

**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 | Arcwoc |
| 学号 | xxxxx |
| 班号 | xxxxxx |
| 电子邮件 | xxxxx |
| 手机号码 | xxxxx |

**目录**

[1 实验目标概述 1](#_Toc513573923)

[2 实验环境配置 1](#_Toc513573924)

[3 实验过程 1](#_Toc513573925)

[3.1 Static Program Analysis 1](#_Toc513573926)

[3.1.1 人工代码走查（walkthrough） 1](#_Toc513573927)

[3.1.2 使用CheckStyle和FindBugs进行静态代码分析 1](#_Toc513573928)

[3.2 Java I/O 1](#_Toc513573929)

[3.2.1 多种I/O实现方式 1](#_Toc513573930)

[3.2.2 多种I/O实现方式的效率对比分析 2](#_Toc513573931)

[3.3 Java Memory Management and Garbage Collection (GC) 3](#_Toc513573932)

[3.3.1 使用-verbose:gc参数 3](#_Toc513573933)

[3.3.2 用jstat命令行工具的-gc和-gcutil参数 3](#_Toc513573934)

[3.3.3 使用jmap -heap命令行工具 3](#_Toc513573935)

[3.3.4 使用jmap -histo命令行工具（可选） 3](#_Toc513573936)

[3.3.5 使用jmap -permstat命令行工具（可选） 3](#_Toc513573937)

[3.3.6 使用jconsole或VisualVM工具 3](#_Toc513573938)

[3.3.7 分析垃圾回收过程是否正常、异常 3](#_Toc513573939)

[3.3.8 配置JVM参数并发现最优参数配置 3](#_Toc513573940)

[3.4 Dynamic Program Profiling 3](#_Toc513573941)

[3.4.1 使用Visual VM进行CPU Profiling 3](#_Toc513573942)

[3.4.2 使用Visual VM进行Memory profiling 3](#_Toc513573943)

[3.5 Memory Dump Analysis and Performance Optimization 3](#_Toc513573944)

[3.5.1 内存导出(memory dump) 3](#_Toc513573945)

[3.5.2 使用MAT分析内存导出文件 3](#_Toc513573946)

[3.5.3 发现热点/瓶颈并改进、改进前后的性能对比分析 3](#_Toc513573947)

[3.5.4 jhat和OQL查询内存导出（可选） 4](#_Toc513573948)

[3.5.5 jstack导出java程序运行时的调用栈（可选） 4](#_Toc513573949)

[4 实验进度记录 4](#_Toc513573950)

[5 实验过程中遇到的困难与解决途径 4](#_Toc513573951)

[6 实验过程中收获的经验、教训、感想 4](#_Toc513573952)

# 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FindBugs）,动态代码分析（Java命令行工具jstat、jmap、jConsole、VisualVM）,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

1. Checkstyle安装

Help -> install new software -> add -> 输入网址 -> install

1. MAT安装

Help -> Eclipse MatketSpace 查询安装。

1. VisualVM

下载jdk时自带。

https://github.com/ComputerScienceHIT/Lab5-1160301012

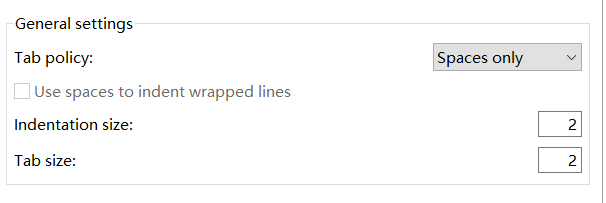
# 实验过程

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！）。

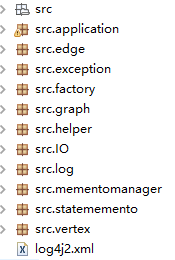
## Static Program Analysis

### 人工代码走查（walkthrough）

1. 不允许使用tab，并且tab以两个空格的大小为单位。



1. 每个方法前都需要写JavaDoc
2. 包的名字首字母要小写。



1. 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。
2. 消除所有的warning，比如没有被使用到的变量造成的warning.
3. 类内的属性是否都已经被初始化
4. 检查代码是否尽量使用if-else，不要一直if去判断，影响效率
5. 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

**提交版本号： 3c5a5f**

### 使用CheckStyle和FindBugs进行静态代码分析

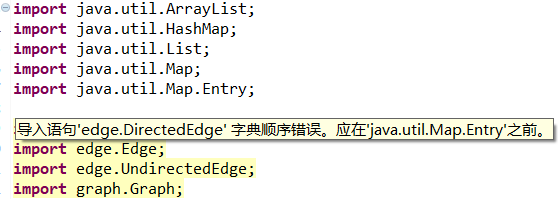
**CheckStyle：**

先下载了google的java代码规范的xml文件导入eclipse，重新保存后，自动重置了代码的格式(包括换行，缩进，等等)

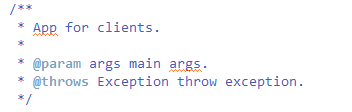


1.If分支后一定要有大括号，无论后面是否只有一条语句。

2.自己定义的类的import，要放在其他import之前。



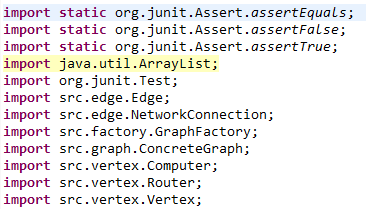
3.要为每个方法写JavaDoc，并且JavaDoc中注释结束要有”.”



4.不能通过.\*的方式引用包，要修改为包中具体的内容

5.变量声明与使用最多不能间隔超过三行。如果确定变量不变，将变量定义为final类型。

6.导入组之间应该有空行



**FindBugs:**

没有发现报错。

**提交版本号： beaac0**

## Java I/O

### 多种I/O实现方式

工程中采用了Stream，Writer/Reader，NIO来读写文件。

**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。

**提交版本号：159d7b**

**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。

**NIO读：**

根据文件路径创建一个Path对象，并且调用java.nio.file.Files中的Files类的readAllLines方法来按行读取文件，并且将每一行都存储在一个list里面，然后根据相应的图的类型，调用其parse函数，解析list里面的每一个字符串元素。将解析的信息加入图中，直到最后一个元素被解析，整个过程结束。

**NIO写：**

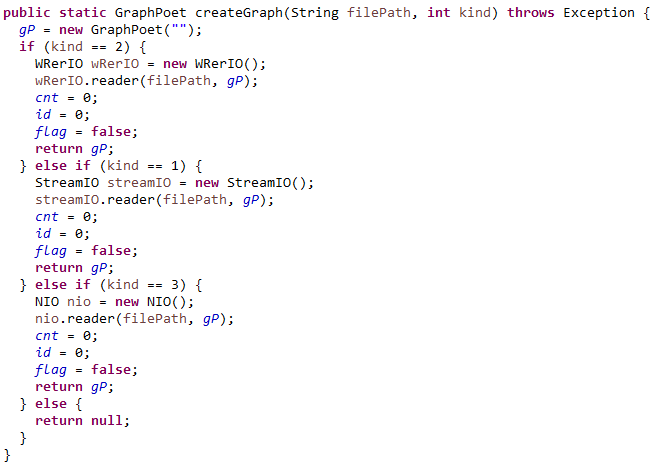
创建一个FIle对象，传递给FileOutputStream类的构造方法创建一个对象，再调用该对象的getChannel方法，返回一个FileChannel类型的对象，遍历图中的所有结点和边，用channel.write()方法，按行将信息写入文件，并且清理缓冲区。

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reader()和writer()两个方法：

IMG_256

并且在NIO.java，StreamIO.java，WRerIO.java中用不同的读写策略实现了该接口的读写方法。先根据方法的传入的参数的图的类型判断要调用哪个类的parse方法来解析字符串，然后没读入一行文件，就把读入的字符串传递给相应的类的parse方法来解析字符串获取信息。Parse方法位于Factory方法中，是静态函数。并且Factory类中的createGraph方法可以根据用户传入的kind参数确定用哪种读写策略来读写文件，返回一个完整的图。以GraphPoetFactory为例，createGraph方法实现如下：



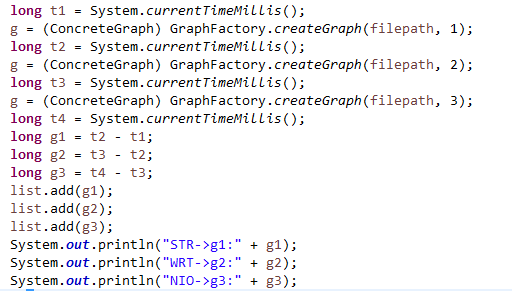
**提交版本号： 7a1aac**

### 多种I/O实现方式的效率对比分析

**收集IO语法文件的时间**

收集从文件读图的时间：

因为在该工程中是每读一行，就把这一行字符串解析，并把相应信息加入图中，所以收集从文件读图的时间最终根据老师的建议收集的是读入文件和建图一起的时间。在程序中代码如下：

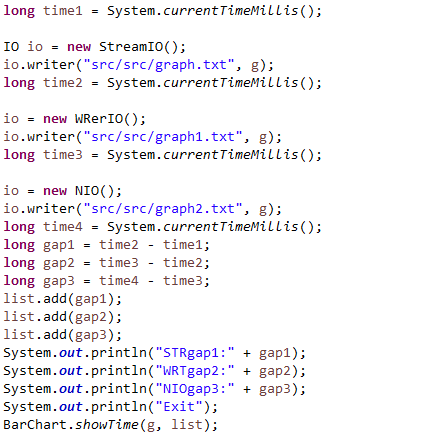


GraphFactory.createGraph(filepath,kind)的第二个参数是用来选择读取文件的方式的。kind=1时，采用stream来读取文件；kind=2时，采用writer/reader来读取文件；kind=3时，采用NIO来读取文件。

下面的g1是计算的是stream读取文件并建图的过程的时间消耗，g2计算的是reader读取文件并建图的过程的时间消耗，g3计算的是nio读取文件并建图的过程的时间消耗。最后将时间打印在控制台并记录起来，最后绘制成图。

将图写入文件消耗的时间：

将图的信息写入文件消耗的时间也包括两部分，第一部分是遍历图并生成string类型的信息，第二部分写入文件，由于两部分也是同时交替进行，并且为了与前面读文件相匹配，所以一起测量用时。在程序中代码如下：



很明显，StreamIO是采用stream策略写文件，WRerIO是采用writer策略写文件，NIO是采用NIO策略写文件。gap1记录的是stream遍历图并写入文件的时间，gap2记录的是writer遍历图并写入文件的时间，gap3记录的是NIO遍历图并写入文件的时间。最后将时间打印在控制台，并且绘制成图。

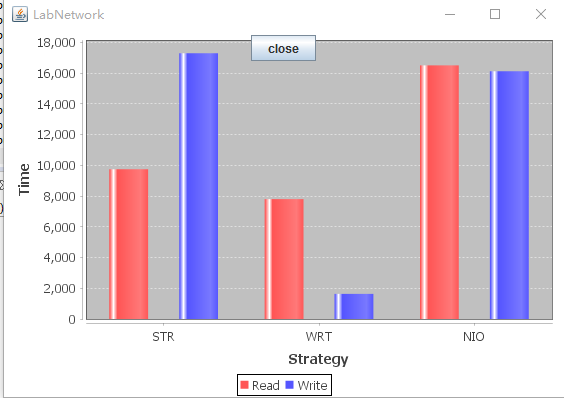
注意：为了使得最后写入文件的信息是完整的(即由文件和用户命令行共同作用后的图)，所以**写入图在用户输入命令exit之后发生**。为了使得写入操作的时间和读入操作的时间在柱状图中形成对比，所以在写入文件之后，程序结束之前绘制柱状图。(调用的是BarChart.showTime(g,list)，上图代码最后一行可见)。

**表格对比不同IO的性能**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **时间(ms)** | **读/写** | **File1.txt** | **File2.txt** | **File3.txt** | **File4.txt** |
| **I/O策略1**  **（stream）** | **读文件** | 9752 | 24583 | 21694 | 17758 |
| **写文件** | 17300 | 17324 | 11814 | 22966 |
| **I/O策略2**  **(writer/reader）** | **读文件** | 7804 | 24814 | 22416 | 19908 |
| **写文件** | 1635 | 2455 | 3106 | 4008 |
| **I/O策略3**  **(NIO)** | **读文件** | 16516 | 21447 | 15392 | 23500 |
| **写文件** | 16128 | 16037 | 19764 | 21464 |

**图形对比不同I/O的性能（可选）**

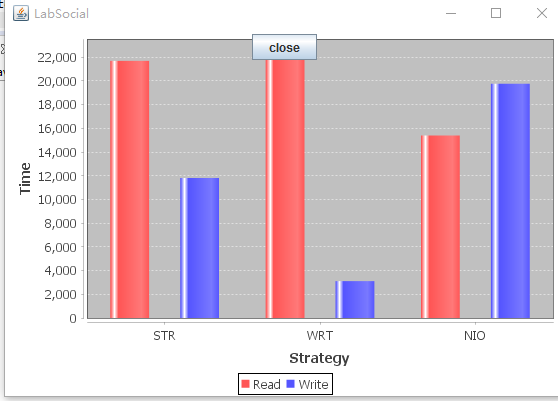
file1.txt(NetworkTopology):



file2.txt(MovieGraph):



file3.txt(SocialNetwork):



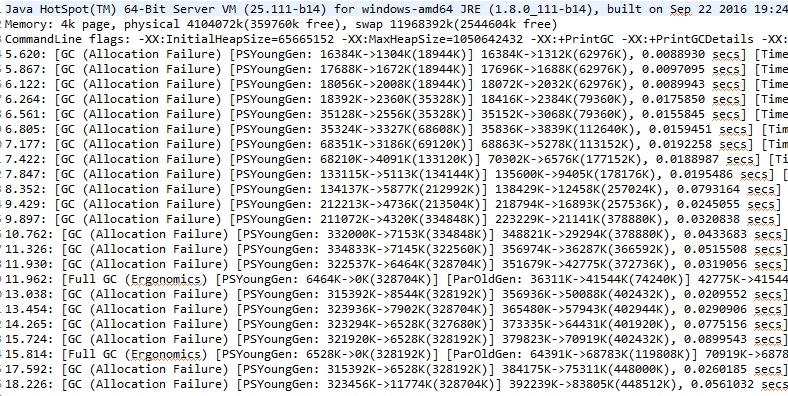
file4.txt(WordGraph):



## Java Memory Management and Garbage Collection (GC)

### 使用-verbose:gc参数

日志记录：



由日志内容可知：

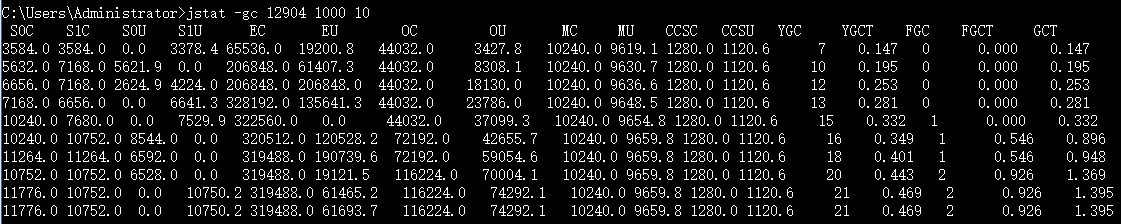
1. 12.66秒内，Minor GC一共发生了21次，Full GC发生了2次。
2. 21次Minor GC一共耗费了0.7秒，平均每次0.033秒
3. 2次Full GC一共耗费了1.49秒，平均每次0.744秒
4. 每次GC前后heap中各区域占用情况变化：

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)一共进行了21次Minor GC和两次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；

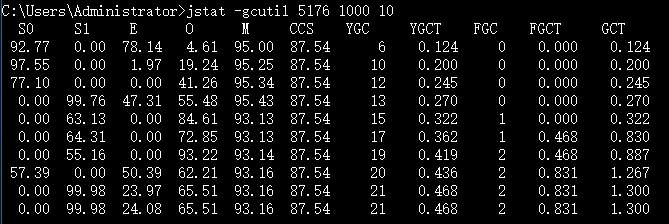
EC：Eden区的大小；EU：Eden区的使用大小

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

②可以看到在读取文件过程中，S0U和S1U在不断通过交换空间来满足内存需求。可以看到老年代回收执行了两次，程序运行过程中Full GC发生了两次。年轻代垃圾回收消耗的时间要短于老年代垃圾回收消耗的时间



**分析：**

①图中参数如下：

S0、S1 代表两个Survivor区；E 代表 Eden 区；O（Old）代表老年代；

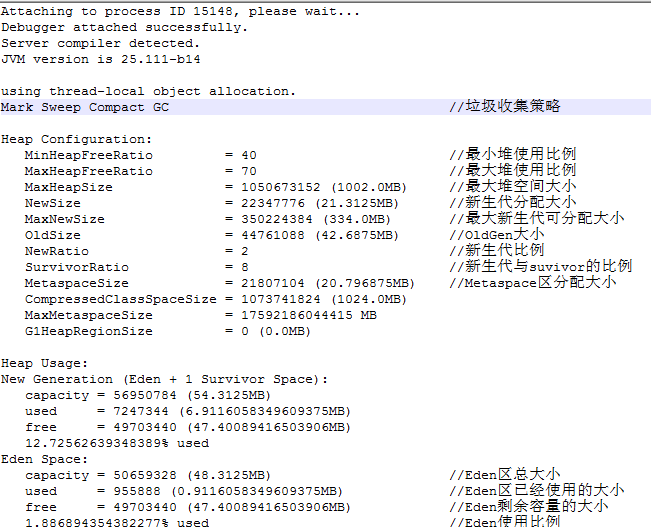
YGC（Young GC）代表Minor GC；YGCT代表Minor GC耗时；

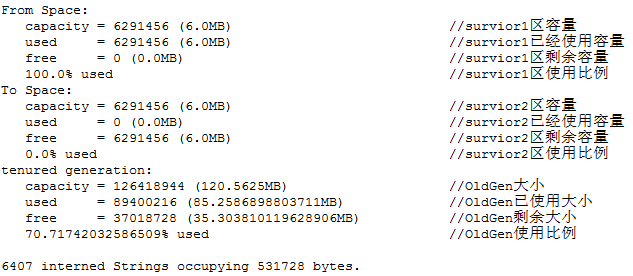
FGC（Full GC）代表Full GC耗时；GCT代表Minor & Full GC共计耗时。

②先向eden区申请分配空间，如果空间够，就直接进行分配，否则进行一次Minor GC。minor GC首先会对Eden区的对象进行标记，标记出来存活的对象。然后把存活的对象copy到from空间。如果from空间足够，则回收eden区可回收的对象。如果from内存空间不够，则把from空间存活的对象复制到to区，如果to区的内存空间也不够的话，则把to区存活的对象复制到老年代。如果老年代空间也不够（或者达到触发老年年垃圾回收条件的话）则触发一次Full GC。

### 使用jmap -heap命令行工具

具体参数含义标注在图中：

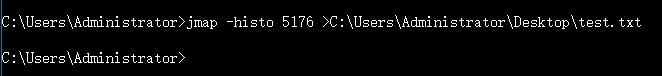


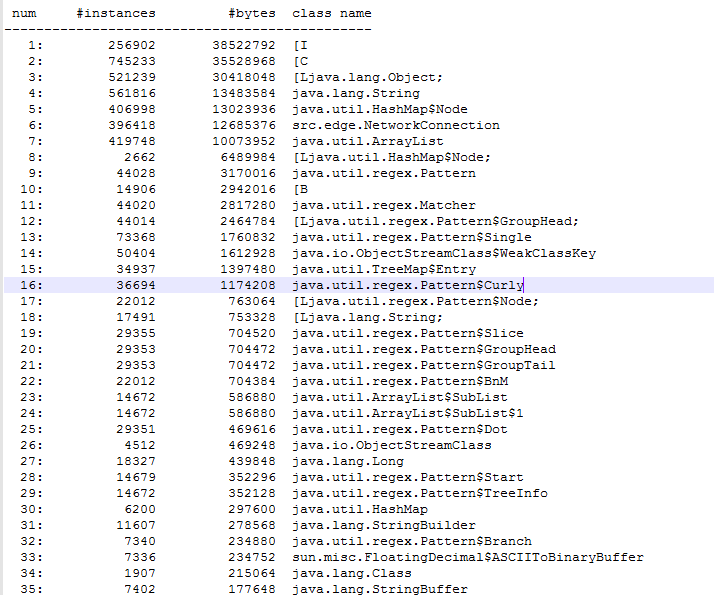


### 使用jmap -histo命令行工具（可选）

指令**jmap -histo Pid:**打印每个class的实例数目，内存占用，类全名信息，其中VM的内部类名字开头会加上前缀”\*”，如果live自参数加上后，只统计活的对象数量，则指令为：

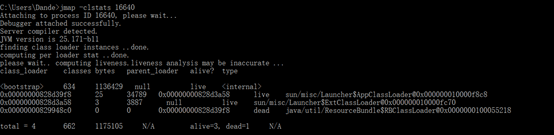
**jamp -histo:live Pid**





### 使用jmap -permstat命令行工具（可选）

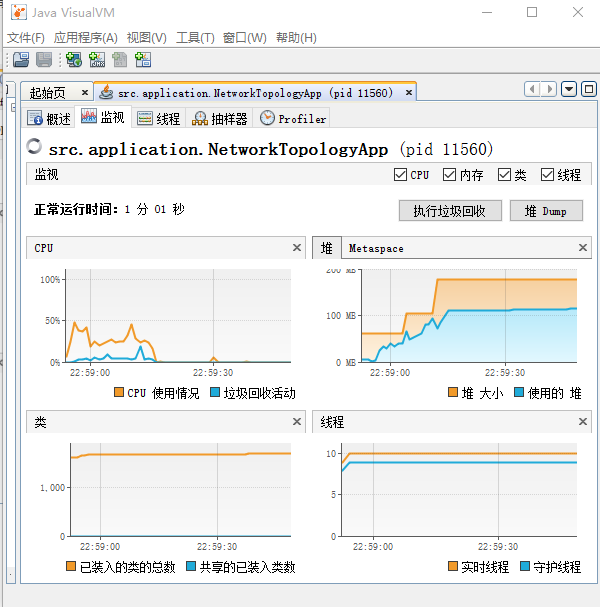
Java1.8之后完全除去了PermGen，所以没有 jmap -permstat Pid指令，取而代之的可以采用jamp -clstats Pid命令。



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。

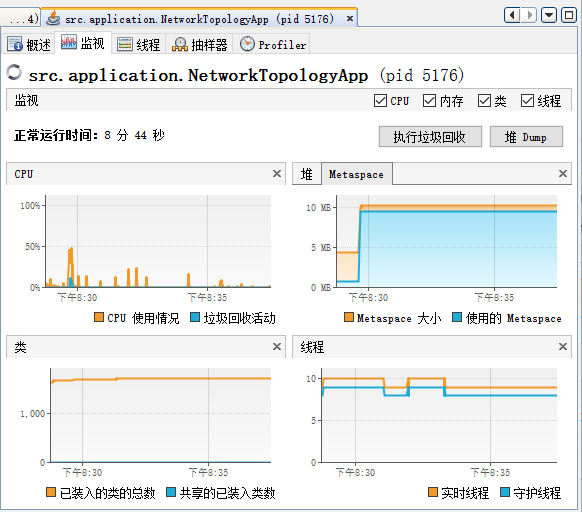
这里显示的是一次读取文件时装载的class和method相关信息。

### 使用jconsole或VisualVM工具



图一是CPU的使用情况以及垃圾回收占用CPU的情况。图二是对的使用情况

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。图三是装载的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。图四是实时线程和守护线程的变化情况。



图二展示的是Metaspace的大小和占用情况，可以发现Metaspace的大小基本在10MB稳定，使用的Metaspace大小也很稳定。

### 分析垃圾回收过程是否正常、异常

从3.3.6中的图一可以看到，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参数并发现最优参数配置

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

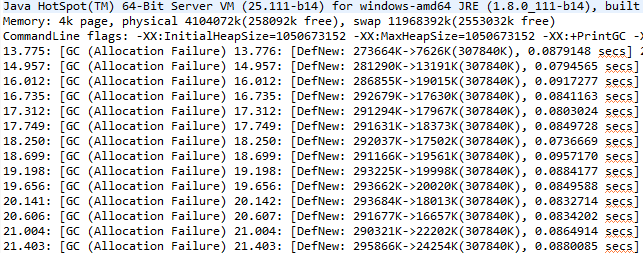
**测试配置①**：-Xmx32m -Xms4m -Xmn1m -XX:PermSize=5m -XX:MaxPermSize=10m -XX:+UseSerialGC

测试结果：发生了异常

**测试配置②**：-Xmx1002m -Xms1002m -Xmn320m -XX:+UseSerialGC

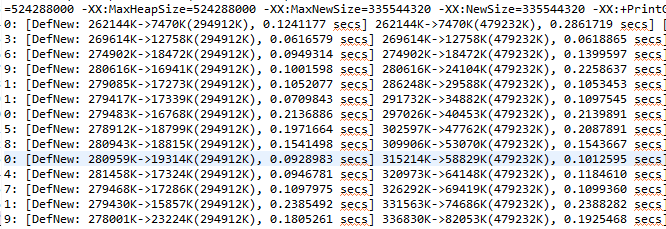
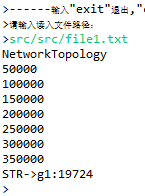
测试结果：

从日志来看，没有了Full GC，Minor GC次数也变少了，只发生了14次。用时较少，大概9000ms.



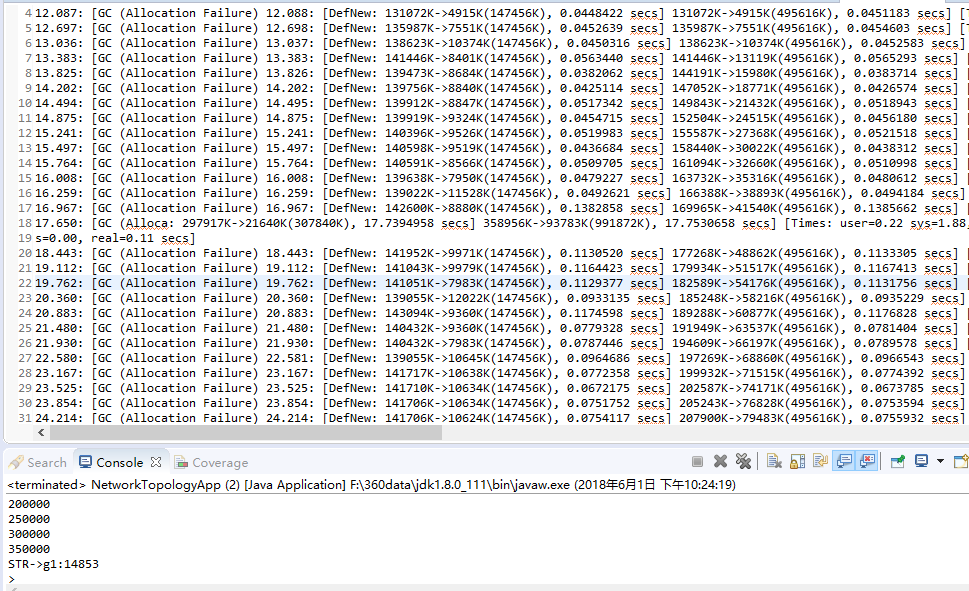
**测试配置③：**-Xmx500m -Xms500m -Xmn320m -XX:+UseSerialGC

测试结果：GC次数与测试②大致相同，但是每次GC消耗的时间变多，并且总用时比测试②慢了一倍。



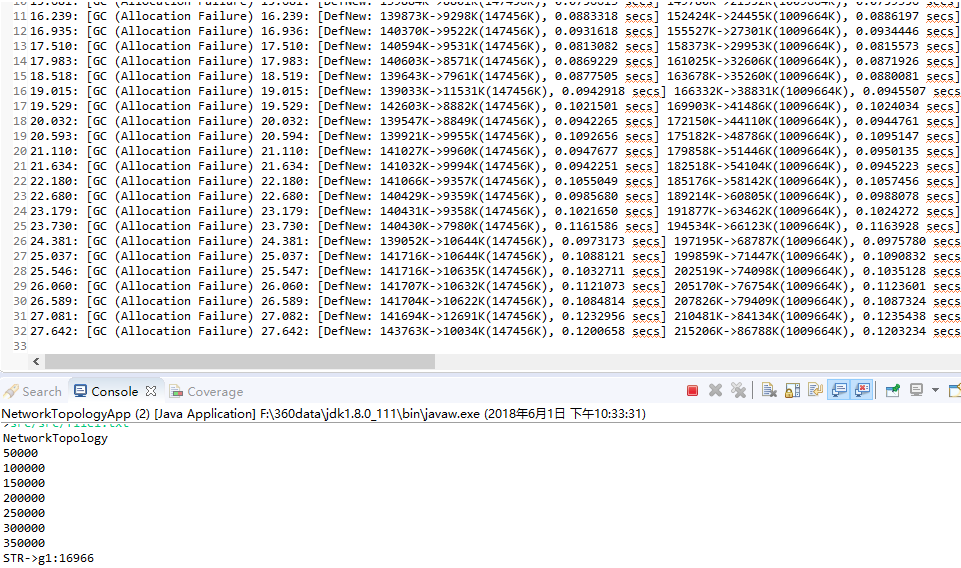
**测试配置④：**-Xmx500m -Xms500m -Xmn160m -XX:+UseSerialGC

测试结果：GC次数比前两次增加了一倍，进行了大约30次，每次消耗的时间比测试③变短，时间总消耗优于测试③，慢于测试②。



**测试配置⑤：**-Xmx1002m -Xms1002m -Xmn160m -XX:+UseSerialGC

测试结果：用时与④差不多，GC次数也是大约30次，每次消耗时间与④相差不多。总用时也与④差不多。

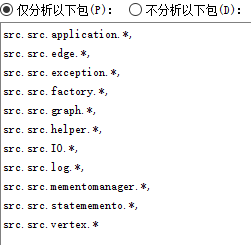


综上，最优的策略应该是：-Xmx1002m -Xms1002m -Xmn320m -XX:+UseSerialGC

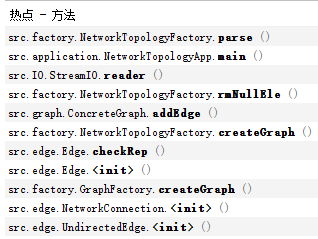
## Dynamic Program Profiling

### 使用Visual VM进行CPU Profiling

设置：

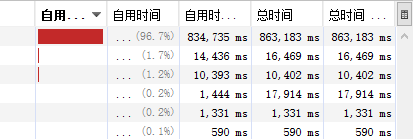
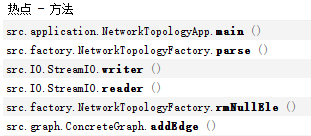


读操作：



从图中可知src.factory.NetworkTopologyFactory.parse()方法占据的时间最多，占据了57.7%.其次是src.application.NetworkTopologyApp.main()方法，占据了27.9%，剩下依次是StreamIO.reader()方法，rmNullEle()方法，和addEdge()方法。从图中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parse()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：



从图中可知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writer()方法也占用了较多时间，合理，因为要向文件中写入信息。

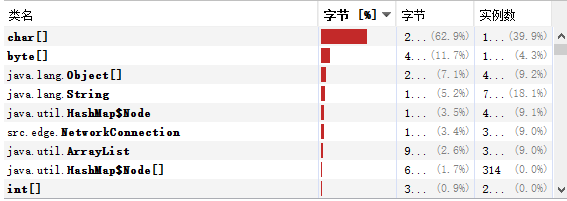
### 使用Visual VM进行Memory profiling

**读操作**：



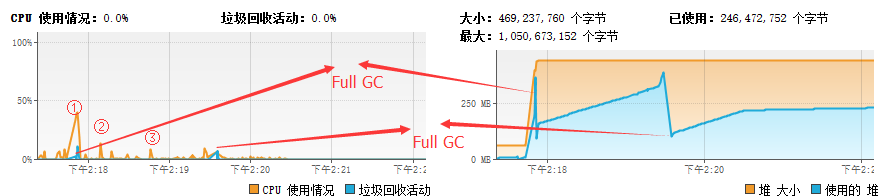
从图中可以知道int[]最多，有302044个，占据了4448万字节。其次是char[]和Object[]，分别有826456个和623215个，占据4239万字节和3378万字节。这些耗费内存是合理的。

**写操作：**



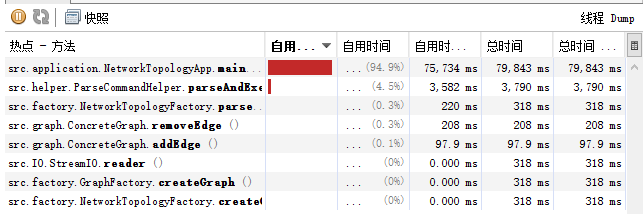
从图中可以看到char[]最多，其次是byte[]和Object[]，合理，因为在写操作时，需要生成很多字符串，并且写入文件。

**删除顶点和边**



可以看到图中2:18是读图的时候CPU的占用情况，②是删除顶点时的占用情况，③是删除边时的占用情况。

IMG_256

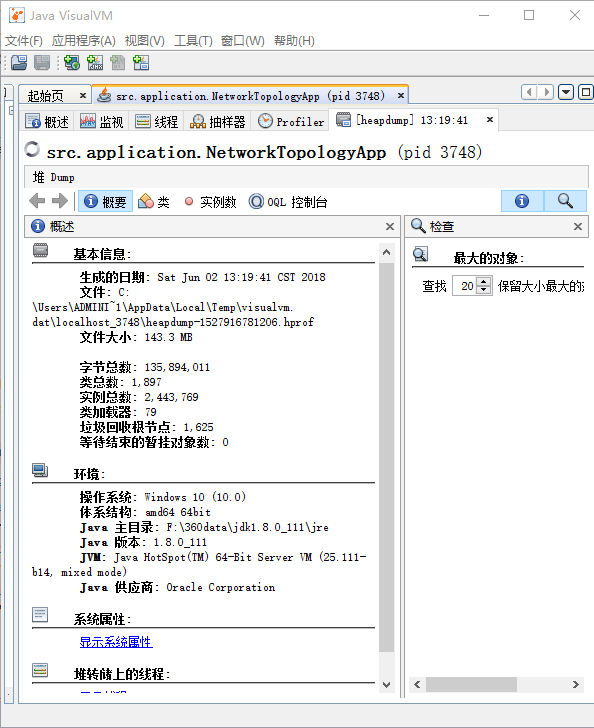


可以看到解析指令方法占用的时间以及删除顶点和边占用的时间。

## Memory Dump Analysis and Performance Optimization

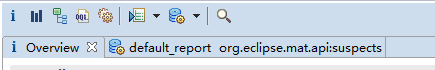
### 内存导出(memory dump)

在读取完file1.txt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

在Eclipse中安装MAT，然后打开生成的.hprof文件，选择Leak Suspects Report选项，这时候MAT就会生成报告，这个报告分为了两个标签页，一个是Overview，另一个是Leak Suspects（可能的内存泄漏）。



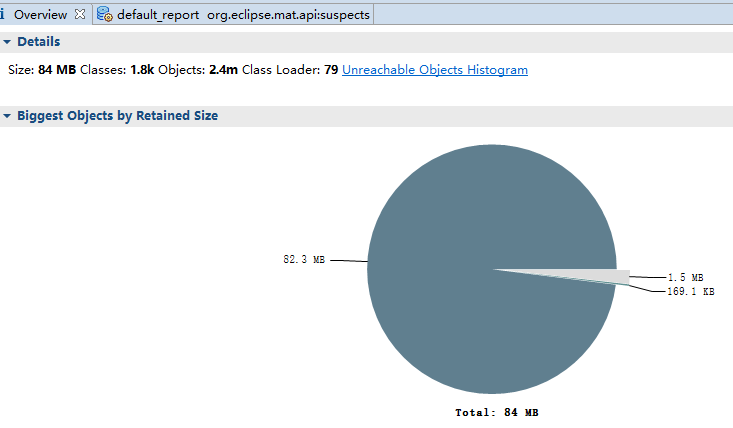
**在开始之前，先解释两个名词：**

**Shallow Size**   
对象自身占用的内存大小，不包括它引用的对象。

针对非数组类型的对象，它的大小就是对象与它所有的成员变量大小的总和。当然这里面还会包括一些java语言特性的数据存储单元。针对数组类型的对象，它的大小是数组元素对象的大小总和。  
**Retained Size**   
Retained Size=当前对象大小+当前对象可直接或间接引用到的对象的大小总和。(间接引用的含义：A->B->C, C就是间接引用) 换句话说，Retained Size就是当前对象被GC后，从Heap上总共能释放掉的内存。当然，释放的时候还要排除被GC Roots直接或间接引用的对象。他们暂时不会被被当做Garbage。

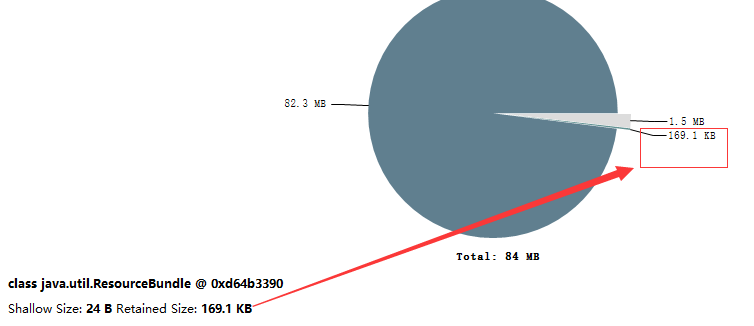
①Overview and Suspects

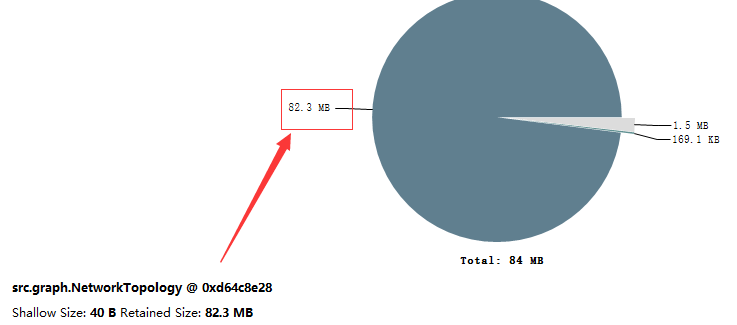
Size: 84 MB Classes: 1.8k Objects: 2.4m Class Loader: 79



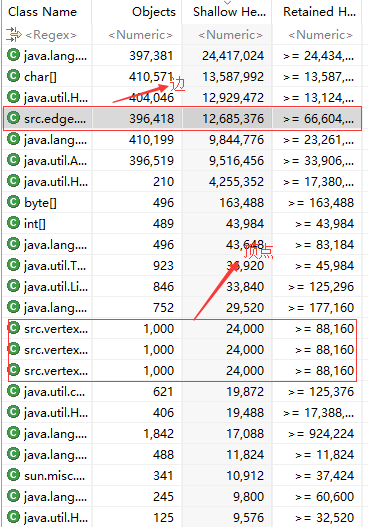
**分析：**

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。





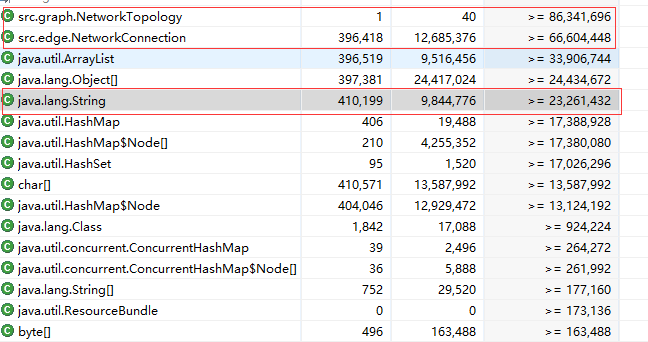
②Histogram: Lists number of instances per class(列出每个类的实例的数目来分析)



**分析：**

可以看到file1中边的个数和各类顶点的个数。通过上面的Regex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。

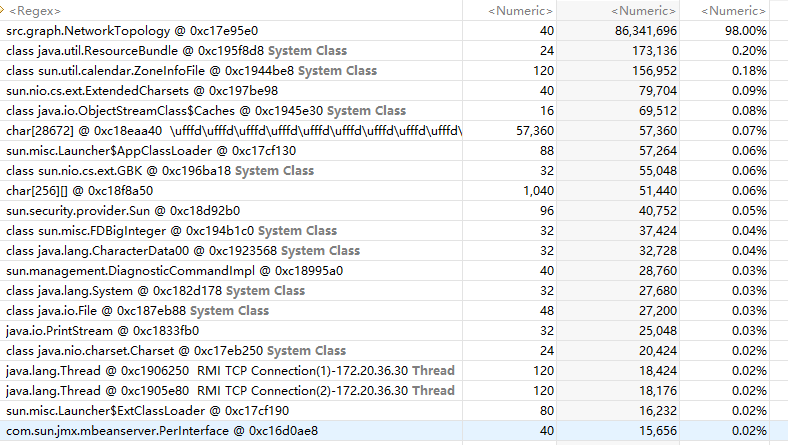
根据retained heap进行排序后可见：



IMG_256

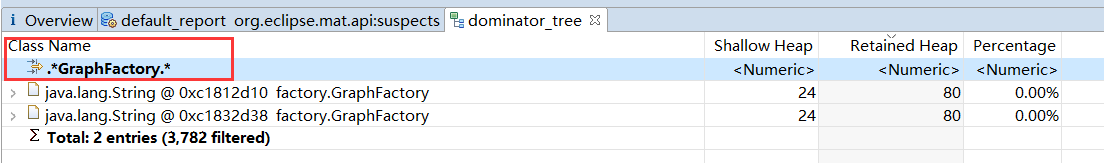
可以看到，红色方框内的对象的个数和占用的内存都是正常的，另外与之相关的比如String类，ArrayList类，Server，Router和Computer三个节点类的实例的数目和占用内存的大小都是合理的。因为NetworkTopology持有大量的对象的引用，所以图中显示不能说明NetworkTopology发生了内存泄漏。

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

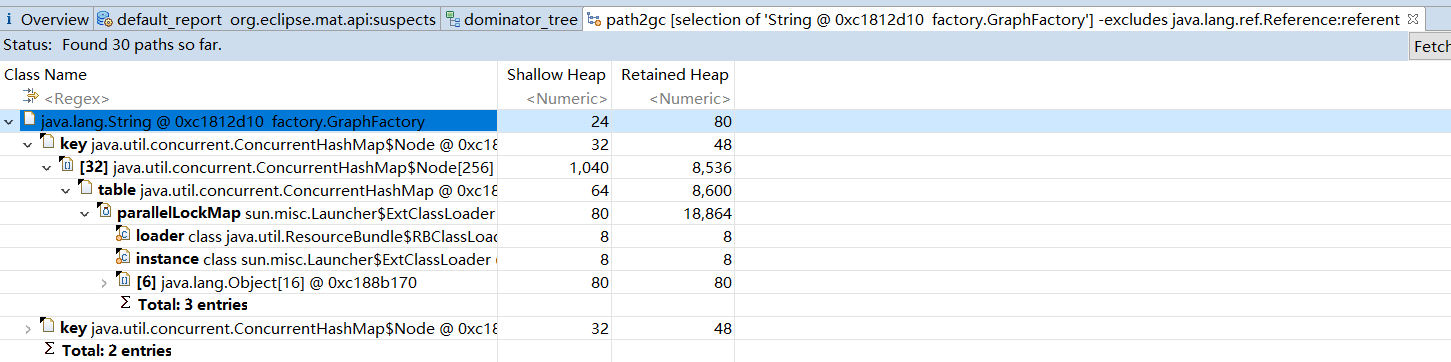


**分析：**

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。可以通过顶部的<Regex>来加入过滤条件：

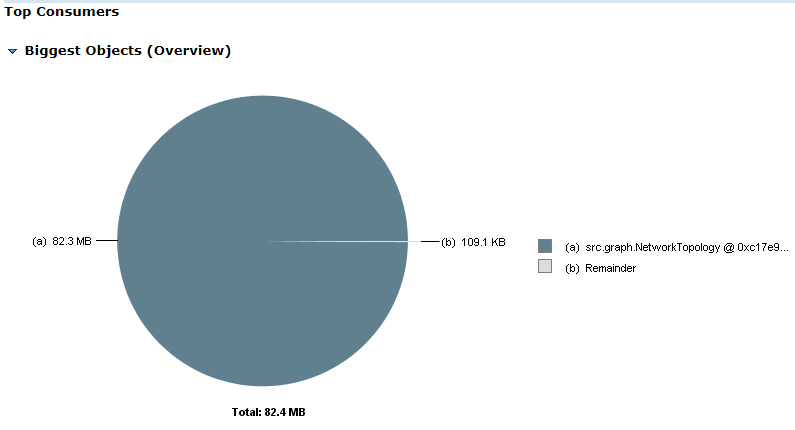


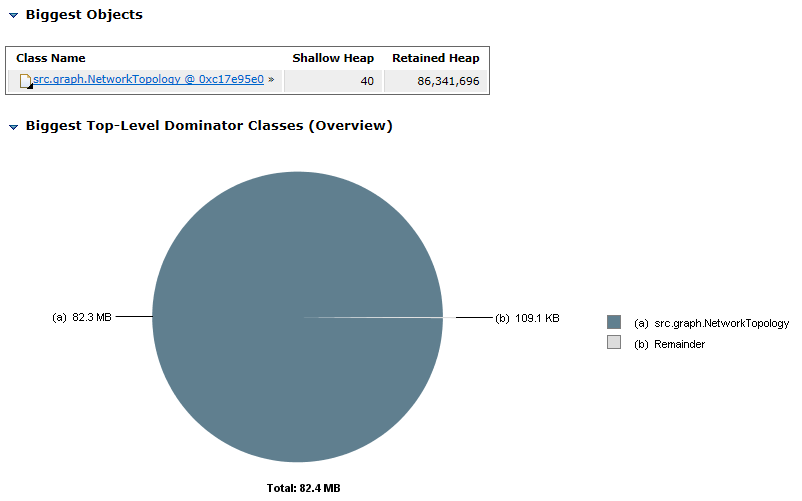
可以右键一行，选择Path to GC Roots-->exclude all，来查看GC链，

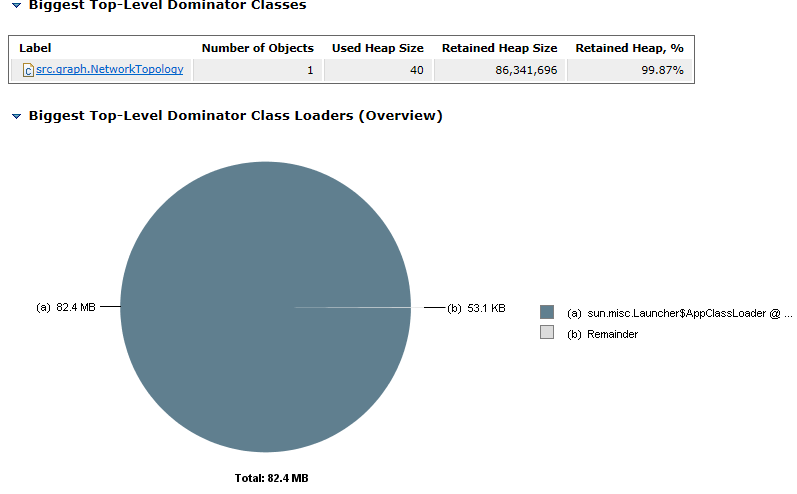


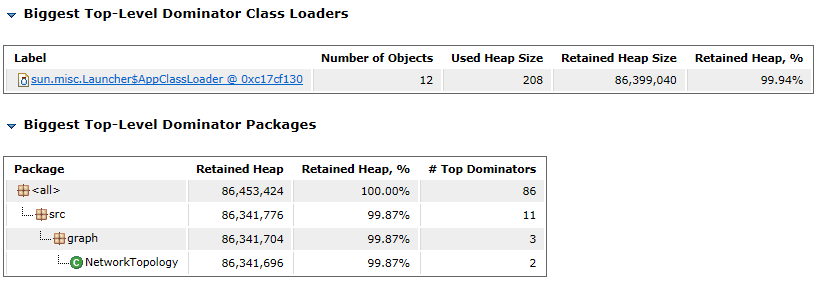
④Top Components: list reports for components bigger than 1 percent of the total heap.

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。

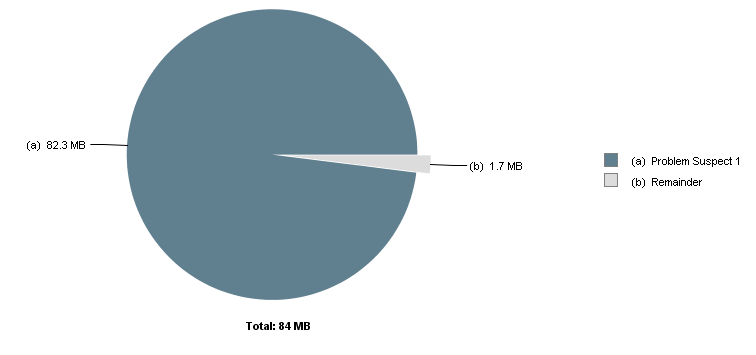




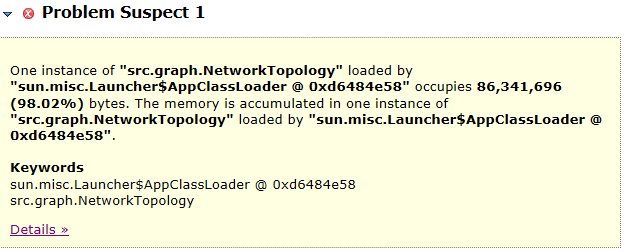


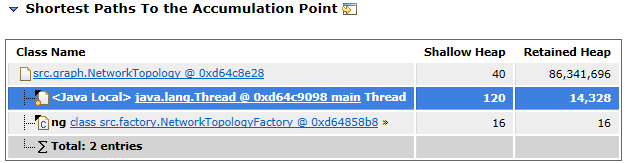


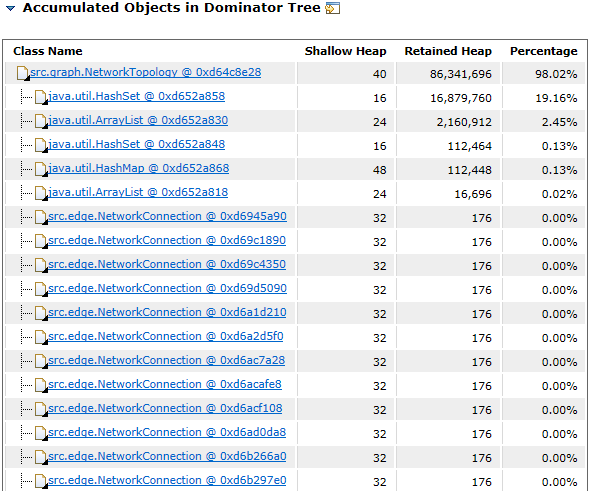
⑤Leak suspects report：

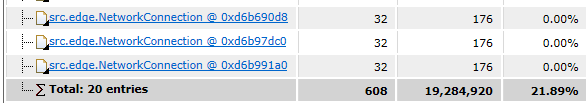


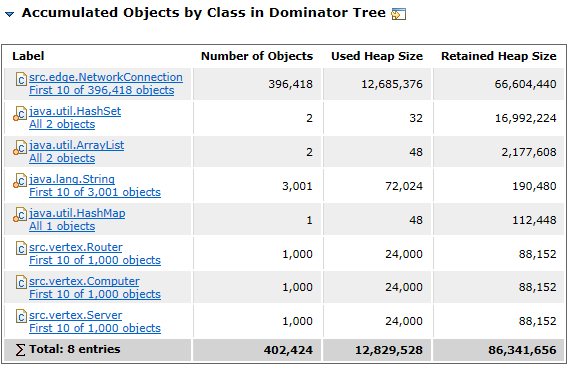
(a)区域表示可能发生内存泄漏，点击内存泄漏猜想的Details可以看到更深入的分析清理情况。

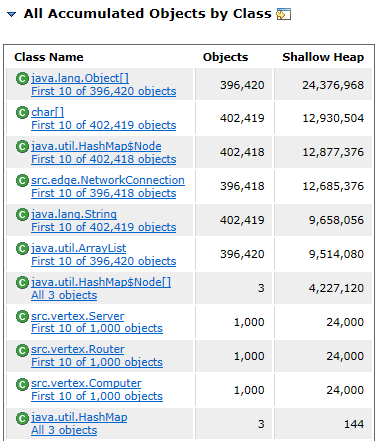














### 发现热点/瓶颈并改进、改进前后的性能对比分析

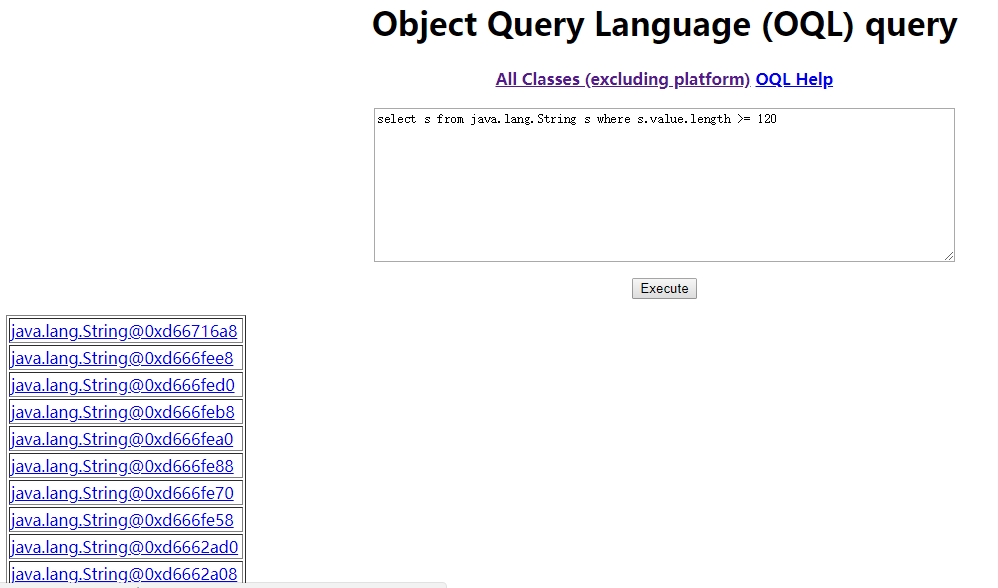
①改进前，读文档遇到边的端点的标签，要把相应的顶点加入边的list属性是通过遍历一遍所有的顶点，匹配相应的顶点，加入边的list中，这样太耗费时间，每次向边中加入端点耗费的时间都是O(n)的操作。改进策略：在图中加入一个map属性，map的key是顶点的label，map的value是顶点对象。在读文档遇到边的端点的label时，可以直接根据map中的映射利用label得到结点，花费时间是O(1)的，大大节省了时间。

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。

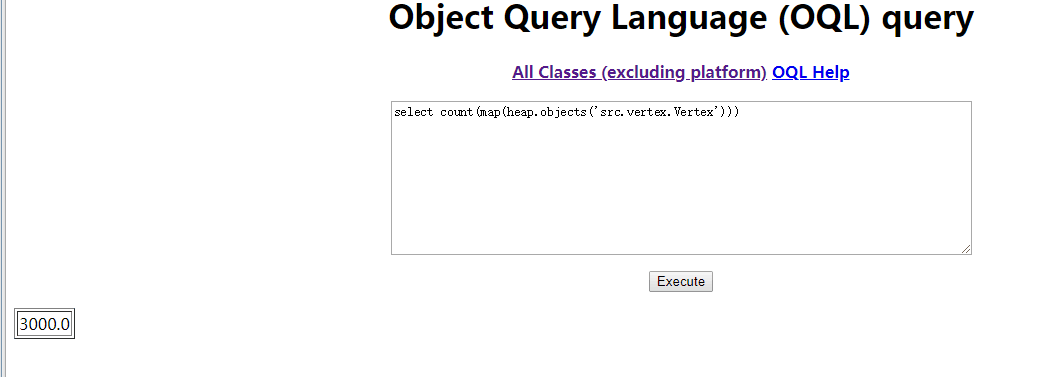
### jhat和OQL查询内存导出（可选）

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

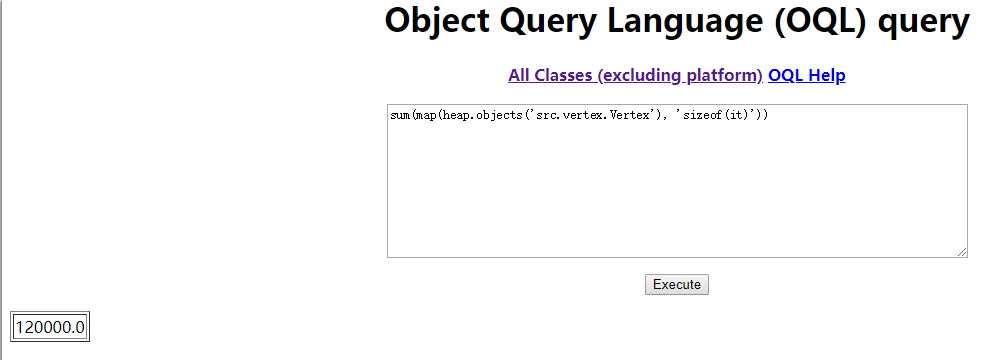


②Vertex（及其每个子类）的对象实例的数量和总占用内存大小

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

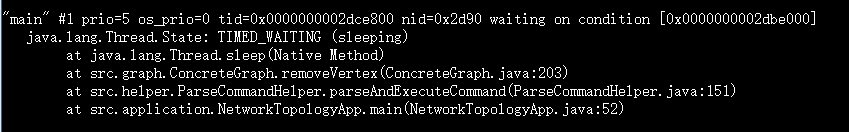


③对象实例的总大小: sum(map(heap.objects('src.vertex.Vertex'), 'sizeof(it)'))



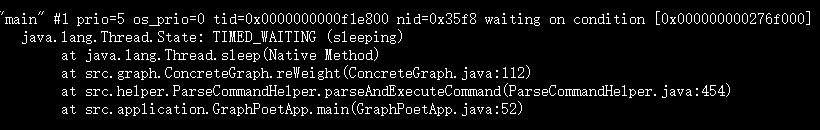
### jstack导出java程序运行时的调用栈（可选）

1. 删除某个节点



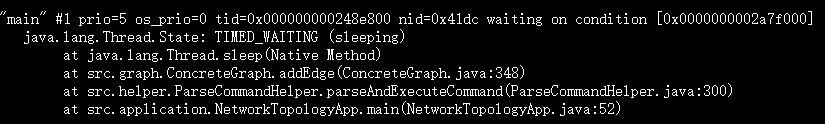
先调用parseAndExecuteCommand()方法解析命令并且执行命令，然后调用removeVertex()方法删除结点及其相应的边，最后是为了抓取此时的调用栈轨迹而特意调用的sleep()方法。

1. 修改某条边的权值（针对 GraphPoet 应用）



可以看到先调用了parseAndExecuteCommand()方法解析命令并且执行命令,然后调用图中的reWeight()方法来修改指定边的权重，最后是为了抓取此时调用栈轨迹调用的sleep()方法。

1. 增加一条边（针对 NetworkTopology 应用）



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

# 实验进度记录

请尽可能详细的记录你的进度情况。

|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 时间段 | 计划任务 | 实际完成情况 |
| 5-25~6-3 | 空闲时间 | 用各种工具进行堆内存及性能分析 | 完成 |
| 5-24 | 空闲时间 | 测读写文件时间 | 完成 |
| 5-23 | 空闲时间 | 读文件策略和写文件策略 | 完成 |

# 实验过程中遇到的困难与解决途径

在最开始读图建图的时候，速度非常慢，浪费了大量时间，后来用visualVM发现热点方法，找到消耗时间的原因，使得速度有了很大的提升。另外，安装，配置和使用各种工具，具体参数的含义不理解。通过查阅文档，博客解决。

# 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？

代码“看起来很美”和“运行起来很美”没有冲突。代码看起来美是结构的美观，运行起来美是效率高。往往一个好的结构也能让效率有一定的提高，让维护的成本降低。两者都应该在保证程序执行的正确性和健壮性得到保证的前提下，保证程序结构的清晰易懂，便于理解和维护，然后再去优化效率。在有限的变成时间里，我更倾向于把精力放在代码的看起来美上面。

1. 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？

我认为它们是一个规范，给程序员提醒，程序员可以参考，修改必须要改的，留下不需要改的。它们确实会帮我改善代码质量，改掉不良习惯带来的不好的代码风格。

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？

通过查阅很多资料，逐渐能够梳理清楚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。

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？

JVM的内存管理机制与计算机系统中学习的内存管理机制有很多相似的地方，JVM的内存分配过程比较有新意，分区申请内存，空间不够就进行垃圾清理。我觉得这一点很好，提高了效率。

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是否足够好？还有改进的空间吗？

C++手工进行垃圾回收比较灵活，用得恰当能够提高效率，但是容易出错。JVM自动进行垃圾回收，不容易出错，无需额外去特别关注这个问题。JVM的垃圾回收机制还是有改进的空间。

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？

我认为还是可以有足够的回报的，在这次实验中不同的配置让我感受到了一些效率上的差别，但是也没有特别明显。

1. 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？

这些工具可以帮我找到工程运行时耗时较多的地方，然后我可以对这些地方进行分析，分析耗时占比是否合理，如果不合理，哪里还可以继续优化，是优化程序性能非常好的工具。另外可以通过这些工具来调整自己java虚拟机的合适的参数，并发现代码中某些可能存在内存泄漏的地方，进行改进，提高程序性能。还可以发现程序运行过程中，哪些地方运行缓慢，哪些类占用了太多的空间，哪些类没有被GC掉，都可以做出改进。

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？

确实很多地方的效率降低就是仅仅因为一句代码，所以从细节注意，对于代码效率的提升是非常有必要的。

1. 关于本实验的工作量、难度、deadline。

这次实验相对于之前的实验，工作量降了非常非常多。也没有多少需要编写代码的地方，有很多工作都是在测试一些指标。

1. 到目前为止，你对《软件构造》课程的意见与建议。

这门课确实让我对软件构造的过程有一个大概的了解。从策划到动手，再到测试，改良。