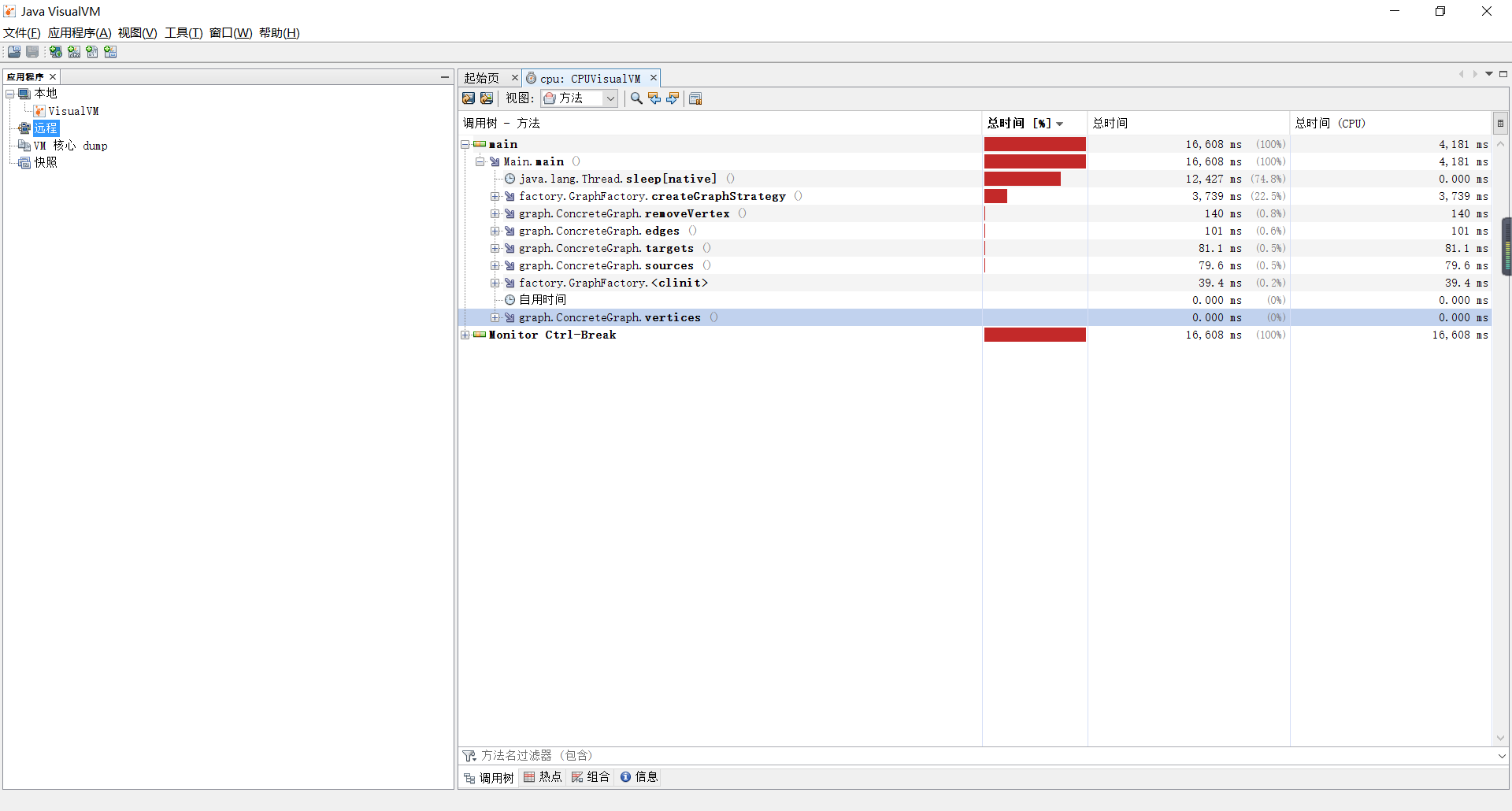


最终的运行时间从7min23s下降到7min18s，可以看到主要优化的时间就是在FullGC少了7次，减少了3s左右。此时的配置应该已经接近最优的JVM配置，因为更低配置运行和更高配置的运行速度都有所下降。

## Dynamic Program Profiling

### 使用Visual VM进行CPU Profiling

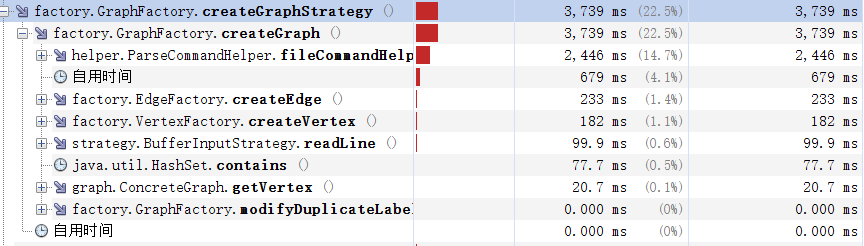
使用Visual VM启动程序读入file1对图进行CPU profiler。可以得到下图：



**在最前面用时最长的是sleep函数，此处是防止Visual VM程序打开较慢不能及时打开Profiler进行监控。**

我们的测试文件先是建图，然后分别运行在Graph接口中定义的所有函数，监控使用CPU的时间。

最终得到的测试结果可以看到，建图所用的时间最长，这点显而易见由于其中的边和点的数量非常多。在CreateGraphStrategy这个函数中，又多次解析来自输入文件的输入信息，所以更详细的监控时间图如下：



（1）在建图过程中，更多的时间用在了fileCommandHelper中，也就是用于解析输入信息的函数耗时最长，所以如有进一步的优化，应该在函数内部将处理输入信息的方式做一些优化，如更好的使用正则表达式等等。但总体来说运行时间已经比较合理。

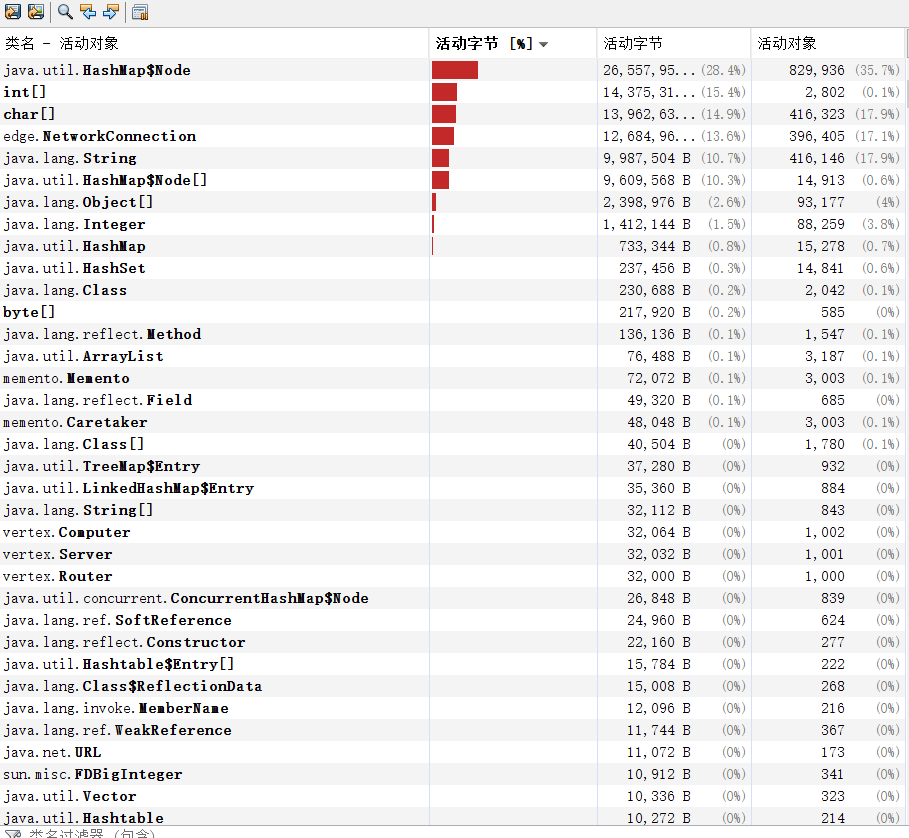
（2）再对比多个操作之间运行时间其中对于removeVertex耗时最长，这是由于删除的顶点是有多条边邻接的，那么为了找到所有邻接该顶点的边，必然涉及到一次对边的遍历，所以运行时间耗时合理。

（3）然后时间运行较长的是edges函数，即返回整个图的所有边的集合，由于此处为了方式表示泄露，使用的是将返回的集合进行一次拷贝，所以用时较长合理。

（4）对于剩余的两个函数可以一起来看，targets和sources都是找到以某一个顶点为出发点或者终结点的边，所以仍然需要类似于上面提到的edges函数来遍历所有的边并将符合要求的边加入map，所以运行时间合理。

### 使用Visual VM进行Memory profiling

对于Memory Profiler的监控使用的是仍然是与CPU监控相同的方式，可以得到下图：



此处只截出来了所占内存较大的一些类。

可以看到最多的是HashMap$Node，这是由于大量使用了HashSet并且在测试最后测试了targets和sources函数，他们的返回值均为HashMap。

另外查看顶点的个数，可以看到：



computer的数量恰好为从文件中读入的1000个对象和在测试中新加入的两个对象相符，server对象的数量与文件中读入1000个对象和测试中新加入1个对象相符。综上，认为在实验中memory的分配是合理的没有多余浪费的。

## Memory Dump Analysis and Performance Optimization

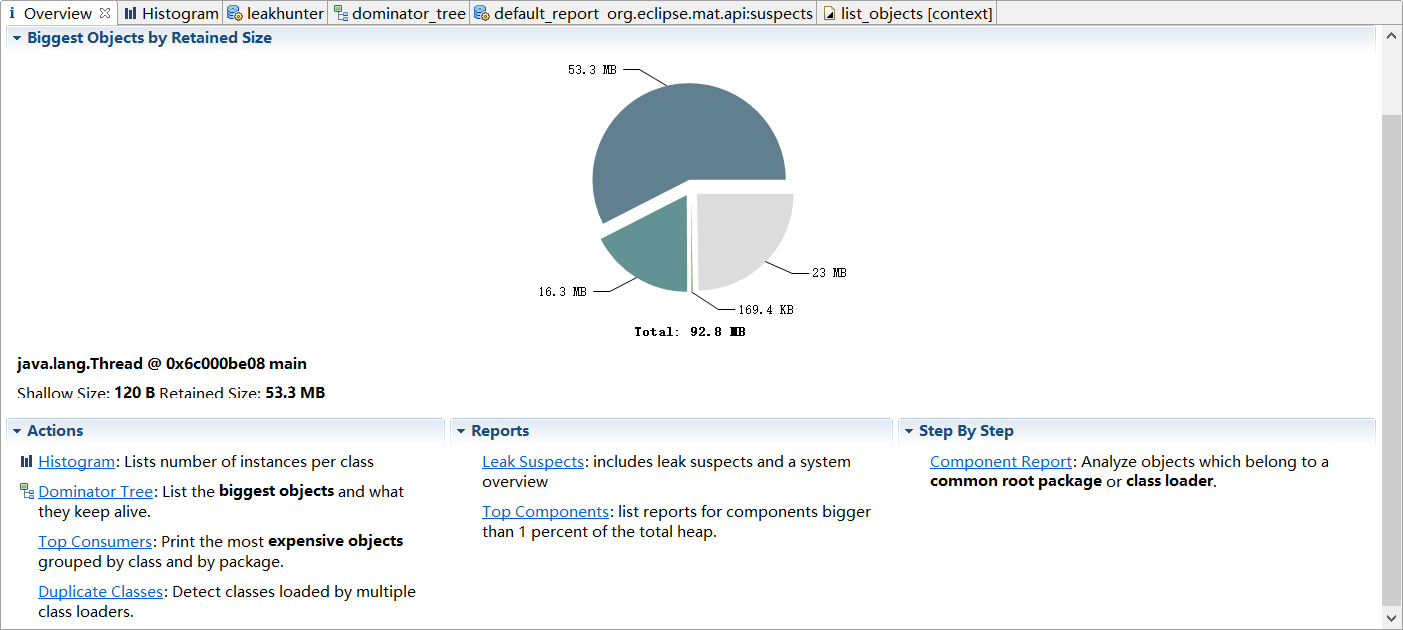
### 内存导出(memory dump)

为了避免在程序运行结束后立即退出，使用Thread.sleep()方法将程序在建图完毕的时刻停下。并利用Visual VM对CPU使用率和堆的大小进行监控，当CPU的使用率接近于0并且堆的大小几乎不变的时刻，即为程序进行“休眠”。此时导出堆文件，恰好为刚刚建图完毕的堆文件。

### 使用MAT分析内存导出文件

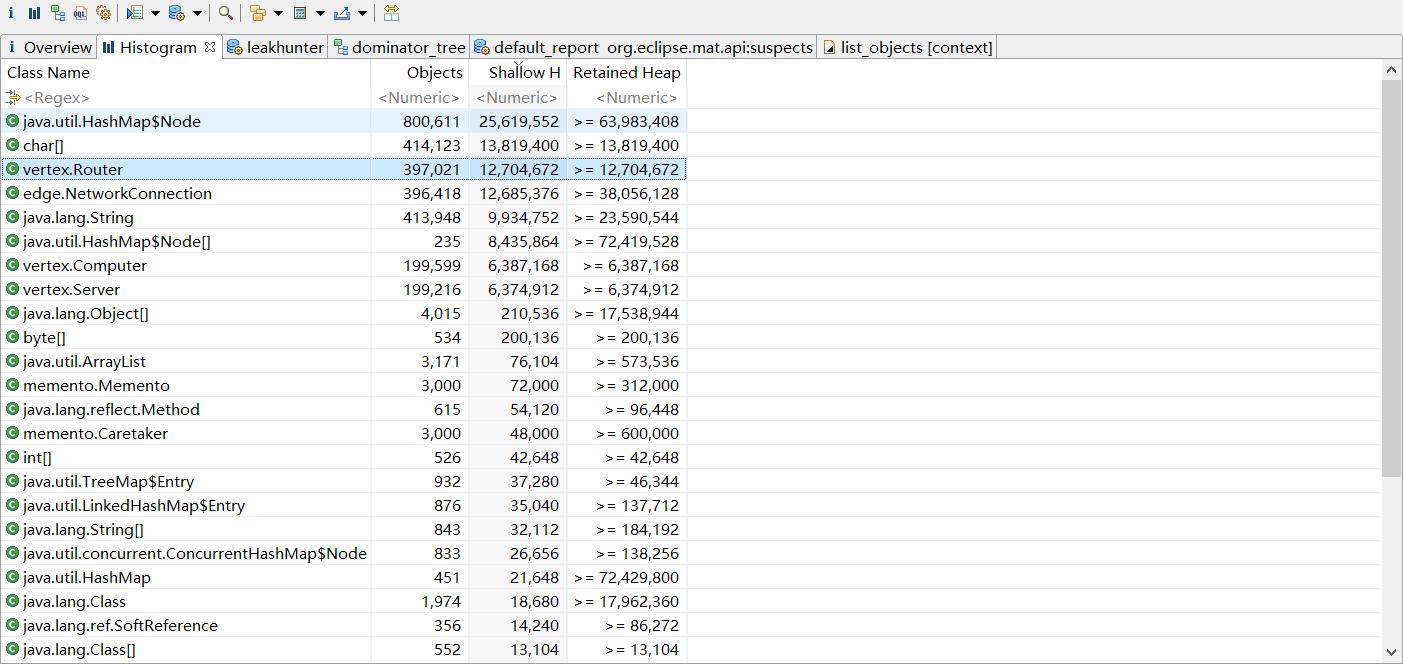
此处使用的导出堆文件为未进行优化时的程序导出的堆文件，**但是在3.5节之前的所有测试都是基于优化后的程序进行的测试！**

（1）Overview



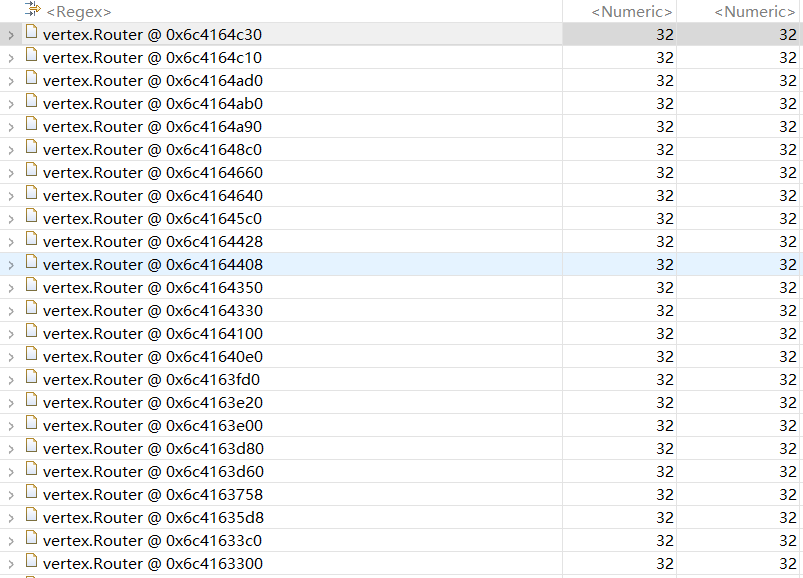
最主要的还是main线程。

（2）查看Histogram视图

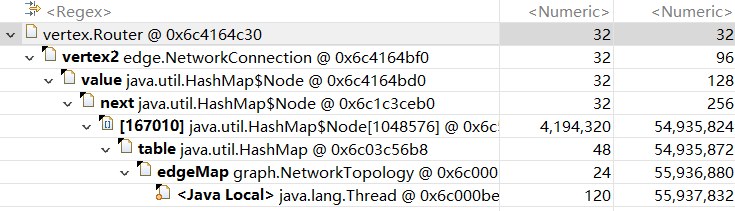


可以看到在输入文件中仅有各类顶点Router、Computer和Server都是1000个但在其中却有39万个Router实例存在。

使用list objects->with incoming refs查看该Router类的实例如下：

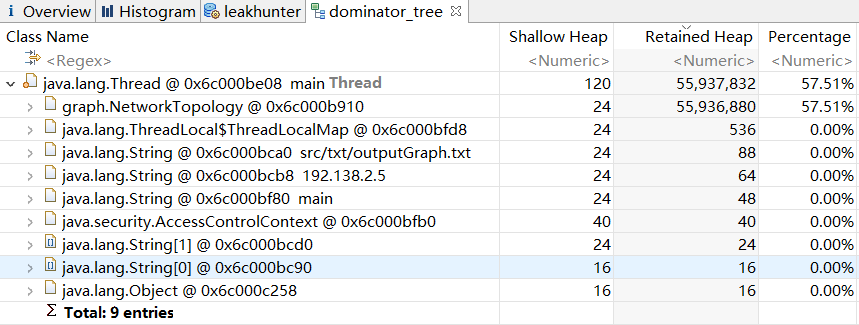


此时还不能看出来什么问题，所以进一步查看Path to GC Roots，查看结果如下：



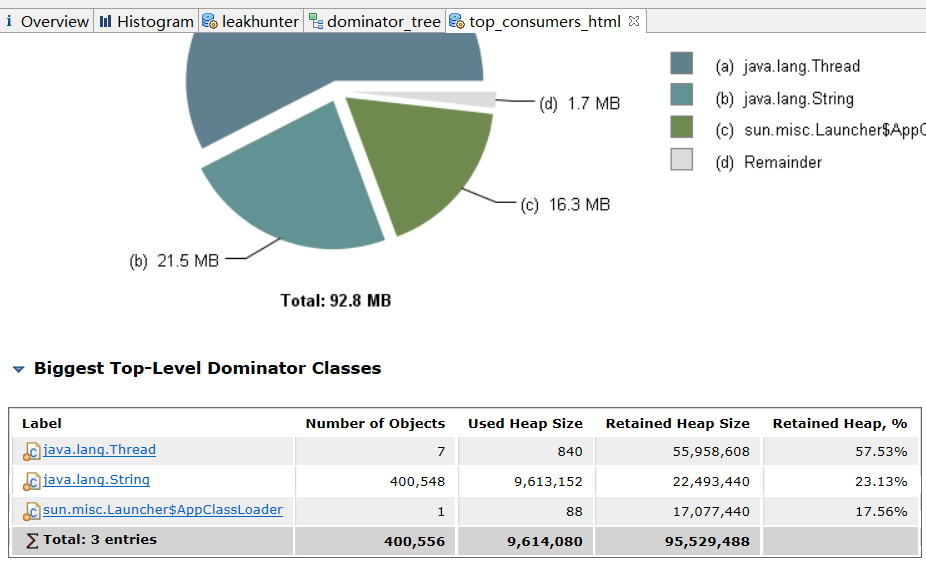
此时发现所有顶点都是作为NetworkConnection的一个顶点，进而是在主线程中，所以没有被释放。**这样来看似乎是合理的。**

（3）查看Dominator tree视图



可以看到所有的调用关系，但是问题的所在似乎还是没有解决。

（4）Top consumers视图

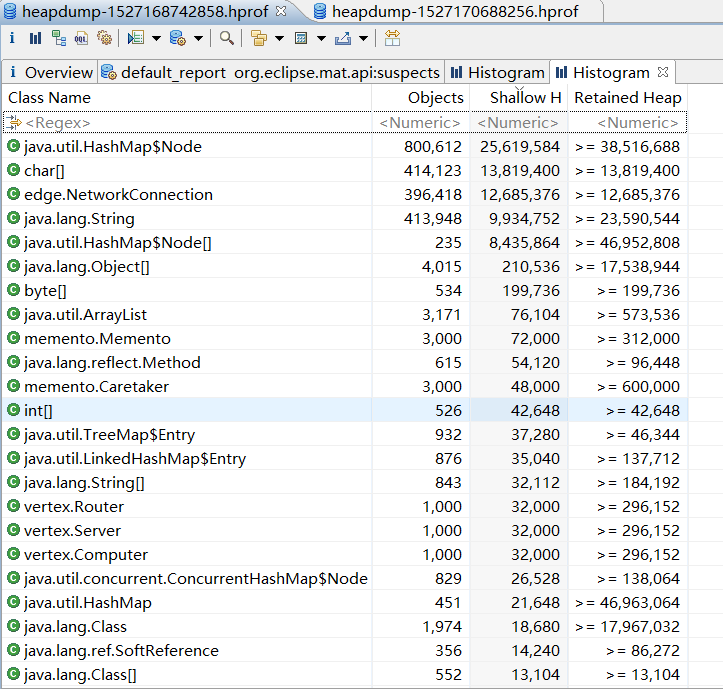


看到Top 级别的是Thread，似乎也是合理的。但是我们明显可以看到时间偏长，而且其中运用的顶点的类的实例太多。

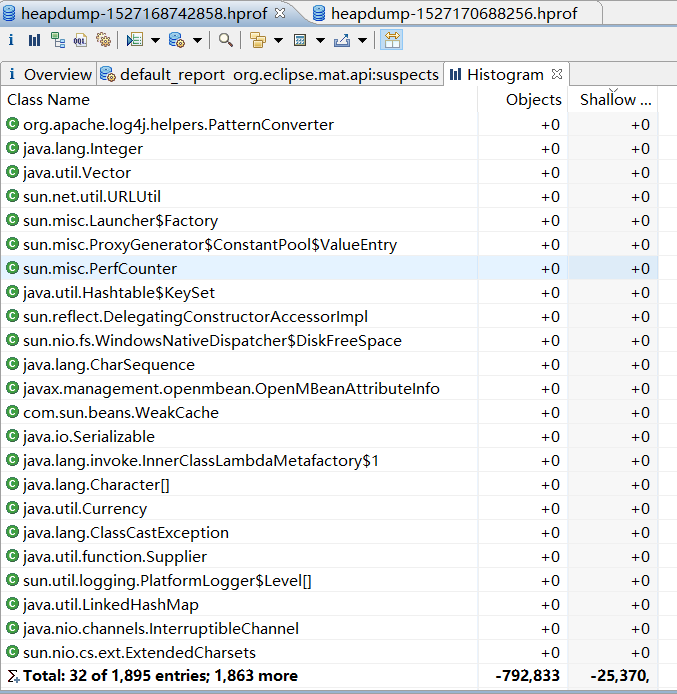
### 发现热点/瓶颈并改进、改进前后的性能对比分析

**分析代码可以发现所有的边加入其中的顶点的时候**，都进行了一次clone。也就是造成了顶点的实例**远远超过其原本应该有的实例数。**并且这种复制是没有必要的。因为复制后也不能防止内存泄露，因为我们并没有开放给用户直接使用Vertex向里面传递参数的功能，因此这种clone是没有实际意义的。基于这种考虑将原本的clone去掉。

经过上面的修改后，重新利用Visual VM导出堆文件，然后利用MAT分析，得到以下结果：



我们可以看到，经过修改后的顶点数目已经明显减少顶点实例的数目同时缩短了读图所需的时间。并将这个结果与之前没有做过优化的情况进行对比可以得到下面的结果。



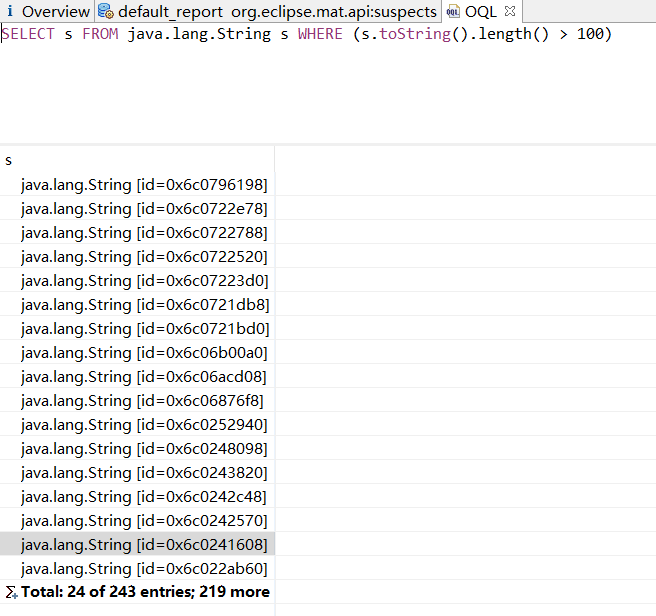
减少的顶点实例的数量恰好有792k左右，这与我们之前的目的是相符合的。

优化达到了目的。

### jhat和OQL查询内存导出（可选）

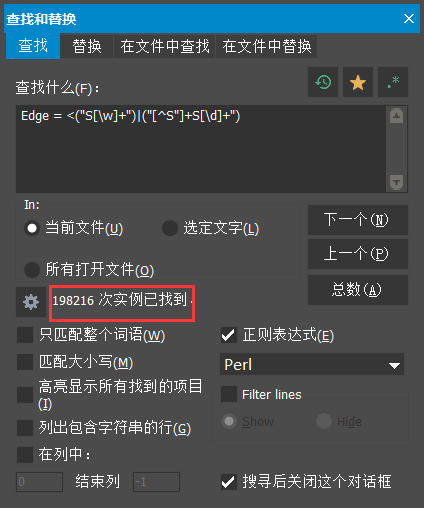
此处使用的是利用MAT中的OQL查询功能

（1）查询长度大于100的String对象



查询结果见上图。可以看到一共有243个String的对象的长度大于100.

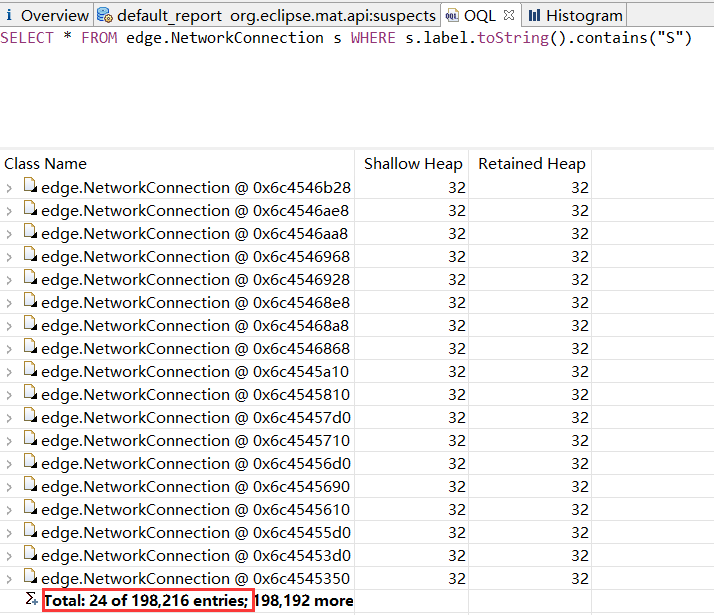
（2）查询所有包含”Server”的边的实例数量



首先用文本编辑器在file1.txt中直接查询，查询的结果是198216次，转而用MAT进行搜索，使用的搜索条件如下：

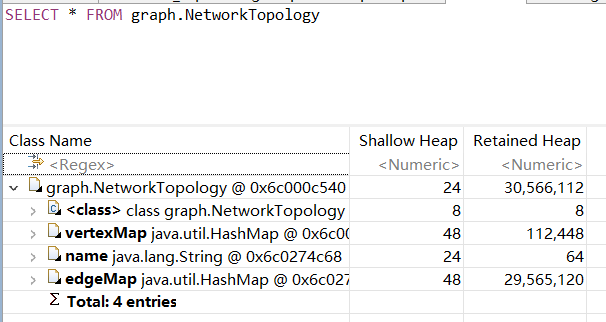
|  |
| --- |
|  |

查询的结果如下：



同样也有198216个结果，说明查询结果正确。

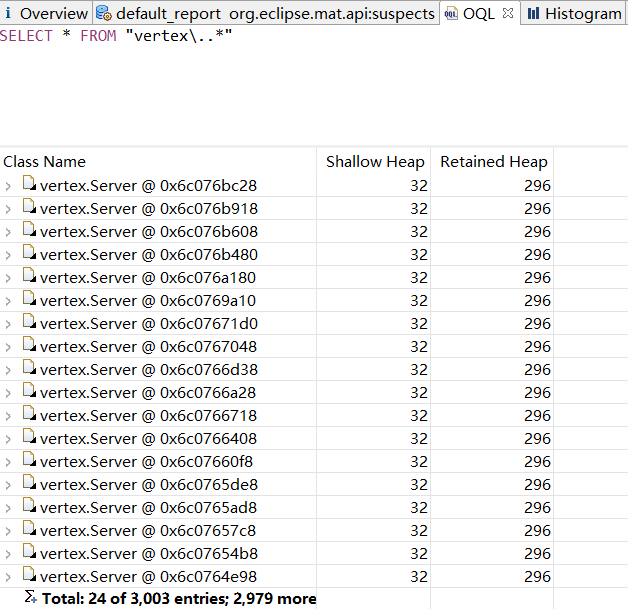
（3）查询Graph对象的个数，此处由于只是使用了一个Graph对象进行操作，所以查询的结果也仅仅有file1中使用的一个对象



（4）查询所有的Vertex（及其每个子类）的对象，由于所有的Vertex的相关类都存放在“src/vertex”包下，所以可以书写相关的查询条件如下：

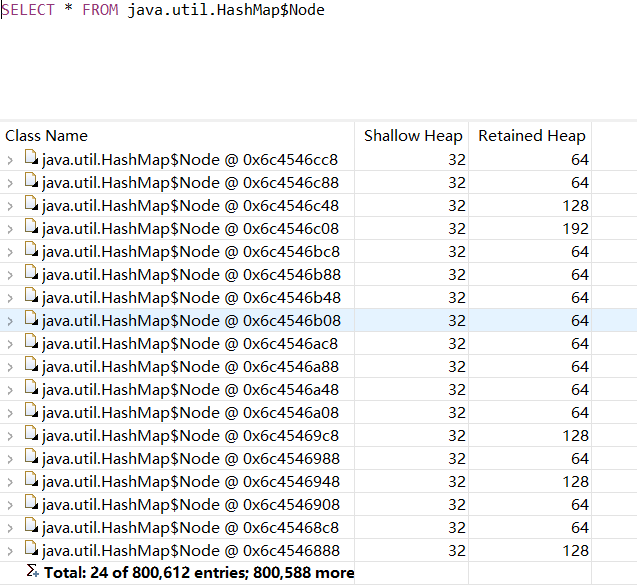
|  |
| --- |
|  |

查询的结果如下：



由于此处在操作中又另外引入3个顶点类的实例作为测试，所以此处的顶点的实例数恰好比file1文件中给出的3000个顶点多出3个。

（5）查询所有的HashMap的节点，由于大量使用了Hashmap作为存储信息的工具，所以此处查询一下到底有多少的Hashmap的节点被使用。



答案是超过了80万个。

### jstack导出java程序运行时的调用栈（可选）

（1）删除某个顶点的时候的调用栈

由于单独执行删除顶点的时间非常短，所以为了查看删除某个顶点时的调用栈，只能调用sleep函数将程序暂停，运行后的结果如下：



可以看到处理最里面的Thread.sleep函数之外，调用关系是正确的。

（2）增加某条边在NetworkTopology中，同样由于执行速度的问题，此处仍然选择使用sleep的方式进行查看其中的调用栈，具体如下：



此处的调用关系是由于NetworkTopology之中的addEdge函数重写了其父类ConcreteGraph中的相关函数，但是在最终的使用时仍然用到了其中的逻辑，调用关系正确。

# 实验进度记录

请尽可能详细的记录你的进度情况。

见git log 如下（源文件./log/git.log）

commit 015e82b9cf8aa2f7cacb65752533dfbc4591b2c0

Author: 1160300314 <hiterzmy@outlook.com>

Date: Sat May 26 21:27:44 2018 +0800

[+] Update src

增加源文件

commit aa0c86af53941e629f9c5a28bc468a316b4f38fc

Author: 1160300314 <hiterzmy@outlook.com>

Date: Sat May 26 21:26:45 2018 +0800

[+] Update doc

增加实验报告中提到的原始数据文件

commit c978d819c00791ade1d3cdeae3600e423e5506fc

Author: 1160300314 <hiterzmy@outlook.com>

Date: Wed May 23 23:01:22 2018 +0800

[+] 增加数据文件

commit 24aafa1d378a69a801bb1b00d46f2e42e4da5038

Author: 1160300314 <hiterzmy@outlook.com>

Date: Mon May 21 23:29:32 2018 +0800

[+] Update doc

commit a4964603e5913935291f264d5c05a4454dc40292

Author: 1160300314 <hiterzmy@outlook.com>

Date: Mon May 21 22:14:56 2018 +0800

[+] 实现3.2(1)中的要求 输出文件

在inputNetworkTopology.txt的基础上增加一个顶点(router2)和一条边

输出到outputgraph.txt

commit 295515a61a35c17f3b522f0603a8dc468e07a6f7

Author: 1160300314 <hiterzmy@outlook.com>

Date: Mon May 21 22:13:19 2018 +0800

[+] 完善所有checkstyle的检测

commit 089749d9b58d089ffa7937376b626a9803734aca

Author: 1160300314 <hiterzmy@outlook.com>

Date: Mon May 21 21:44:13 2018 +0800

[+] 完成3.2相关要求 使用checkstyle检查代码

使用checkstyle和findbugs分别检查代码至没有错误

commit a8328c6fdce2d83dc56ecbb58a574768234de930

Author: 1160300314 <hiterzmy@outlook.com>

Date: Mon May 21 21:34:23 2018 +0800

[+] 完成Findbugs的测试 完成3.2相关要求

改善了可能引入bug的代码

commit 9bc5266a93fc5845ddfa2d3fd31398d8c5ff3102

Author: 1160300314 <hiterzmy@outlook.com>

Date: Mon May 21 21:30:10 2018 +0800

[+] Update with lab4

commit b94e2ce1919c75ed524956fb627cc3a0fd3fb2e5

Author: 1160300314 <hiterzmy@outlook.com>

Date: Tue May 15 19:58:20 2018 +0800

[\*] Fix the bugs

commit 64f1006bed46e6fdc769965f9f271fa1d8dd4872

Author: 1160300314 <hiterzmy@outlook.com>

Date: Tue May 15 19:46:29 2018 +0800

[+] Update output graph strategy

commit ae2f77870b82c3d54c25ad406440d625fa998244

Author: 1160300314 <hiterzmy@outlook.com>

Date: Tue May 15 17:35:59 2018 +0800

[\*] Fix the order of sex and edge

commit f6360db4003039fba23959e9c9ba5cfcec0a1227

Author: 1160300314 <hiterzmy@outlook.com>

Date: Tue May 15 16:59:00 2018 +0800

[+] Update input strategy

commit 1b24a827d62dc12e2fad13c1265651b930adaeed

Author: 1160300314 <hiterzmy@outlook.com>

Date: Sat May 12 16:52:29 2018 +0800

[+] Update Almost all test cases

commit 8e10f397e94fe146fc6249a30d58d910e23d236a

Author: 1160300314 <hiterzmy@outlook.com>

Date: Sat May 12 14:35:52 2018 +0800

[+] add doc

commit 8add59043611b0d5730d2efd049019072e9666cb

Author: 1160300314 <hiterzmy@outlook.com>

Date: Sat May 12 14:32:54 2018 +0800

[-] delete target/out/output

commit 8ba0a40492169c2b76fba6b4c41271a8f06613eb

Author: 1160300314 <hiterzmy@outlook.com>

Date: Sat May 12 14:29:54 2018 +0800

[-] delete .idea

commit d199a90d3ee63532dc427033fbb247b9705b04d1

Author: 1160300314 <hiterzmy@outlook.com>

Date: Sat May 12 14:25:45 2018 +0800

[+] Update Output Graph

增加graph按照Lab3输出格式输出的方法

commit 5f3c5274926523f1ac019b632e8a1a80c264423b

Author: 1160300314 <hiterzmy@outlook.com>

Date: Thu May 10 20:11:15 2018 +0800

[+] Update maven

commit 1c11bea38d1f87cb70584a1620dc37ac2fef54a5

Author: 1160300314 <hiterzmy@outlook.com>

Date: Thu May 10 11:21:19 2018 +0800

[+] Update

完成Google Java Style的修复

增加Javadoc

完成edge和vertex的测试

commit 387e6defd3c75241a2608e77b23a89d638b6ae57

Author: 1160300314 <hiterzmy@outlook.com>

Date: Wed May 9 21:59:56 2018 +0800

Add all Lab4 into Lab5 respository

# 实验过程中遇到的困难与解决途径

实验中由于没有使用eclipse作为IDE，所以对于MAT没有相应的IDEA插件，基于此，只好利用Visual VM将堆文件导出之后，再利用eclipse中的MAT插件对导出的堆文件进行分析。

# 实验过程中收获的经验、教训、感想

本节除了总结你在实验过程中收获的经验和教训，也可就以下方面谈谈你的感受（非必须）：

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FindBugs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是否足够好？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，VisualVM和MAT这两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。

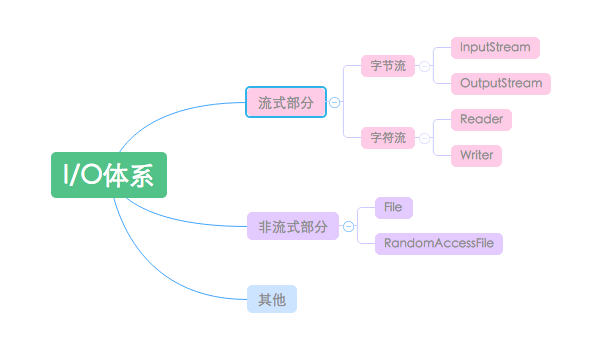
（1）“代码看起来很美”和“代码运行起来很美”没有必然的联系。“代码运行起来很美”，关注的是代码的运行效率，即我们如何将代码的算法效率、IO效率尽量提高；而“代码看起来很美”关注的是我代码的可理解性，因为在当前这个世界里任何一个程序员都不可能独立工作。足够好的代码可理解性可以更好的帮助程序员之间的交流，便于更好的提高。

不存在两者哪个更重要的比较，两者各有侧重，在关注效率的场合效率可以更重要一些。但绝大多数的程序员应该更加侧重于一般的程序的可理解性即，我们如何可以更好的交流。

可能在时间非常有限的情况下，可能更会关注可理解性。首先代码的优化是非常困难的是放在所有的工作做完的最后去完成的。也就是说我可能一次优化效率并达不到我的要求。但是我的代码的风格非常难看，这对于我后面的调试和进一步的优化都会是非常困难的。而选择较好的代码风格可以让我在以后调试的时候更容易立即自己当时的想法。

（2）首先来说一下findbugs，对于其主要是为了发现文件中潜在的bug，这样的功能对于发现未知的错误是非常有帮助的。简单的以我为例，如果不使用findbugs可能我会在近期都不会关注平台编码的问题，而这正是潜在的影响可移植性的bug所在；对于checkstyle来说，可能这更像一种习惯，对于不同的人来说习惯没有优劣之分，比如有的人就是习惯所有的花括号新起一行，但是有的人就是习惯将左花括号不新起一行。这些都是无可厚非的，只要合理就好，但是使用了checkstyle可以帮助我们始终遵循一种代码规范，这是非常有利于可理解性的提高的。

（3）Java提供多种IO策略一是为了提供给用户多种选择另外其本身也是在发展过程中不断优化IO策略的。Java的IO体系主要可以总结为一下几种：



File类主要用于管理文件的增删。

输入流选择InputStream或者Reader，输出流选择OutputStream或者Writer，如果操作对象是纯文本，一般选择Reader和Writer，而对于其余的字节流则一般选择InputStream和OutputStream。

（4）JVM的内存管理，使用栈和堆进行内存的管理，与《计算机系统》所讲的很像。特别对于重点讲的JVM的四种垃圾回收策略，在csapp中都有涉及。但是对于csapp更侧重于内存的分配和整理，而JVM的讲述中更侧重于垃圾的自动回收。

（5）JVM所提供的自动回收垃圾的策略就是针对我们这些比较菜的程序员，相比于精通c++的选手可以做到手动回收垃圾做到更好来说，Java确实做的更加适合于上手。

JVM当前采用的在young generation和old generation分别使用不同的垃圾回收策略，这种方式已经做得相当好。但并不是毫无优化的空间，比如Java允许用户选择不同的IO策略，就说明对于JVM的垃圾回收也没有做得针对任意一种情况都是最好的。

（6）在直接通过JVM参数进行调优，由于调优的策略很多不能够做到完全做到其中的要求，所以调优的效果并不十分明显，提升的效率提升在10%左右，这与利用算法进行优化以及利用IO进行优化的效果相比效果较差.

（7）利用Visual VM和MAT进行优化,可以看到其中的优化空间,以及问题之所在.另外与Jprofiler的功能相比,上述两个软件的功能还不够强大,希望以后可以引入对于JProfiler作为调优工具.

（8）调优考验内心，在短时间内修复几十个findbugs提出的问题和checkstyle几百个的问题也考验的是内心，感觉经过软件构造实验的历练，我觉得我们都具备了这种强大的内心。

（9）本次试验难度适中,比较适合在期末做.

（10）对于软件构造，希望最后一部分的实验可以有着比较详细的实验手册，为最后一次实验画上一个圆满句号。