



哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

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
手机号码	18846082306

目录

1 实验目标概述	1
2 实验环境配置	1
3 实验过程	1
3.1 Static Program Analysis	1
3.1.1 人工代码走查 (walkthrough)	1
3.1.2 使用 CheckStyle 和 FindBugs 进行静态代码分析	2
3.1.2.1 checkStyle 发现问题	2
3.1.2.2 findbugs 发现的问题	4
3.1.2.3 checkstyle 和 findbugs 的比较	5
3.2 Java I/O	5
3.2.1 多种 I/O 实现方式	5
3.2.2 多种 I/O 实现方式的效率对比分析	8
3.3 Java Memory Management and Garbage Collection (GC)	11
3.3.1 使用-verbose:gc 参数	11
3.3.2 用 jstat 命令行工具的-gc 和-gcutil 参数	12
3.3.3 使用 jmap -heap 命令行工具	13
3.3.4 使用 jmap -histo 命令行工具 (可选)	13
3.3.5 使用 jmap -permstat 命令行工具 (可选)	14
3.3.6 使用 jconsole 或 VisualVM 工具	14
3.3.7 分析垃圾回收过程是否正常、异常	16
3.3.8 配置 JVM 参数并发现最优参数配置	17
3.4 Dynamic Program Profiling	21
3.4.1 使用 Visual VM 进行 CPU Profiling	21
3.4.2 使用 Visual VM 进行 Memory profiling	22
3.5 Memory Dump Analysis and Performance Optimization	24
3.5.1 内存导出(memory dump)	24
3.5.2 使用 MAT 分析内存导出文件	24
3.5.3 发现热点/瓶颈并改进、改进前后的性能对比分析	27

3.5.4 jhat 和 OQL 查询内存导出 (可选)	29
3.5.5 jstack 导出 java 程序运行时的调用栈 (可选)	33
4 实验进度记录	34
5 实验过程中遇到的困难与解决途径	39
6 实验过程中收获的经验、教训、感想	39

1 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”。

2 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>

3 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！）。

3.1 Static Program Analysis

3.1.1 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：

```
private static String typeString = "^(GraphType|VertexType|EdgeType|GraphName) *=-  
*(\\\\"[\\w]+\\")(\", *\\\\"[\\w]+\\")* *$";
```

（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:

```
private static String typeString = "^(GraphType|VertexType|EdgeType|GraphName) *"
    + "= *(\\"[\\w]+\\")(\\"[\\w]+\\")* *$";
```

3.1.2 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>

3.1.2.1 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。

```
public class Main{
    public static void main(String [] args){
        System.out.println("Hello, world!");
    }
}
```

所以谷歌就建议使用两个空格进行缩进, 如下:

```
public class Main{
    public static void main(String [] args){
        System.out.println("Hello, world!");
    }
}
```

(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

```
public class Main{
    /**
     * For some test.
     */
    public static void main(String [] args){
        System.out.println("Hello, world!");
    }
}
```

(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:

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello world!");
        //      System.err.println("hello world");
    }
}
```

而应该将注释的位置向后移动, 转为如下形式:

```
public class Main {  
    public static void main(String[] args) {  
        System.out.println("Hello world!");  
        //System.err.println("hello world");  
    }  
}
```

(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:

```
import static org.junit.Assert.*;
```

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

```
import static org.junit.Assert.assertEquals;
```

3.1.2.2 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:

```
InputStreamReader inputStreamReader  
    = new InputStreamReader(new FileInputStream(fileName));
```

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

```
InputStreamReader inputStreamReader  
    = new InputStreamReader(new FileInputStream(fileName), "utf-8");
```

就可以避免这种潜在 bug 的引入。

3.1.2.3 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。

3.2 Java I/O

3.2.1 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：


```
public interface InputStrategy {

    /**
     * Get the path of file inputting.
     *
     * @return file path
     */
    public String getFilePath();

    /**
     * Read file by each line.
     *
     * @return the whole line with "\n"
     * @throws IOException if there is some IO errors.
     */
    public String readLine() throws IOException;

    /**
     * Get current number of line.
     *
     * @return line number
     */
    public long getLineNumber();

    /**
     * Close the stream.
     */
    public void close() throws IOException;
}
```

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：

```
public interface OutputStrategy {  
  
    /**  
     * Get the path of file outputting.  
     *  
     * @return file path  
     */  
    public String getFilePath();  
  
    /**  
     * Write string in file.  
     *  
     * @param string the output string.  
     * @throws IOException if there is no such file.  
     */  
    public void write(String string) throws IOException;  
  
    /**  
     * Close the stream.  
     */  
    public void close() throws IOException;  
}
```

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`。

```
// Lab3-Lab4 使用的工厂函数
public static Graph createGraph(String filePath)
    throws InputFileAgainException, IOException

// 原工厂函数的Adaptor 可以选择输入的策略
public static Graph createGraphStrategy(
    InputStrategy inputStrategy, String filePath)
    throws InputFileAgainException, IOException
```

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：

```
// 可以选择输出策略的输出图函数
public static void outputGraphStrategy(Graph graph, String fileName,
    OutputStrategy outputStrategy) throws IOException

// 默认使用的不带有参数的outputGraph
public static void outputGraph(Graph graph, String fileName) throws IOException
```

3.2.2 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。

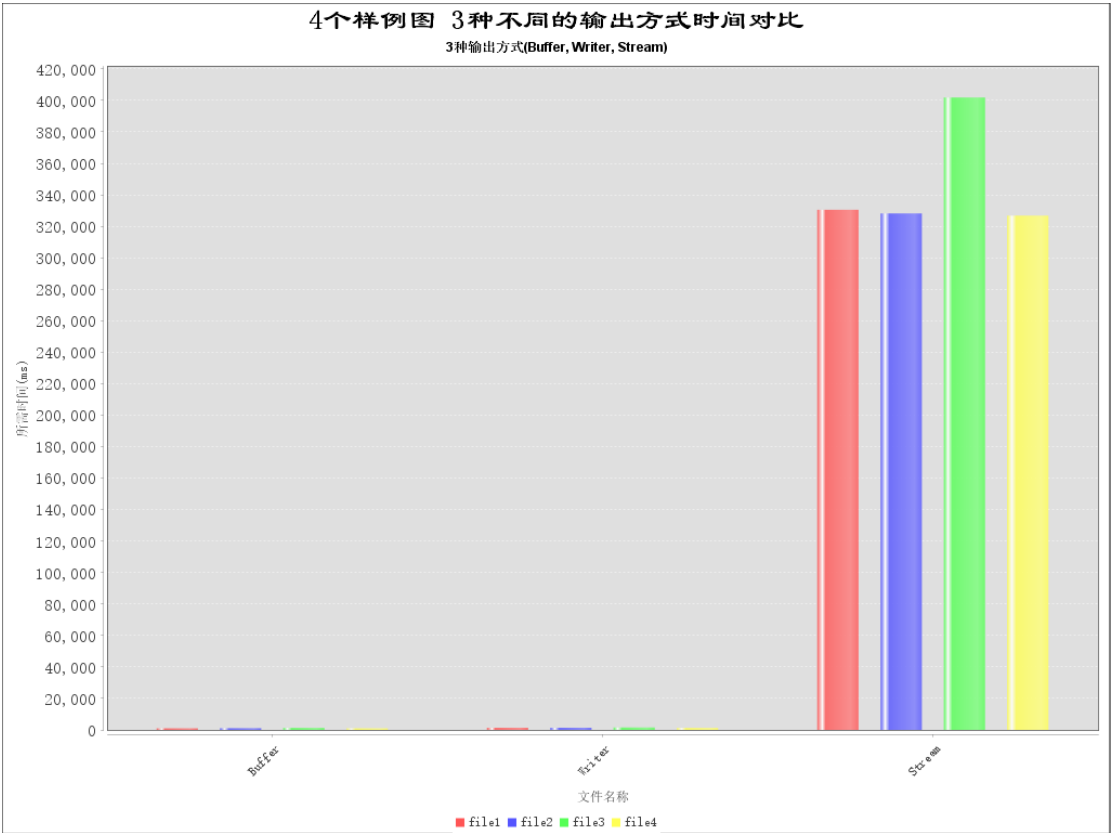
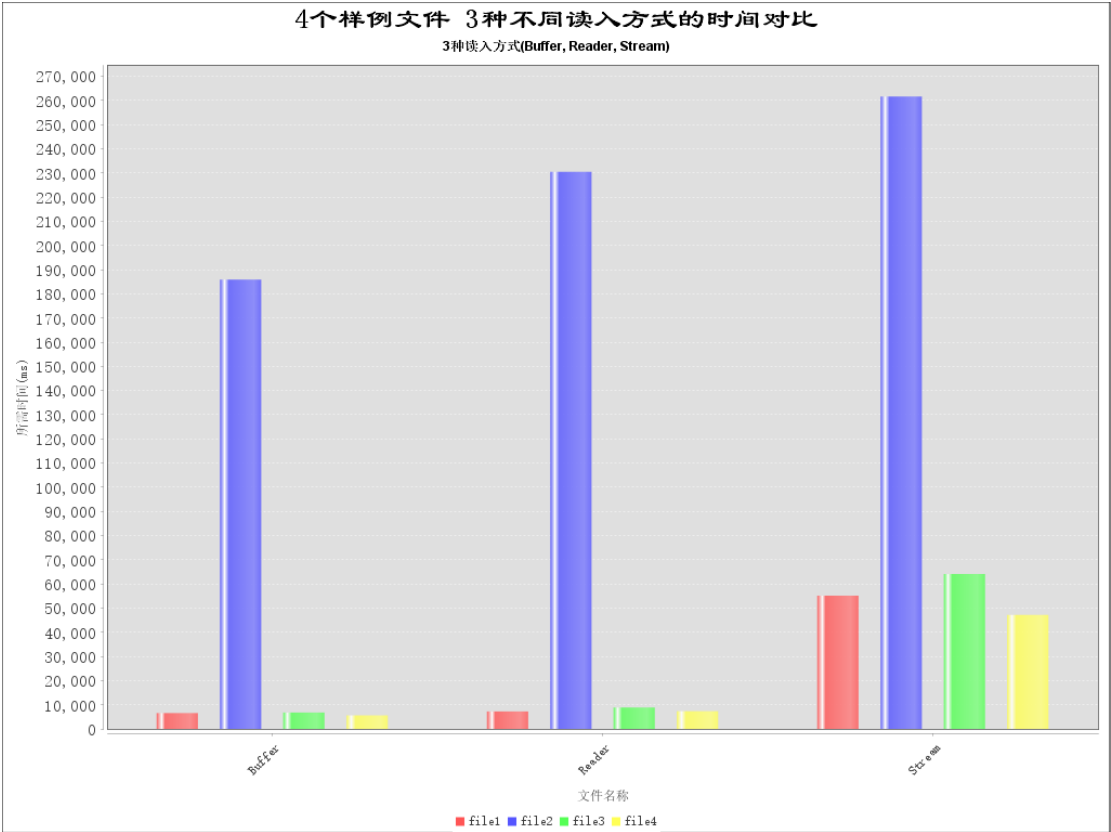
```
public class Main {  
  
    /**  
     * For some test.  
     */  
  
    public static void main(String[] args) throws IOException, InputFileAgainException {  
        long startTime, endTime;  
        startTime = System.currentTimeMillis();  
        Graph graph  
            = GraphFactory.createGraphStrategy("src/txt/Spring2018_HITCS_SC_Lab5/file1.txt");  
        endTime = System.currentTimeMillis();  
        System.err.println("Read " + graph.getGraphName() + " " + (endTime - startTime));  
    }  
}
```

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。

3.3 Java Memory Management and Garbage Collection (GC)

3.3.1 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：

```
-XX:InitialHeapSize=1073741824  
-XX:MaxHeapSize=2147483648  
-XX:+PrintGC  
-XX:+PrintGCDetails  
-XX:+PrintGCTimeStamps  
-XX:ReservedCodeCacheSize=1073741824  
-XX:+UseCompressedClassPointers  
-XX:+UseCompressedOops  
-XX:-UseLargePagesIndividualAllocation  
-XX:+UseParallelGC
```

(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%。

3.3.2 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。

S0	S1	E	O	M	CCS	YGC	YGCT	FGC	FGCT	GCT
0	57.81	89.99	33.55	95.45	94.34	39	1.662	1	0.322	1.985
59.55	0	22	37.45	95.45	94.34	40	1.706	1	0.322	2.028

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。

S0	S1	E	O	M	CCS	YGC	YGCT	FGC	FGCT	GCT
42.36	29.54	100	92.64	95.49	94.34	53	2.29	1	0.322	2.612
0	99.16	0	93.98	95.49	94.34	53	2.332	2	0.322	2.655
0	0	43.91	8.94	95.49	94.34	53	2.332	2	0.61	2.942

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。

3.3.3 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%。

3.3.4 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：

排名	实例数目	占用内存	类名
1	4470455	435806960	[C
2	2526832	164400896	[I
3	1459670	46709440	java.util.HashMap\$Node
4	681665	43626560	java.util.regex.Matcher
5	1701639	40839336	java.lang.String
6	409008	29448576	java.util.regex.Pattern
7	723237	23143584	edge.NetworkConnection
8	409000	22904000	[Ljava.util.regex.Pattern\$GroupHead;
9	68173	18542360	[Z
10	72	14726512	[Ljava.util.HashMap\$Node;

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。

3.3.5 使用 jmap -permstat 命令行工具（可选）

class_loader	classes	bytes	parent_loader	alive?	type
<bootstrap>	811	1476436	null	live	<internal>
0x0000000080079680	1	880	null	dead	sun/reflect/DelegatingClassLoader@0x0000000100009df8
0x0000000080018648	196	398506	0x00000000800186a8	live	sun/misc/Launcher\$AppClassLoader@0x000000010000f6a0
0x00000000800186a8	8	15664	null	live	sun/misc/Launcher\$ExtClassLoader@0x000000010000fa48
0x000000008013d4d8	0	0	0x0000000080018648	dead	java/util/ResourceBundle\$RBCClassLoader@0x000000010008e9b8

使用 jmap -clstats 可以看到以上的结果。

3.3.6 使用 jconsole 或 VisualVM 工具

使用 Visual VM 可以得到以下图例

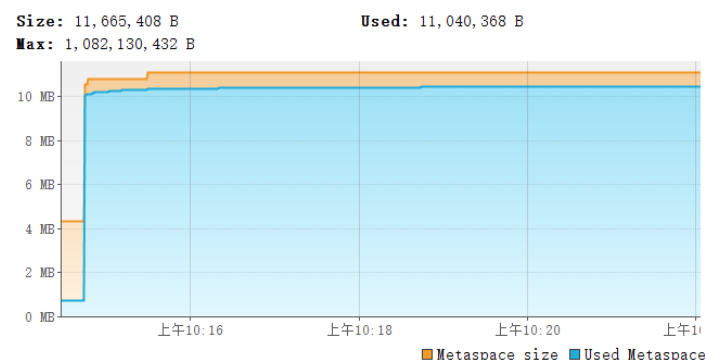


可以看到在使用当前参数的情况下：

```
-verbose:gc
-Xms1024m
-Xmx2048m
-XX:ReservedCodeCacheSize=1024m
-XX:+UseCompressedOops
-Xloggc:E:\test\gcTestVisualVm.log
```

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。

3.3.7 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。

```
-verbose:gc
-Xms1024m
-Xmx2048m
-XX:ReservedCodeCacheSize=1024m
-XX:+UseCompressedOops
-Xloggc:E:\test\gcTestVisualVm.log
```

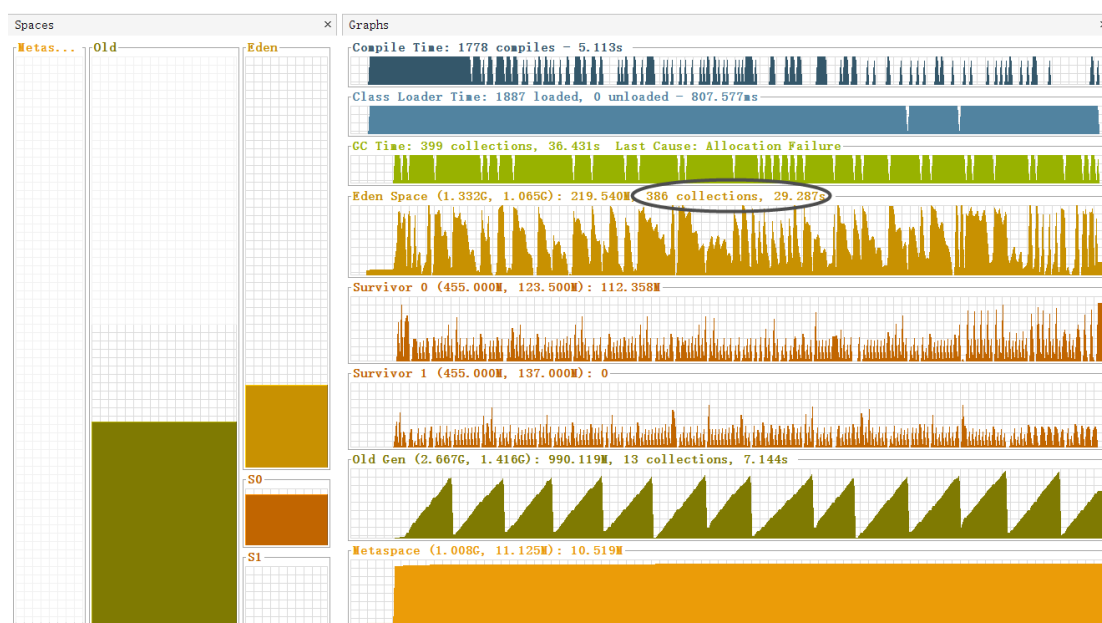
针对以上问题在 3.3.8 节进行优化。

3.3.8 配置 JVM 参数并发现最优参数配置

(1) 调整参数为以下：

```
-verbose:gc  
-Xms2048m  
-Xmx4g  
-XX:MetaspaceSize=20m  
-XX:MaxMetaspaceSize=30m  
-XX:ReservedCodeCacheSize=1024m  
-XX:+UseCompressedOops  
-Xloggc:E:\test\gcTestVisualVm.log
```

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

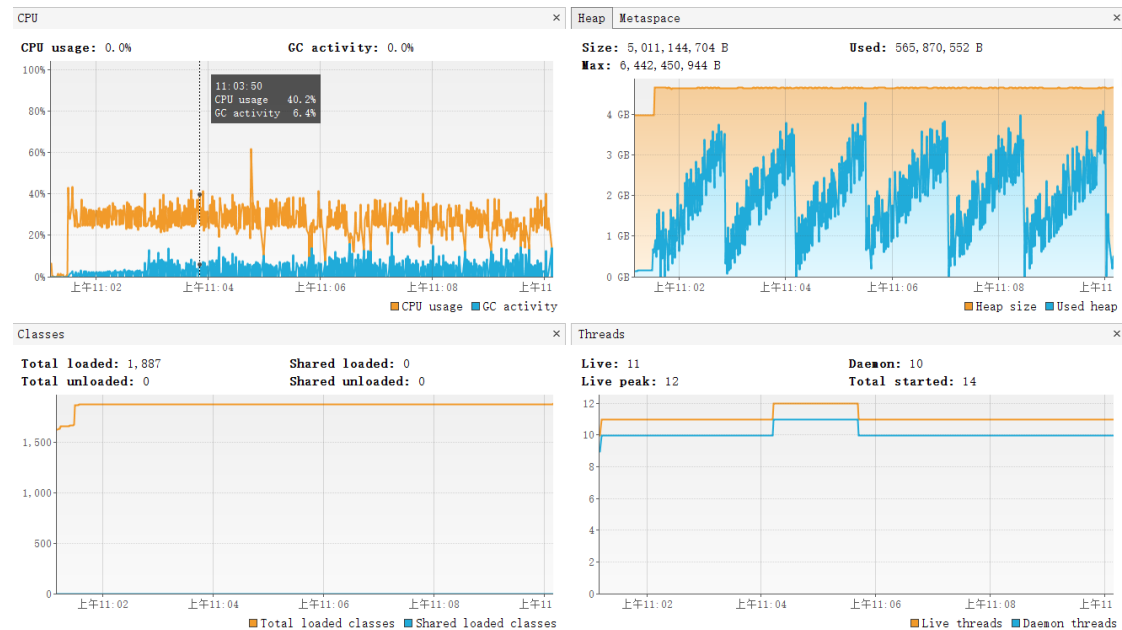


可以看到 Young Gen 的 Minor GC 进行的仍然过于频繁，基于此可以推断 new Size 过小，因此修改参数进一步进行试验。

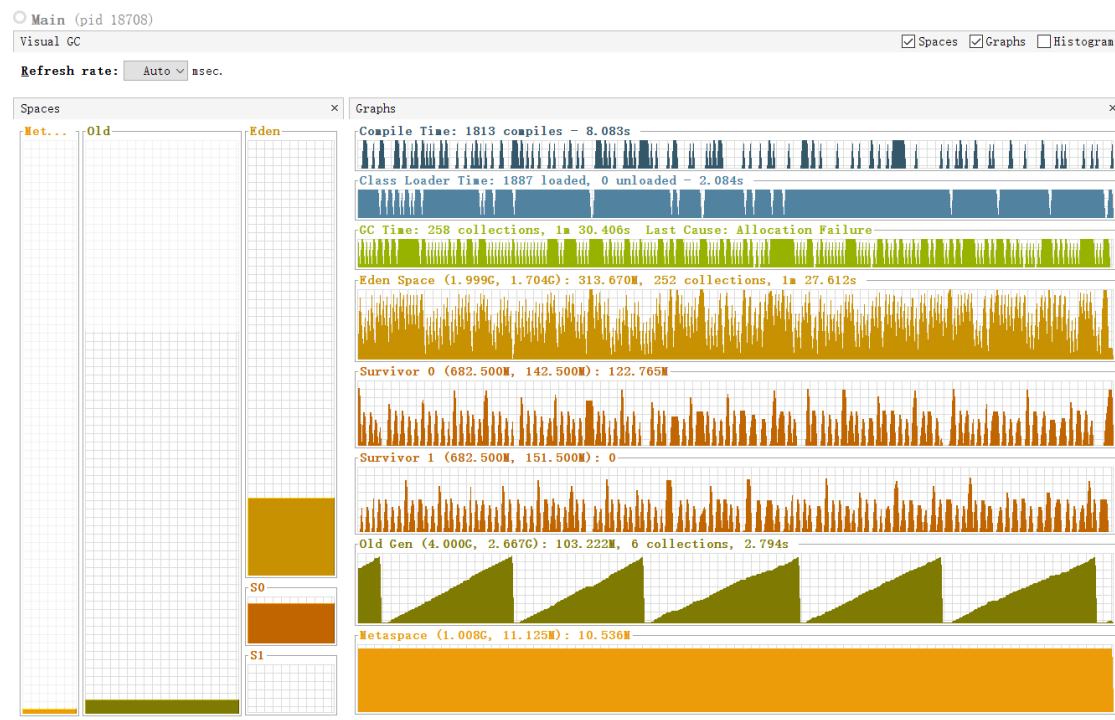
(2) 调整参数，增大 Young Gen 的规模，具体参数如下：

```
-verbose:gc  
-Xms4g  
-Xmx6g  
-XX:MetaspaceSize=20m  
-XX:MaxMetaspaceSize=30m  
-XX:ReservedCodeCacheSize=1024m  
-XX:+UseCompressedOops  
-Xloggc:E:\test\gcTestVisualVm.log
```

然后时间运行的更加慢了，具体的原因可以看到：



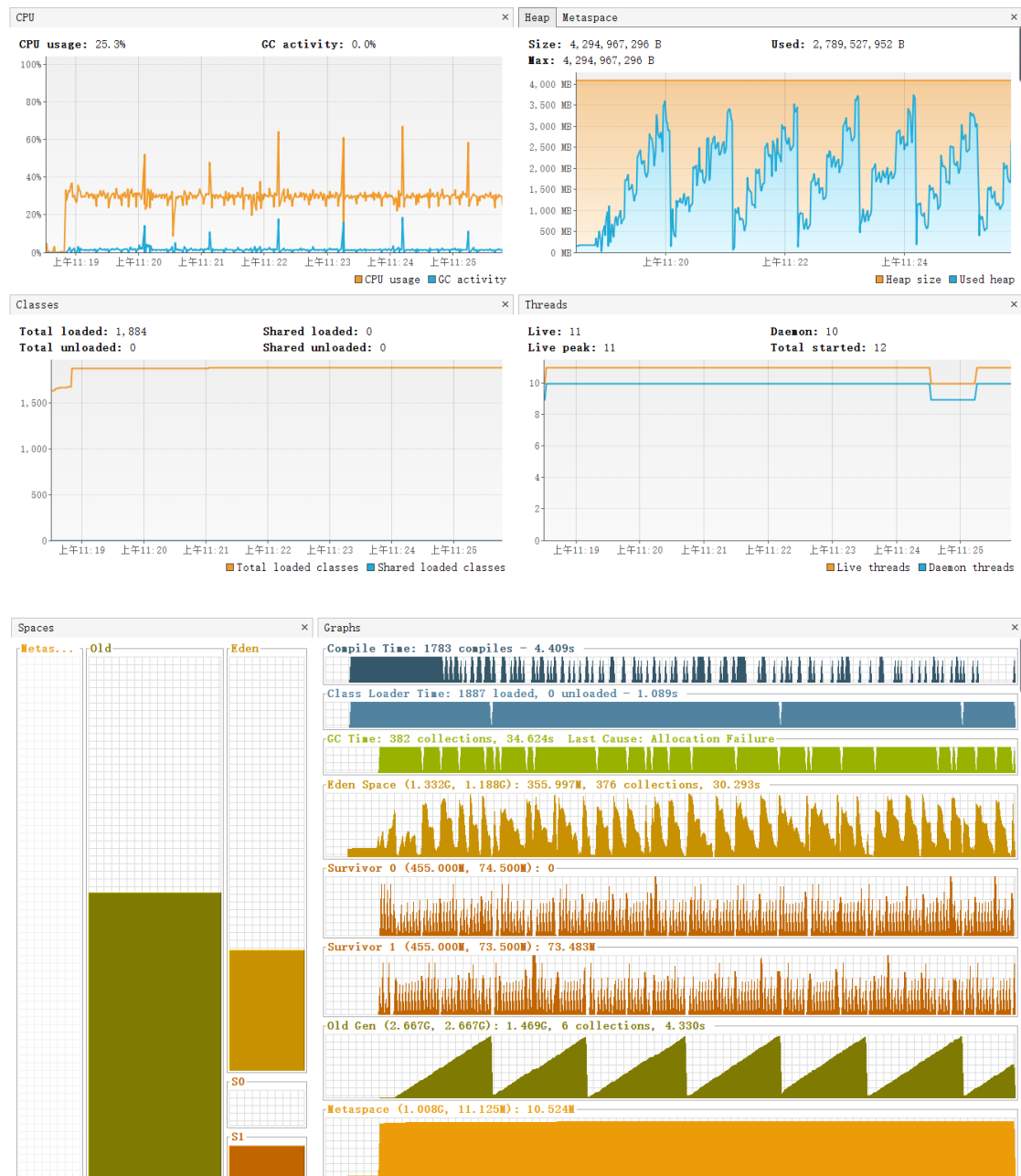
由于每一次 Full GC 耗费的时间和 CPU 过长，导致整体的运行时间不但没有相比增大之前下降。GC 时间如下：



(3) 调整参数至如下：

```
-verbose:gc
-Xms4g
-Xmx4g
-XX:MetaspaceSize=20m
-XX:MaxMetaspaceSize=30m
-XX:ReservedCodeCacheSize=1024m
-XX:+UseCompressedOops
-Xloggc:E:\test\gcTestVisualVm.log
```

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。

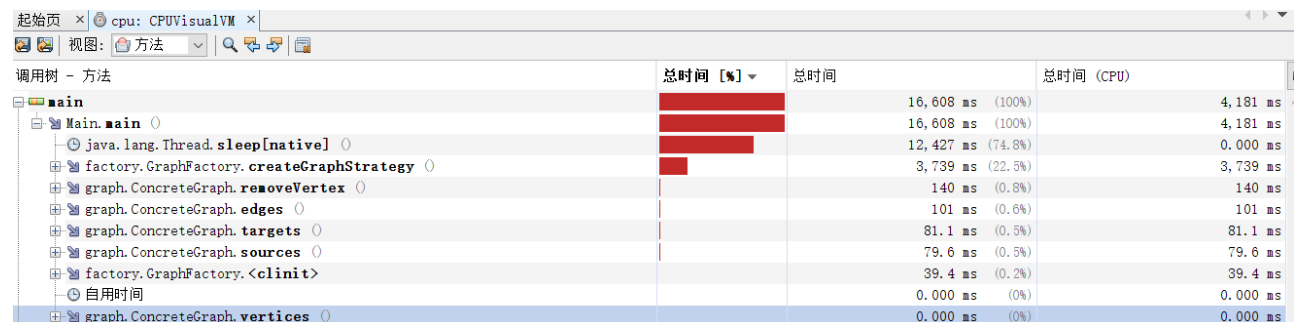


最终的运行时间从 7min23s 下降到 7min18s，可以看到主要优化的时间就是在 FullGC 少了 7 次，减少了 3s 左右。此时的配置应该已经接近最优的 JVM 配置，因为更低配置运行和更高配置的运行速度都有所下降。

3.4 Dynamic Program Profiling

3.4.1 使用 Visual VM 进行 CPU Profiling

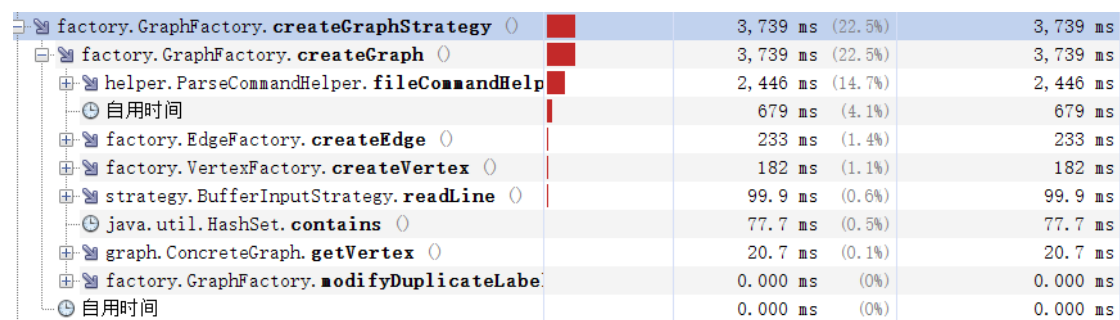
使用 Visual VM 启动程序读入 file1 对图进行 CPU profiler。可以得到下图：



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。

3.4.2 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：

类名 - 活动对象	活动字节 [%] ▼	活动字节	活动对象
java.util.HashMap\$Node		26,557,95... (28.4%)	829,936 (35.7%)
int[]		14,375,31... (15.4%)	2,802 (0.1%)
char[]		13,962,63... (14.9%)	416,323 (17.9%)
edge.NetworkConnection		12,684,96... (13.6%)	396,405 (17.1%)
java.lang.String		9,987,504 B (10.7%)	416,146 (17.9%)
java.util.HashMap\$Node[]		9,609,568 B (10.3%)	14,913 (0.6%)
java.lang.Object[]		2,398,976 B (2.6%)	93,177 (4%)
java.lang.Integer		1,412,144 B (1.5%)	88,259 (3.8%)
java.util.HashMap		733,344 B (0.8%)	15,278 (0.7%)
java.util.HashSet		237,456 B (0.3%)	14,841 (0.6%)
java.lang.Class		230,688 B (0.2%)	2,042 (0.1%)
byte[]		217,920 B (0.2%)	585 (0%)
java.lang.reflect.Method		136,136 B (0.1%)	1,547 (0.1%)
java.util.ArrayList		76,488 B (0.1%)	3,187 (0.1%)
memento.Memento		72,072 B (0.1%)	3,003 (0.1%)
java.lang.reflect.Field		49,320 B (0.1%)	685 (0%)
memento.Caretaker		48,048 B (0.1%)	3,003 (0.1%)
java.lang.Class[]		40,504 B (0%)	1,780 (0.1%)
java.util.TreeMap\$Entry		37,280 B (0%)	932 (0%)
java.util.LinkedHashMap\$Entry		35,360 B (0%)	884 (0%)
java.lang.String[]		32,112 B (0%)	843 (0%)
vertex.Computer		32,064 B (0%)	1,002 (0%)
vertex.Server		32,032 B (0%)	1,001 (0%)
vertex.Router		32,000 B (0%)	1,000 (0%)
java.util.concurrent.ConcurrentHashMap\$Node		26,848 B (0%)	839 (0%)
java.lang.ref.SoftReference		24,960 B (0%)	624 (0%)
java.lang.reflect.Constructor		22,160 B (0%)	277 (0%)
java.util.Hashtable\$Entry[]		15,784 B (0%)	222 (0%)
java.lang.Class\$ReflectionData		15,008 B (0%)	268 (0%)
java.lang.invoke.MemberName		12,096 B (0%)	216 (0%)
java.lang.ref.WeakReference		11,744 B (0%)	367 (0%)
java.net.URL		11,072 B (0%)	173 (0%)
sun.misc.FDBigInteger		10,912 B (0%)	341 (0%)
java.util.Vector		10,336 B (0%)	323 (0%)
java.util.Hashtable		10,272 B (0%)	214 (0%)

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：

vertex.Computer	32,064 B (0%)	1,002 (0%)
vertex.Server	32,032 B (0%)	1,001 (0%)
vertex.Router	32,000 B (0%)	1,000 (0%)

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。

3.5 Memory Dump Analysis and Performance Optimization

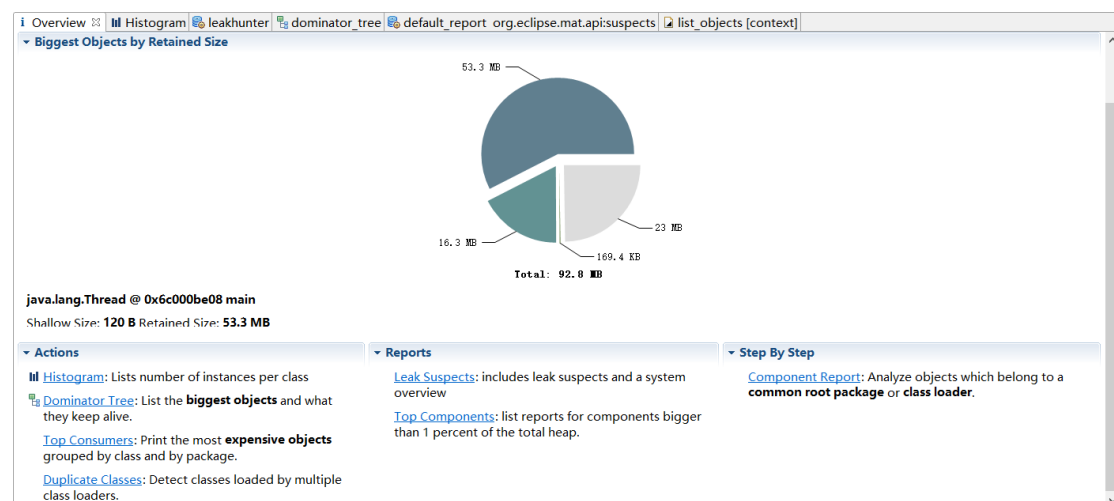
3.5.1 内存导出(memory dump)

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。

3.5.2 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

Class Name	Objects	Shallow H	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
java.util.HashMap\$Node	800,611	25,619,552	>= 63,983,408
char[]	414,123	13,819,400	>= 13,819,400
vertex.Router	397,021	12,704,672	>= 12,704,672
edge.NetworkConnection	396,418	12,685,376	>= 38,056,128
java.lang.String	413,948	9,934,752	>= 23,590,544
java.util.HashMap\$Node[]	235	8,435,864	>= 72,419,528
vertex.Computer	199,599	6,387,168	>= 6,387,168
vertex.Server	199,216	6,374,912	>= 6,374,912
java.lang.Object[]	4,015	210,536	>= 17,538,944
byte[]	534	200,136	>= 200,136
java.util.ArrayList	3,171	76,104	>= 573,536
memento.Memento	3,000	72,000	>= 312,000
java.lang.reflect.Method	615	54,120	>= 96,448
memento.Caretaker	3,000	48,000	>= 600,000
int[]	526	42,648	>= 42,648
java.util.TreeMap\$Entry	932	37,280	>= 46,344
java.util.LinkedHashMap\$Entry	876	35,040	>= 137,712
java.lang.String[]	843	32,112	>= 184,192
java.util.concurrent.ConcurrentHashMap\$Node	833	26,656	>= 138,256
java.util.HashMap	451	21,648	>= 72,429,800
java.lang.Class	1,974	18,680	>= 17,962,360
java.lang.ref.SoftReference	356	14,240	>= 86,272
java.lang.Class[]	552	13,104	>= 13,104

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：

<Regex>	<Numeric>	<Numeric>
> vertex.Router @ 0x6c4164c30	32	32
> vertex.Router @ 0x6c4164c10	32	32
> vertex.Router @ 0x6c4164ad0	32	32
> vertex.Router @ 0x6c4164ab0	32	32
> vertex.Router @ 0x6c4164a90	32	32
> vertex.Router @ 0x6c41648c0	32	32
> vertex.Router @ 0x6c4164660	32	32
> vertex.Router @ 0x6c4164640	32	32
> vertex.Router @ 0x6c41645c0	32	32
> vertex.Router @ 0x6c4164428	32	32
> vertex.Router @ 0x6c4164408	32	32
> vertex.Router @ 0x6c4164350	32	32
> vertex.Router @ 0x6c4164330	32	32
> vertex.Router @ 0x6c4164100	32	32
> vertex.Router @ 0x6c41640e0	32	32
> vertex.Router @ 0x6c4163fd0	32	32
> vertex.Router @ 0x6c4163e20	32	32
> vertex.Router @ 0x6c4163e00	32	32
> vertex.Router @ 0x6c4163d80	32	32
> vertex.Router @ 0x6c4163d60	32	32
> vertex.Router @ 0x6c4163758	32	32
> vertex.Router @ 0x6c41635d8	32	32
> vertex.Router @ 0x6c41633c0	32	32
> vertex.Router @ 0x6c4163300	32	32

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：

<Regex>	<Numeric>	<Numeric>
vertex.Router @ 0x6c4164c30	32	32
vertex2 edge.NetworkConnection @ 0x6c4164bf0	32	96
value java.util.HashMap\$Node @ 0x6c4164bd0	32	128
next java.util.HashMap\$Node @ 0x6c1c3ceb0	32	256
[167010] java.util.HashMap\$Node[1048576] @ 0x6c...	4,194,320	54,935,824
table java.util.HashMap @ 0x6c03c56b8	48	54,935,872
edgeMap graph.NetworkTopology @ 0x6c000...	24	55,936,880
<Java Local> java.lang.Thread @ 0x6c000be...	120	55,937,832

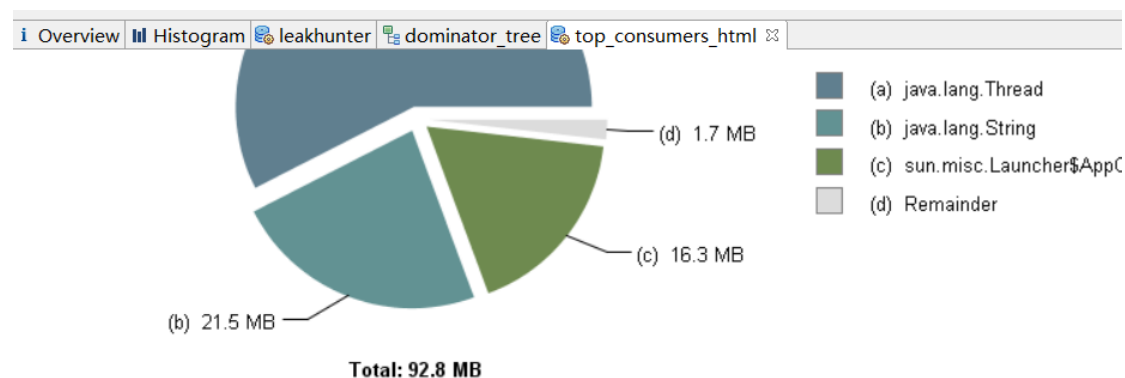
此时发现所有顶点都是作为 NetworkConnection 的一个顶点，进而是在主线程中，所以没有被释放。这样来看似乎是合理的。

(3) 查看 Dominator tree 视图

Class Name	Shallow Heap <Numeric>	Retained Heap <Numeric>	Percentage <Numeric>
java.lang.Thread @ 0x6c000be08 main Thread	120	55,937,832	57.51%
graph.NetworkTopology @ 0x6c000b910	24	55,936,880	57.51%
java.lang.ThreadLocal\$ThreadLocalMap @ 0x6c000bfd8	24	536	0.00%
java.lang.String @ 0x6c000bca0 src/txt/outputGraph.txt	24	88	0.00%
java.lang.String @ 0x6c000bcb8 192.138.2.5	24	64	0.00%
java.lang.String @ 0x6c000bf80 main	24	48	0.00%
java.security.AccessControlContext @ 0x6c000bfb0	40	40	0.00%
java.lang.String[1] @ 0x6c000bcd0	24	24	0.00%
java.lang.String[0] @ 0x6c000bc90	16	16	0.00%
java.lang.Object @ 0x6c000c258	16	16	0.00%
Σ Total: 9 entries			

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



▼ Biggest Top-Level Dominator Classes

Label	Number of Objects	Used Heap Size	Retained Heap Size	Retained Heap, %
java.lang.Thread	7	840	55,958,608	57.53%
java.lang.String	400,548	9,613,152	22,493,440	23.13%
sun.misc.Launcher\$AppClassLoader	1	88	17,077,440	17.56%
Σ Total: 3 entries	400,556	9,614,080	95,529,488	

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。

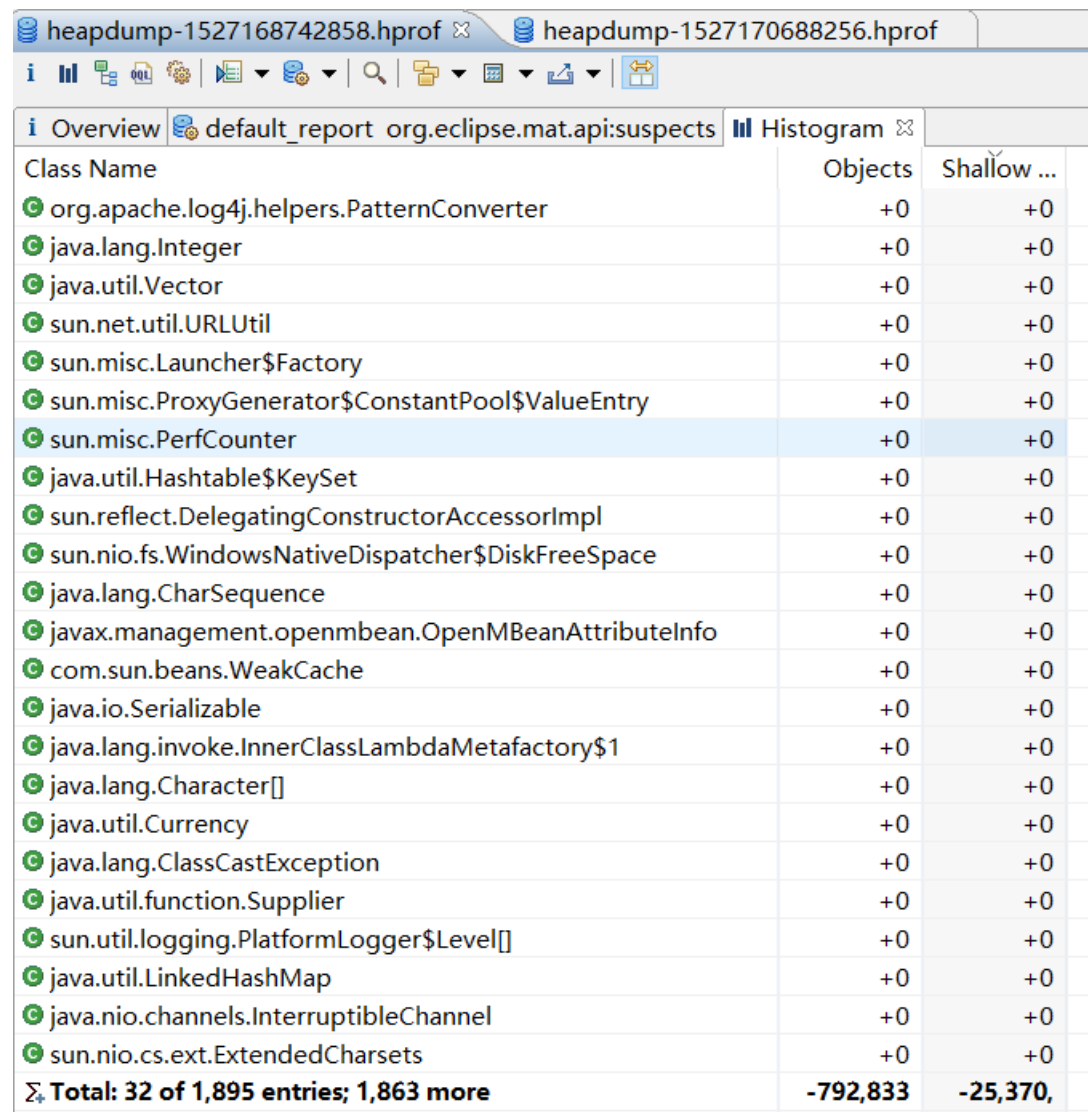
3.5.3 发现热点/瓶颈并改进、改进前后的性能对比分析

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：

heapdump-1527168742858.hprof			
heapdump-1527170688256.hprof			
i Overview default_report org.eclipse.mat.api:suspects Histogram Histogram			
Class Name	Objects	Shallow H	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
java.util.HashMap\$Node	800,612	25,619,584	>= 38,516,688
char[]	414,123	13,819,400	>= 13,819,400
edge.NetworkConnection	396,418	12,685,376	>= 12,685,376
java.lang.String	413,948	9,934,752	>= 23,590,544
java.util.HashMap\$Node[]	235	8,435,864	>= 46,952,808
java.lang.Object[]	4,015	210,536	>= 17,538,944
byte[]	534	199,736	>= 199,736
java.util.ArrayList	3,171	76,104	>= 573,536
memento.Memento	3,000	72,000	>= 312,000
java.lang.reflect.Method	615	54,120	>= 96,448
memento.Caretaker	3,000	48,000	>= 600,000
int[]	526	42,648	>= 42,648
java.util.TreeMap\$Entry	932	37,280	>= 46,344
java.util.LinkedHashMap\$Entry	876	35,040	>= 137,712
java.lang.String[]	843	32,112	>= 184,192
vertex.Router	1,000	32,000	>= 296,152
vertex.Server	1,000	32,000	>= 296,152
vertex.Computer	1,000	32,000	>= 296,152
java.util.concurrent.ConcurrentHashMap\$Node	829	26,528	>= 138,064
java.util.HashMap	451	21,648	>= 46,963,064
java.lang.Class	1,974	18,680	>= 17,967,032
java.lang.ref.SoftReference	356	14,240	>= 86,272
java.lang.Class[]	552	13,104	>= 13,104

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。



Class Name	Objects	Shallow ...
org.apache.log4j.helpers.PatternConverter	+0	+0
java.lang.Integer	+0	+0
java.util.Vector	+0	+0
sun.net.util.URLUtil	+0	+0
sun.misc.Launcher\$Factory	+0	+0
sun.misc.ProxyGenerator\$ConstantPool\$ValueEntry	+0	+0
sun.misc.PerfCounter	+0	+0
java.util.Hashtable\$KeySet	+0	+0
sun.reflect.DelegatingConstructorAccessorImpl	+0	+0
sun.nio.fs.WindowsNativeDispatcher\$DiskFreeSpace	+0	+0
java.lang.CharSequence	+0	+0
javax.management.openmbean.OpenMBeanAttributeInfo	+0	+0
com.sun.beans.WeakCache	+0	+0
java.io.Serializable	+0	+0
java.lang.invoke.InnerClassLambdaMetafactory\$1	+0	+0
java.lang.Character[]	+0	+0
java.util.Currency	+0	+0
java.lang.ClassCastException	+0	+0
java.util.function.Supplier	+0	+0
sun.util.logging.PlatformLogger\$Level[]	+0	+0
java.util.LinkedHashMap	+0	+0
java.nio.channels.InterruptibleChannel	+0	+0
sun.nio.cs.ext.ExtendedCharsets	+0	+0
Σ Total: 32 of 1,895 entries; 1,863 more	-792,833	-25,370,

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。

3.5.4 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

Overview default_report org.eclipse.mat.api:suspects OQL

```
SELECT s FROM java.lang.String s WHERE (s.toString().length() > 100)
```

s
java.lang.String [id=0x6c0796198]
java.lang.String [id=0x6c0722e78]
java.lang.String [id=0x6c0722788]
java.lang.String [id=0x6c0722520]
java.lang.String [id=0x6c07223d0]
java.lang.String [id=0x6c0721db8]
java.lang.String [id=0x6c0721bd0]
java.lang.String [id=0x6c06b00a0]
java.lang.String [id=0x6c06acd08]
java.lang.String [id=0x6c06876f8]
java.lang.String [id=0x6c0252940]
java.lang.String [id=0x6c0248098]
java.lang.String [id=0x6c0243820]
java.lang.String [id=0x6c0242c48]
java.lang.String [id=0x6c0242570]
java.lang.String [id=0x6c0241608]
java.lang.String [id=0x6c022ab60]

Σ Total: 24 of 243 entries; 219 more

查询结果见上图。可以看到一共有 243 个 String 的对象长度大于 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，转用 MAT 进行搜索，使用的搜索条件如下：

```
SELECT *
FROM edge.NetworkConnection s
WHERE s.label.toString().contains("S")
```

查询的结果如下：

i Overview default_report org.eclipse.mat.api:suspects OQL Histogram			
SELECT * FROM edge.NetworkConnection s WHERE s.label.toString().contains("S")			
Class Name	Shallow Heap	Retained Heap	
> edge.NetworkConnection @ 0x6c4546b28	32	32	
> edge.NetworkConnection @ 0x6c4546ae8	32	32	
> edge.NetworkConnection @ 0x6c4546aa8	32	32	
> edge.NetworkConnection @ 0x6c4546968	32	32	
> edge.NetworkConnection @ 0x6c4546928	32	32	
> edge.NetworkConnection @ 0x6c45468e8	32	32	
> edge.NetworkConnection @ 0x6c45468a8	32	32	
> edge.NetworkConnection @ 0x6c4546868	32	32	
> edge.NetworkConnection @ 0x6c4545a10	32	32	
> edge.NetworkConnection @ 0x6c4545810	32	32	
> edge.NetworkConnection @ 0x6c45457d0	32	32	
> edge.NetworkConnection @ 0x6c4545710	32	32	
> edge.NetworkConnection @ 0x6c45456d0	32	32	
> edge.NetworkConnection @ 0x6c4545690	32	32	
> edge.NetworkConnection @ 0x6c4545610	32	32	
> edge.NetworkConnection @ 0x6c45455d0	32	32	
> edge.NetworkConnection @ 0x6c45453d0	32	32	
> edge.NetworkConnection @ 0x6c4545350	32	32	
Σ Total: 24 of 198,216 entries; 198,192 more			

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

SELECT * FROM graph.NetworkTopology

Class Name	Shallow Heap <Numeric>	Retained Heap <Numeric>
<Regex>		
graph.NetworkTopology @ 0x6c000c540	24	30,566,112
<class> class graph.NetworkTopology	8	8
vertexMap java.util.HashMap @ 0x6c00...	48	112,448
name java.lang.String @ 0x6c0274c68	24	64
edgeMap java.util.HashMap @ 0x6c027...	48	29,565,120
Σ Total: 4 entries		

(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：

```
SELECT * FROM "vertex\..*"
```

查询的结果如下：

Overview default_report org.eclipse.mat.api:suspects OQL Histogram

SELECT * FROM "vertex\..*"

Class Name	Shallow Heap	Retained Heap
> vertex.Server @ 0x6c076bc28	32	296
> vertex.Server @ 0x6c076b918	32	296
> vertex.Server @ 0x6c076b608	32	296
> vertex.Server @ 0x6c076b480	32	296
> vertex.Server @ 0x6c076a180	32	296
> vertex.Server @ 0x6c0769a10	32	296
> vertex.Server @ 0x6c07671d0	32	296
> vertex.Server @ 0x6c0767048	32	296
> vertex.Server @ 0x6c0766d38	32	296
> vertex.Server @ 0x6c0766a28	32	296
> vertex.Server @ 0x6c0766718	32	296
> vertex.Server @ 0x6c0766408	32	296
> vertex.Server @ 0x6c07660f8	32	296
> vertex.Server @ 0x6c0765de8	32	296
> vertex.Server @ 0x6c0765ad8	32	296
> vertex.Server @ 0x6c07657c8	32	296
> vertex.Server @ 0x6c07654b8	32	296
> vertex.Server @ 0x6c0764e98	32	296
Σ Total: 24 of 3,003 entries; 2,979 more		

由于此处操作中又另外引入 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。

```
SELECT * FROM java.util.HashMap$Node
```

Class Name	Shallow Heap	Retained Heap
> java.util.HashMap\$Node @ 0x6c4546cc8	32	64
> java.util.HashMap\$Node @ 0x6c4546c88	32	64
> java.util.HashMap\$Node @ 0x6c4546c48	32	128
> java.util.HashMap\$Node @ 0x6c4546c08	32	192
> java.util.HashMap\$Node @ 0x6c4546bc8	32	64
> java.util.HashMap\$Node @ 0x6c4546b88	32	64
> java.util.HashMap\$Node @ 0x6c4546b48	32	64
> java.util.HashMap\$Node @ 0x6c4546b08	32	64
> java.util.HashMap\$Node @ 0x6c4546ac8	32	64
> java.util.HashMap\$Node @ 0x6c4546a88	32	64
> java.util.HashMap\$Node @ 0x6c4546a48	32	64
> java.util.HashMap\$Node @ 0x6c4546a08	32	64
> java.util.HashMap\$Node @ 0x6c45469c8	32	128
> java.util.HashMap\$Node @ 0x6c4546988	32	64
> java.util.HashMap\$Node @ 0x6c4546948	32	128
> java.util.HashMap\$Node @ 0x6c4546908	32	64
> java.util.HashMap\$Node @ 0x6c45468c8	32	64
> java.util.HashMap\$Node @ 0x6c4546888	32	128
Σ Total: 24 of 800,612 entries; 800,588 more		

答案是超过了 80 万个。

3.5.5 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：

```
"main" #1 prio=5 os_prio=0 tid=0x000000003134800 nid=0x169c waiting on
condition [0x00000000308f000]
  java.lang.Thread.State: TIMED_WAITING (sleeping)
    at java.lang.Thread.sleep(Native Method)
    at graph.ConcreteGraph.removeVertex(ConcreteGraph.java:76)
    at graph.ConcreteGraph.removeVertex(ConcreteGraph.java:25)
    at Main.main(Main.java:60)
```

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：

```
"main" #1 prio=5 os_prio=0 tid=0x000000002cf4800 nid=0x31c waiting on
condition [0x0000000028af000]
  java.lang.Thread.State: TIMED_WAITING (sleeping)
    at java.lang.Thread.sleep(Native Method)
    at graph.ConcreteGraph.addEdge(ConcreteGraph.java:153)
    at graph.NetworkTopology.addEdge(NetworkTopology.java:50)
    at graph.NetworkTopology.addEdge(NetworkTopology.java:17)
    at factory.GraphFactory.createGraph(GraphFactory.java:187)
    at factory.GraphFactory.createGraphStrategy(GraphFactory.java:65)
    at Main.main(Main.java:38)
```

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。

4 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

5 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。

6 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。

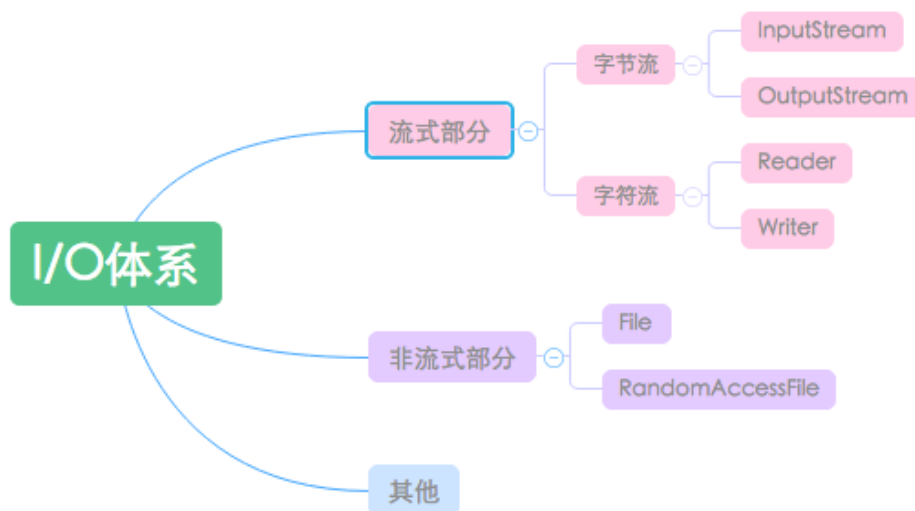
(1) “代码看起来很美”和“代码运行起来很美”没有必然的联系。“代码运行起来很美”，关注的是代码的运行效率，即我们如何将代码的算法效率、IO 效率尽量提高；而“代码看起来很美”关注的是我代码的可理解性，因为在当前这个世界里任何一个程序员都不可能独立工作。足够好的代码可理解性可以更好的帮助程序员之间的交流，便于更好的提高。

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以下几种：



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何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。