



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	穆添愉
学号	1160301008
班号	1603010
电子邮件	1417553133@qq.com
手机号码	15636094072

目录

1 实验目标概述	1
2 实验环境配置	1
3 实验过程	4
3.1 Static Program Analysis	4
3.1.1 人工代码走查 (walkthrough)	4
3.1.2 使用 CheckStyle 和 FindBugs 进行静态代码分析	5
3.2 Java I/O	6
3.2.1 多种 I/O 实现方式	6
3.2.2 多种 I/O 实现方式的效率对比分析	10
3.3 Java Memory Management and Garbage Collection (GC)	12
3.3.1 使用-verbose:gc 参数	12
3.3.2 用 jstat 命令行工具的-gc 和-gcutil 参数	13
3.3.3 使用 jmap -heap 命令行工具	15
3.3.4 使用 jmap -histo 命令行工具 (可选)	15
3.3.5 使用 jmap -permstat 命令行工具 (可选)	16
3.3.6 使用 jconsole 或 VisualVM 工具	16
3.3.7 分析垃圾回收过程是否正常、异常	18
3.3.8 配置 JVM 参数并发现最优参数配置	18
3.4 Dynamic Program Profiling	19
3.4.1 使用 Visual VM 进行 CPU Profiling	19
3.4.2 使用 Visual VM 进行 Memory profiling	20
3.5 Memory Dump Analysis and Performance Optimization	21
3.5.1 内存导出(memory dump)	21
3.5.2 使用 MAT 分析内存导出文件	22
3.5.3 发现热点/瓶颈并改进、改进前后的性能对比分析	28
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	34

6 实验过程中收获的经验、教训、感想 35

1 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:

- (1) 静态代码分析(CheckStyle 和 FindBugs)
- (2) 动态代码分析(Java 命令行工具 jstat、jmap、jConsole、VisualVM)
- (3) JVM 内存管理与垃圾回收(GC)的优化配置
- (4) 运行时内存导出(memory dump)及其分析(Java 命令行工具 jhat、MAT)
- (5) 运行时调用栈及其分析(Java 命令行工具 jstack);
- (6) 高性能 I/O
- (7) 基于设计模式的代码调优
- (8) 代码重构

2 实验环境配置

仓库地址:

<https://github.com/ComputerScienceHIT/Lab5-1160301008>

配置过程:

- (1) Visual VM 的配置:



首先来到官网，点击“IDE Intergrations”，然后会跳转到如下页面：

The screenshot shows the "IDE Integrations" section of the VisualVM website. It starts with a heading "IDE Integrations" and a sub-section "NetBeans". It says: "Use the VisualVM features in your favorite Java IDE!" Below this is a "NetBeans" section with a sub-section "Eclipse". It says: "Use the **Eclipse Launcher** to integrate VisualVM with the Eclipse IDE. The plugin enables starting VisualVM along with the executed application and automatically opens the application tab." It provides installation instructions: "Installation: download the [plugin \(.zip, 38.9KB\)](#), unzip it and add as a local update site, then install the VisualVM Launcher Feature." Configuration instructions: "Configuration: setup the plugin by configuring path to JDK (not JRE) and VisualVM installation using Run/Debug-Launching-VisualVM Configuration." Usage instructions: "Usage: create a custom application configuration and choose the VisualVM Launcher as application launcher for the Run/Debug actions."

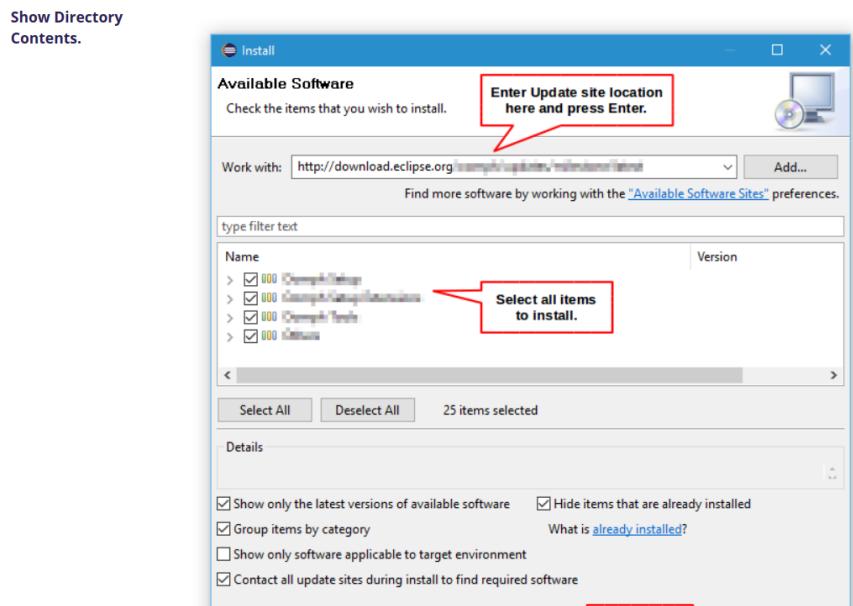
点击“Eclipse”版本的插件，进行下载即可，然后解压文件夹，在项目的Build Path处添加jar包到项目库中，重启IDE，就已经配置好了。

(2) Memory Analyzer (MAT)的配置

官网已经给出了该如何在线安装，只需要照着指引一步步去做就好。

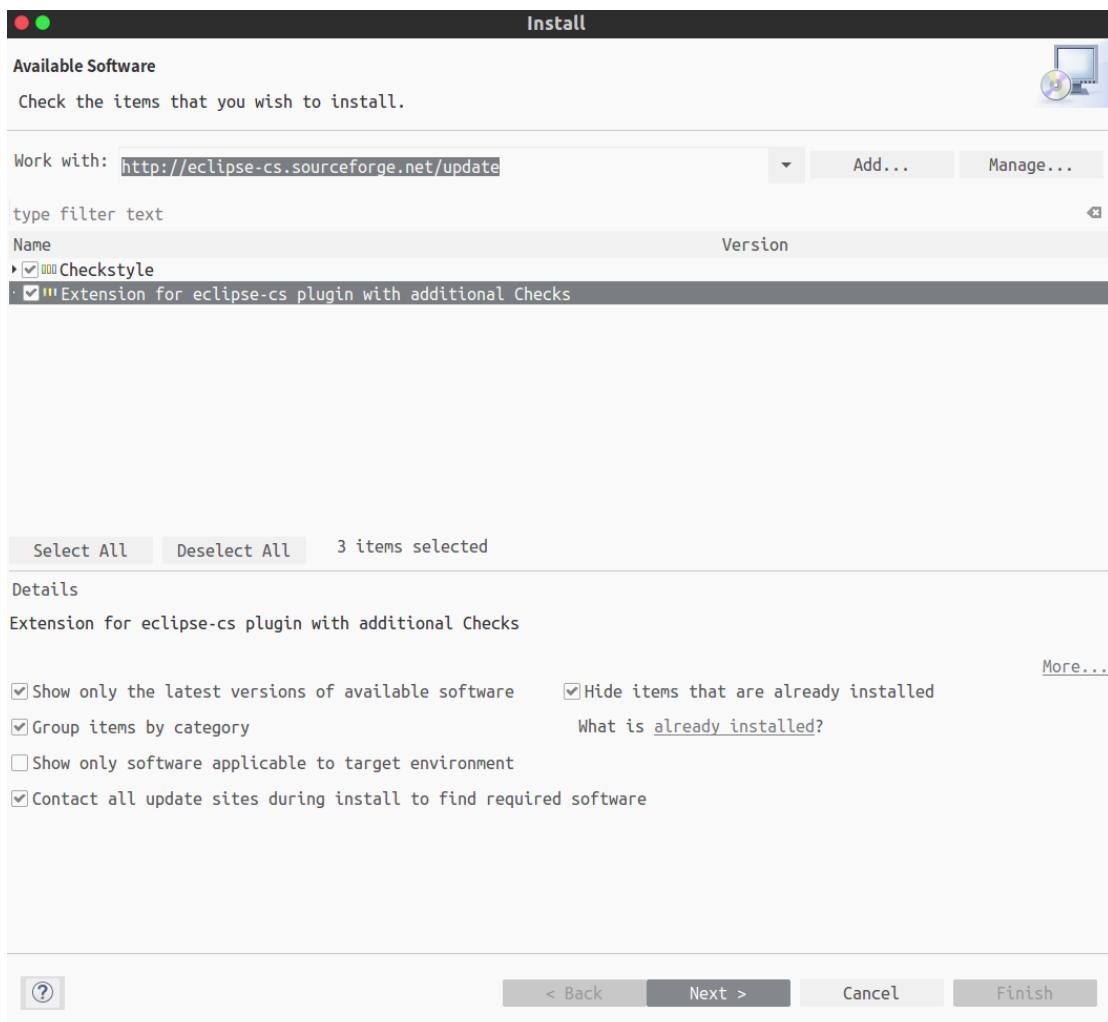
You're close!

This URL is an Eclipse **software repository**; you must use it in Eclipse ([see how](#)).

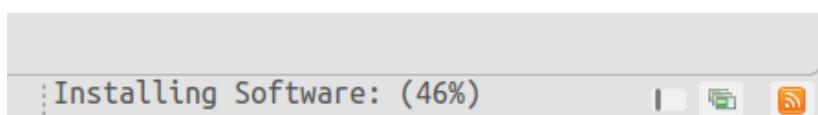


(3) CheckStyle 的安装

采用在线安装的方法：



下图显示正在安装



3 实验过程

3.1 Static Program Analysis

3.1.1 人工代码走查 (walkthrough)

发现的最多的一个问题就是缩进问题，比如是 if-else 的缩进，括号内要有四个缩

进。还有类中的方法缩进，注释的格式，Java Doc 的书写规范等。

比如下面的注释缩进：

```
// Abstraction function:  
//   -allVertices is a list where stored all the vertices in the graph  
//   -allEdges is a list where stored all the edges in the graph.  
//   -The ConcreteGraph class can create a graph.  
// Representation invariant:  
//   -allVertices is a list of L objects, which does not allow the sam  
//   -allEdges is a list of E objects, which also does not allow the s
```

还有一些 if-else 块中的代码，进行修改之后，没有很好的控制缩进，虽然说不影响逻辑，但是确实很不美观。

版本号如下：

```
● ● ●          muty@muty-Lenovo-ideapad-Y700-15ISK:~/Lab5-1160301008  
muty@muty-Lenovo-ideapad-Y700-15ISK:~/Lab5-1160301008$ git log  
commit 26567da2d35dab948c2be01de095d4ad3712953f  
Author: tianyu_mmmu <1417553133@qq.com>  
Date:   Sat May 26 21:04:17 2018 +0800  
  
        code after walkthrough and review
```

commit 版本号前六位：26567d

3.1.2 使用 CheckStyle 和 FindBugs 进行静态代码分析

(1) 首先采用 CheckStyle 进行分析，采用 Google 规范，首先，得到的一个问题是，每一行的缩进，提示信息是每一行都有 tab 符号，我一开始就一下子懵掉了，怎么可能每一行都有问题，难道不是用 tab 控制缩进么？后来进行查找资料发现，是现在要求要使用 2 个空格来进行缩进，而不是 4 个，这是最新的要求。

(2) 我将检查的要求调高了一点，然后首先声明数组要用如下实例：

```
String[] args = new int[]{1, 2, 3};
```

不可以使用 String args[], 即使这符合 Java 语法

(3) 变量命名的时候要采用驼峰命名法 (CamelCase)

(4) 注释的缩进，以及花括号的缩进也很重要。

```
muty@muty-Lenovo-ideapad-Y700-15ISK:~/Lab5-1160301008$ git log
commit c8a645ecab39841ac91b1f270699eb919ec00fb2
Author: tianyu_mmmu <1417553133@qq.com>
Date:   Sun May 27 19:35:44 2018 +0800

        Code after the CheckStyle and FindBugs
```

commit 版本号前六位: c8a645

3.2 Java I/O

3.2.1 多种 I/O 实现方式

1. 实现了哪些 I/O 方式来读写文件，具体如何实现的。

(1) 采用了 Stream 方法、Reader/Writer 方法、Files 方法来进行文件的读写。

首先是 Stream 方法，借助于 FileInputStream 类来进行文件输入流的打开，并进行文件读取，核心代码如下图所示：

```
FileInputStream inputStream = new FileInputStream(fileName);
BufferedReader bf = new BufferedReader(new InputStreamReader(inputStream));
String string = bf.readLine();
int num = 1;
while(string != null) {
    if(!string.matches("\b*")) {
        if(graph instanceof GraphPoet)
            GraphPoetFactory.parse(string);
        else if(graph instanceof SocialNetwork)
            SocialNetworkFactory.parse(string);
        else if(graph instanceof MovieGraph)
            MovieGraphFactory.parse(string);
        else NetworkTopologyFactory.parse(string);
        ++num;
    }
    if(num % 100 == 0)
        System.out.println(num / 100);
    Thread.sleep(100);
    string = bf.readLine();
}
```

(2) 其次是 Reader 方法，这个方法很熟悉，一般我都是采用如下方法来进行文件的读取操作，只需要用 BufferedReader 类即可，进行文件的按行读取就好。核心代码如下图所示：

```

public void readGraphFromFiles(String fileName, Graph<Vertex, Edge> graph) {
    BufferedReader bf = null;
    int num = 1;
    String string;
    try {
        bf = new BufferedReader(new FileReader(fileName));
    } catch (Exception e) {
        System.err.println("Wrong path!");
    }
    long time = System.currentTimeMillis();
    try {
        string = bf.readLine();
        while(string != null) {
            System.err.println(string.length());
            if(!string.matches("\b*")) {
                if(graph instanceof GraphPoet)
                    GraphPoetFactory.parse(string);
                else if(graph instanceof SocialNetwork)
                    SocialNetworkFactory.parse(string);
                else if(graph instanceof MovieGraph)
                    MovieGraphFactory.parse(string);
                else NetworkTopologyFactory.parse(string);
                ++num;
            }
            if(num % 100 == 0)
                System.out.println(num / 100);
    }
}

```

(3)最后是 Files 类进行文件读取，由于之前没有采用过这种方法来进行读文件，所以在网上百度了很久才大概了解了一点，Files.readAllLines()会将文件中的所有字符串用'\n'来进行分割，然后返回一个 String 型的 List，最后只需要去遍历这个 List 去进行建图即可。核心代码如下：

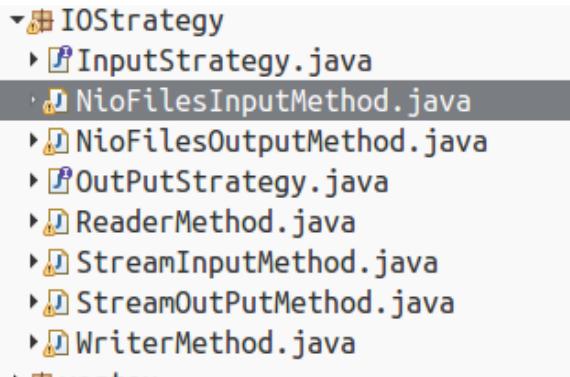
```

public class NioFilesInputMethod implements InputStrategy {
    public void readGraphFromFiles(String fileName, Graph<Vertex, Edge> graph) {
        try {
            int num = 1;
            long time = System.currentTimeMillis();
            List<String> stringStream=Files.readAllLines(Paths.get(fileName));
            for(String string: stringStream) {
                System.err.println(string.length());
                if(!string.matches("\b*")) {
                    if(graph instanceof GraphPoet)
                        GraphPoetFactory.parse(string);
                    else if(graph instanceof SocialNetwork)
                        SocialNetworkFactory.parse(string);
                    else if(graph instanceof MovieGraph)
                        MovieGraphFactory.parse(string);
                    else NetworkTopologyFactory.parse(string);
                    ++num;
                }
                if(num % 100 == 0)
                    System.out.println(num / 100);
                Thread.sleep(100);
            }
            long totalTime = System.currentTimeMillis() - time;
            System.out.println("The reading costs " + totalTime + " ms");
        } catch (IOException e) {
        }
    }
}

```

2. 如何用 strategy 设计模式实现在多种 I/O 策略之间的切换。

目录如下图：



首先有两个接口类: InputStrategy.java 和 OutputStrategy.java, 三种读写策略都是实现自这两个接口

具体接口中的代码如下:

```
public interface InputStrategy {
    public void readGraphFromFiles(String fileName, Graph<Vertex, Edge> graph);
}

public interface OutPutStrategy {
    public void writeToFileFromGraph(String fileName, Graph<Vertex, Edge> graph);
}
```

然后, 改写 factory 中的方法, 这里以 GraphPoetFactory 为例:

```
@Override
public Graph<Vertex, Edge> createGraph(String filePath) {
    System.out.println("Please input the I/O method:");
    System.out.println("1. Stream");
    System.out.println("2. Reader");
    System.out.println("3. java.nio.file.Files");
    Scanner input = new Scanner(System.in);
    int number = Integer.valueOf(input.nextLine());
    if(number == 1) {
        new StreamInputMethod().readGraphFromFiles(filePath, graph);
    }
    else if(number == 2) {
        new ReaderMethod().readGraphFromFiles(filePath, graph);
    }
    else {
        new NioFilesInputMethod().readGraphFromFiles(filePath, graph);
    }
    return graph;
}
```

createGraph 方法中我采取了二级菜单的策略, 让用户选择去用哪种方式去读写,

然后根据用户的输入去调用不同的方法, 具体效果如下:

```
Console [GraphPoetApp [Java Application] /opt/java/bin/java (2018年5月30日 下午8:44:35)]
-----GraphPoetApp-----
1. Input the filepath of graph
2. Command for graph
3. Vertex centrality
4. Graph centrality, radius and diameter
5. Shortest distance between two vertices
6. Exit
7. Filter the edge
8. GUI
9. Write the graph to files
-----Dedigned by muty

Please choose a function(1 - 9) and input the number.
1
Function 1, please input the path to concrete the graph.
src/GraphPoet.txt
Please input the I/O method:
1. Stream
2. Reader
3. java.nio.file.Files
1
The reading costs 59 ms
Concrete graph successfully!
Please choose a function(1 - 9) and input the number.
```

输出文件的时候也是一样:

```
9
Choose the method to write
1. Stream
2. Writer
3. Nio.Files
```

这样就实现了 strategy 模式，随意根据用户的需求去进行切换。

commit 版本号如下：

前六位: fed7af

```
commit fed7af80d2562412a13313287b4e61294c750301
Author: tianyu_mmmu <1417553133@qq.com>
Date:   Wed May 30 21:08:45 2018 +0800           test
Code after Strategy Design pattern for I/O
```

3.2.2 多种 I/O 实现方式的效率对比分析

(1)如何收集你的程序 I/O 语法文件的时间。

在进行 I/O 的之前调用 System.currentTimeMillis()方法来进行当前时间的获取，然后在 I/O 之后再获取一次时间，然后两个时间做差，即可获取 I/O 的总时间。具体代码如下：

```

3*import java.io.IOException;
1 public class NioFilesInputMethod implements InputStrategy {
2  public void readGraphFromFiles(String fileName, Graph<Vertex, Edge> graph) {
3   try {
4    int num = 1;
5    long time = System.currentTimeMillis();
6    List<String> stringStream=Files.readAllLines(Paths.get(fileName));
7    for(String string: stringStream) {
8     if(!string.matches("\\\\b*")) {
9      if(graph instanceof GraphPoet)
10       GraphPoetFactory.parse(string);
11      else if(graph instanceof SocialNetwork)
12       SocialNetworkFactory.parse(string);
13      else if(graph instanceof MovieGraph)
14       MovieGraphFactory.parse(string);
15      else NetworkTopologyFactory.parse(string);
16      ++num;
17     }
18     if(num % 100 == 0)
19      System.out.println(num / 100);
20    }
21    long totalTime = System.currentTimeMillis() - time;
22    System.out.println("The reading costs " + totalTime + " ms");
23  } catch (IOException e) {
24   e.printStackTrace();
25  }
26 }

```

其中，获取时间的代码如下：

```

..,
int num = 1;
long time = System.currentTimeMillis();

}
long totalTime = System.currentTimeMillis() - time;
System.out.println("The reading costs " + totalTime + " ms");
catch (IOException e) {

```

(2)表格方式对比不同 I/O 的性能。

有如下表格：

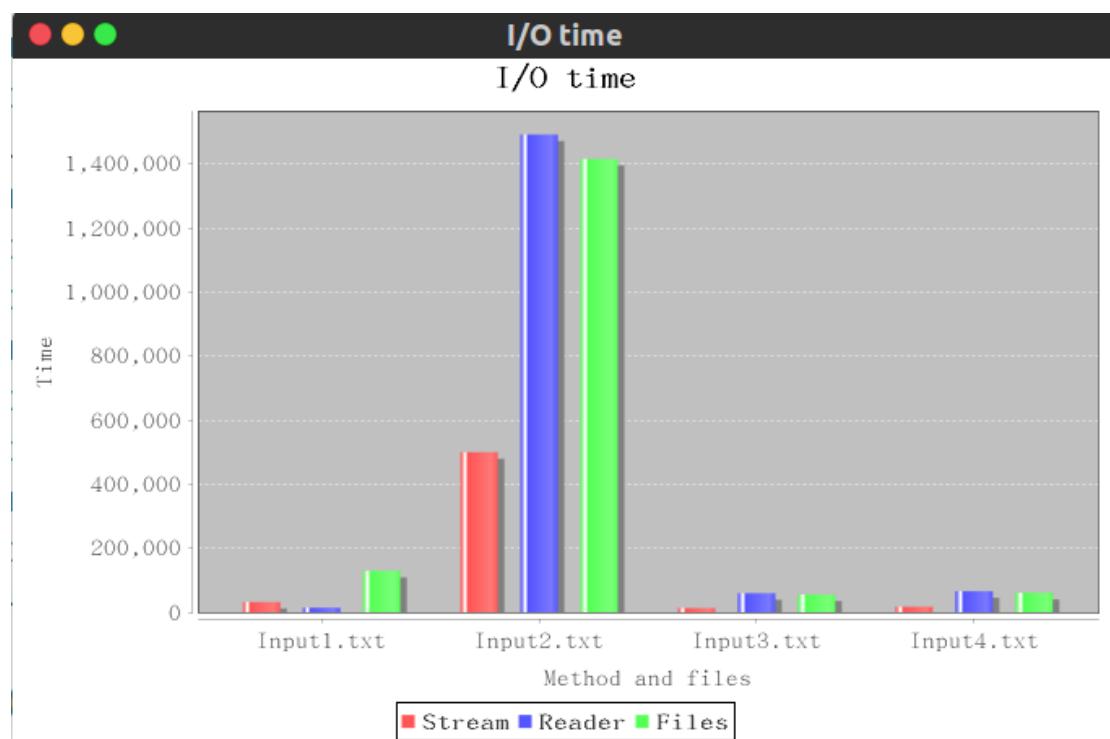
		file1.txt	file2.txt	file3.txt	file4.txt
Stream	读文件并建图	31966ms	499902ms	12663ms	17130ms
	写文件	1082ms	1176ms	1379ms	1046ms
	读文件并建	140419ms	1492216ms	59081ms	65296ms

Reader/Writ er	图				
	写文件	395ms	412ms	603ms	338ms
java.nio.file .Files	读文件并建 图	129173ms	1416286ms	55134ms	60157ms
	写文件	865ms	1000ms	1160ms	932ms

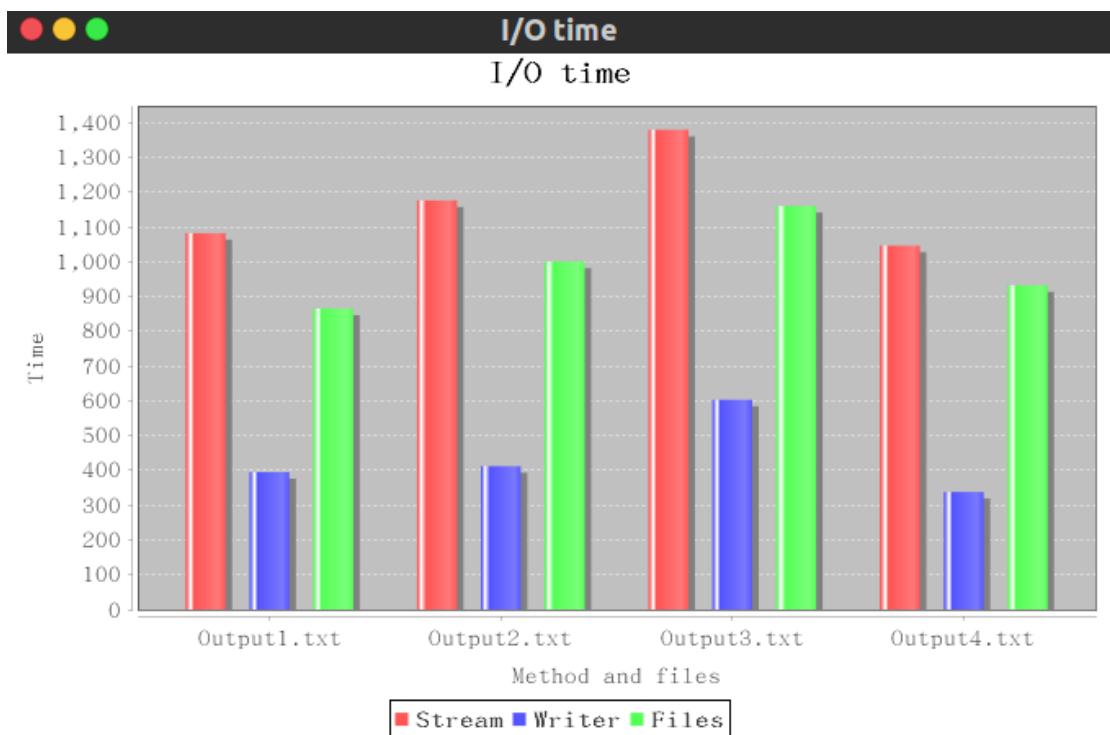
(3)图形对比不同 I/O 的性能 (可选)。

采用老师推荐的 JFreeChart 来绘制柱状图

读入文件时对比图如下(单位: ms):



写入文件时对比图如下(单位: ms):



其中数据均来自于上表中的数据。

通过分析数据，可知，在读文件的时候，stream 是最快的，文件越大，优势越大，Reader 是最慢的。

在写文件的时候，Writer 是最快的，Stream 最慢。

3.3 Java Memory Management and Garbage Collection (GC)

3.3.1 使用-verbose:gc 参数

首先，按照实验手册设置 Run Configurations 如下：

```
VM arguments:
-verbose:gc
-Xloggc:src/log.log
-XX:+PrintGCDetails
-XX:+PrintGCDateStamps
-XX:+PrintGCTimeStamps
```

将日志文件输出到 log.log 文件中，并进行统计，截取部分日志文件内容如下：

```
Java HotSpot(TM) 64-Bit Server VM (25.162-b12) for linux-amd64 JRE (1.8.0_162-b12), built on Dec 19 2017 21:15:48 by "java_re" with gcc 4.3.0 20080428 (Red Hat 4.3.0-8)
Memory: 4k page, physical 7992208K(3966604K free), swap 1999868K(1999868K free)
CommandLine flags: -XX:InitialHeapSize=127875328 -XX:MaxHeapSize=2046085248 -XX:+PrintGCTimeStamps -XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseParallelGC
13.593: [GC (Allocation Failure) 31232K->2161K(1999868K), 0.0001804 secs]
13.601: [GC (Allocation Failure) 1999868K->1999868K, 0.0001804 secs]
13.634: [Full GC (Ergonomics) 58369K->57827K(159744K), 0.1429882 secs]
13.845: [GC (Allocation Failure) 89059K->67842K(159744K), 0.0051903 secs]
13.893: [GC (Allocation Failure) 95974K->67882K(188928K), 0.0068893 secs]
13.975: [GC (Allocation Failure) 12878K->67882K(188928K), 0.0068893 secs]
14.001: [GC (Allocation Failure) 68378K->68138K(288576K), 0.0047921 secs]
14.117: [GC (Allocation Failure) 184354K->68202K(281680K), 0.0041945 secs]
```

可以看出来，在所有的 GC 中，Full GC 出现的频率并不高。想要具体查看时间间隔，就要具体打印 GC 的信息如下：

```
OPTION MESSAGE, TEXT | 打印堆内存使用情况 | 2018-05-25T11:53:08.763+0800:
2018-05-25T11:53:08.763+0800:
18.275: [GC (Allocation Failure) [PSYoungGen: 675646K->6517K(677888K)]
707869K->46651K(736768K), 0.0145812 secs]
[Times: user=0.13 sys=0.00, real=0.01 secs]

2018-05-25T11:53:09.764+0800:
19.166: [GC (Allocation Failure) [PSYoungGen: 674677K->6513K(677888K)]
714811K->52871K(736768K), 0.0131434 secs]
[Times: user=0.13 sys=0.00, real=0.01 secs]

2018-05-25T11:53:10.663+0800:
20.065: [GC (Allocation Failure) [PSYoungGen: 674673K->6825K(678400K)]
721031K->58983K(737280K), 0.0161280 secs]
[Times: user=0.09 sys=0.00, real=0.02 secs]

2018-05-25T11:53:10.679+0800:
20.081: [Full GC (Ergonomics) [PSYoungGen: 6825K->0K(678400K)]
[ParOldGen: 52157K->19341K(89600K) 58983K->19341K(768000K), [Metaspace: 5705K->5705K(1056768K)], 0.1896514 secs]
[Times: user=0.95 sys=0.00, real=0.19 secs]

2018-05-25T11:53:11.708+0800:
21.110: [GC (Allocation Failure) [PSYoungGen: 668672K->7262K(678400K)]
688013K->26604K(768000K), 0.0102940 secs]
[Times: user=0.13 sys=0.00, real=0.01 secs]

2018-05-25T11:53:12.639+0800:
22.040: [GC (Allocation Failure) [PSYoungGen: 675934K->5920K(677376K)]
695276K->32484K(766976K), 0.0148362 secs]
[Times: user=0.11 sys=0.00, real=0.01 secs]
```

可以看出来，两次 MinorGc 发生的间隔大概在 10ms 左右，而两次 Full GC 发生的时间间隔大概是 300ms，可以分析出，年轻一代的生命周期较短，所以垃圾回收机制才会效率较高。

改变-Xmn 参数，可以改变 Minor GC 的频率，改变-Xms 和-Xmx 参数，可以改变 Full GC 的频率。

3.3.2 用 jstat 命令行工具的-gc 和-gcutil 参数

采用 jstat -gc -h5 -t 5619 1000 命令（5619 为 Pid）

muty@muty-Lenovo-ideapad-Y700-15ISK:~																		
Timestamp	SOC	S1C	S0U	S1U	EC	EU	OC	OU	MC	MU	CCSC	CCSU	YGC	YGCT	FGC	FGCT	G	
0.918	149.1	1024.0	1024.0	448.0	0.0	664064.0	318748.9	112128.0	95139.1	4864.0	3819.7	512.0	414.5	320	0.688	1	0.230	
0.918	150.1	1024.0	1024.0	448.0	0.0	664064.0	318748.9	112128.0	95139.1	4864.0	3819.7	512.0	414.5	320	0.688	1	0.230	
0.918	151.1	1024.0	1024.0	448.0	0.0	664064.0	318748.9	112128.0	95139.1	4864.0	3819.7	512.0	414.5	320	0.688	1	0.230	
0.918	152.1	1024.0	1024.0	448.0	0.0	664064.0	318748.9	112128.0	95139.1	4864.0	3819.7	512.0	414.5	320	0.688	1	0.230	
0.918	153.1	1024.0	1024.0	448.0	0.0	664064.0	318748.9	112128.0	95139.1	4864.0	3819.7	512.0	414.5	320	0.688	1	0.230	
0.918	154.1	1024.0	1024.0	448.0	0.0	664064.0	318748.9	112128.0	95139.1	4864.0	3819.7	512.0	414.5	320	0.688	1	0.230	
0.918	155.1	1024.0	1024.0	448.0	0.0	664064.0	318748.9	112128.0	95139.1	4864.0	3819.7	512.0	414.5	320	0.688	1	0.230	
0.918	156.1	1024.0	1024.0	448.0	0.0	664064.0	318748.9	112128.0	95139.1	4864.0	3819.7	512.0	414.5	320	0.688	1	0.230	
0.918	157.1	1024.0	1024.0	448.0	0.0	664064.0	318748.9	112128.0	95139.1	4864.0	3819.7	512.0	414.5	320	0.688	1	0.230	
0.918	158.1	1024.0	1024.0	448.0	0.0	664064.0	318748.9	112128.0	95139.1	4864.0	3819.7	512.0	414.5	320	0.688	1	0.230	
0.918	159.1	1024.0	1024.0	448.0	0.0	664064.0	318748.9	112128.0	95139.1	4864.0	3819.7	512.0	414.5	320	0.688	1	0.230	
0.918	160.1	1024.0	1024.0	448.0	0.0	664064.0	318748.9	112128.0	95139.1	4864.0	3819.7	512.0	414.5	320	0.688	1	0.230	
0.918	161.1	1024.0	1024.0	448.0	0.0	664064.0	318748.9	112128.0	95139.1	4864.0	3819.7	512.0	414.5	320	0.688	1	0.230	
0.918	162.1	1024.0	1024.0	448.0	0.0	664064.0	318748.9	112128.0	95139.1	4864.0	3819.7	512.0	414.5	320	0.688	1	0.230	
0.918	163.1	1024.0	1024.0	448.0	0.0	664064.0	318748.9	112128.0	95139.1	4864.0	3819.7	512.0	414.5	320	0.688	1	0.230	
0.918	164.1	1024.0	1024.0	448.0	0.0	664064.0	318748.9	112128.0	95139.1	4864.0	3819.7	512.0	414.5	320	0.688	1	0.230	
0.918	165.1	1024.0	1024.0	448.0	0.0	664064.0	318748.9	112128.0	95139.1	4864.0	3819.7	512.0	414.5	320	0.688	1	0.230	
0.918	166.1	1024.0	1024.0	448.0	0.0	664064.0	318748.9	112128.0	95139.1	4864.0	3819.7	512.0	414.5	320	0.688	1	0.230	

对于 file1.txt 进行循环读取，每次采取 YoungGC 320 次，FullGC 1 次，YoungGC 的时间大概在 0.688s，FullGC 的时间为 0.23s 左右，根据老师给的博客上面的内容去进行分析，并无什么异常。

采用 jstat -gcutil 15060 1000 1000 命令（15060 为 Pid）

muty@muty-Lenovo-ideapad-Y700-15ISK:~																		
\$ jstat -gcutil 15060 1000 1000																		
S0	S1	E	O	M	CCS	YGC	YGCT	FGC	FGCT	GCT								
0.00	0.00	0.00	25.61	95.36	82.89	68	0.145	0	0.000	0.145								
68.75	0.00	10.00	27.20	95.36	82.89	74	0.155	0	0.000	0.155								
0.00	37.50	18.00	28.71	95.36	82.89	79	0.164	0	0.000	0.164								
62.50	0.00	64.00	30.25	95.36	82.89	84	0.173	0	0.000	0.173								
50.00	0.00	66.00	31.91	95.36	82.89	90	0.183	0	0.000	0.183								
0.00	68.75	24.00	34.81	95.36	82.89	97	0.198	0	0.000	0.198								
93.75	0.00	68.00	35.93	95.36	82.89	100	0.203	0	0.000	0.203								
0.00	46.88	100.00	37.44	95.36	82.89	103	0.209	0	0.000	0.209								
0.00	40.62	78.00	39.09	95.36	82.89	107	0.218	0	0.000	0.218								
0.00	40.62	0.00	41.09	95.36	82.89	113	0.228	0	0.000	0.228								
0.00	21.88	24.00	42.64	95.36	82.89	119	0.238	0	0.000	0.238								
56.25	0.00	10.00	44.31	95.36	82.89	126	0.249	0	0.000	0.249								
81.25	0.00	40.00	45.48	95.36	82.89	130	0.257	0	0.000	0.257								
0.00	43.75	48.00	46.62	95.36	82.89	133	0.267	0	0.000	0.267								
93.75	0.00	54.00	48.00	95.36	82.89	136	0.273	0	0.000	0.273								
75.00	0.00	12.00	49.68	95.36	82.89	140	0.281	0	0.000	0.281								
68.75	0.00	72.00	51.05	95.36	82.89	144	0.289	0	0.000	0.289								
0.00	25.00	0.00	52.97	95.36	82.89	151	0.301	0	0.000	0.301								
68.75	0.00	68.00	54.30	95.36	82.89	156	0.309	0	0.000	0.309								
62.50	0.00	56.00	55.97	95.36	82.89	162	0.319	0	0.000	0.319								
62.50	0.00	72.00	57.58	95.36	82.89	168	0.329	0	0.000	0.329								
0.00	81.25	12.00	59.15	95.36	82.89	173	0.339	0	0.000	0.339								
0.00	34.38	46.00	60.64	95.36	82.89	177	0.349	0	0.000	0.349								
81.25	0.00	62.47	63.36	95.36	82.89	182	0.358	0	0.000	0.358								
0.00	56.25	30.00	66.46	95.36	82.89	187	0.376	0	0.000	0.376								
87.50	0.00	94.00	67.46	95.36	82.89	190	0.377	0	0.000	0.377								
93.75	0.00	6.00	69.48	95.36	82.89	194	0.386	0	0.000	0.386								
0.00	40.62	88.00	70.78	95.36	82.89	197	0.394	0	0.000	0.394								
75.00	0.00	54.00	72.56	95.36	82.89	202	0.405	0	0.000	0.405								
0.00	75.00	76.00	74.02	95.36	82.89	207	0.414	0	0.000	0.414								
68.75	0.00	46.00	75.76	95.36	82.89	212	0.425	0	0.000	0.425								
0.00	81.25	34.00	77.27	95.36	82.89	217	0.435	0	0.000	0.435								
68.75	0.00	0.00	79.04	95.36	82.89	222	0.444	0	0.000	0.444								
0.00	68.75	40.00	80.51	95.36	82.89	227	0.453	0	0.000	0.453								
68.75	0.00	78.00	81.93	95.36	82.89	232	0.463	0	0.000	0.463								
0.00	75.00	62.00	83.64	95.36	82.89	237	0.472	0	0.000	0.472								
81.25	0.00	40.00	85.18	95.36	82.89	242	0.482	0	0.000	0.482								
0.00	75.00	16.00	86.92	95.36	82.89	247	0.491	0	0.000	0.491								
0.00																		

3.3.3 使用 jmap -heap 命令行工具

指令 jmap -heap Pid

```

using thread-local object allocation.
Parallel GC with 8 thread(s) //同步并行垃圾回收

Heap Configuration:
MinHeapFreeRatio      = 0          //最小堆使用比例
MaxHeapFreeRatio      = 100        //最大堆使用比例
MaxHeapSize           = 2113929216 (2016.0MB) //最大堆空间大小
NewSize               = 44040192 (42.0MB)   //新生代分配大小
MaxNewSize             = 704643072 (672.0MB) //最大新生代可分配大小
OldSize               = 88080384 (84.0MB)  //OldGen大小
NewRatio              = 2          //新生代比例
SurvivorRatio         = 8          //新生代与survivor的比例
MetaspaceSize          = 21807104 (20.796875MB) //Metaspace区分配大小
CompressedClassSpaceSize = 1073741824 (1024.0MB)
MaxMetaspaceSize       = 17592186044415 MB
G1HeapRegionSize       = 0 (0.0MB)

Heap Usage:
PS Young Generation
Eden Space:
capacity = 683147264 (651.5MB)          //Eden区总大小
used     = 517875672 (493.88472747802734MB) //Eden已经使用的大小
free     = 165271592 (157.61527252197266MB) //Eden剩余容量的大小
75.80732578327357% used                //Eden使用比例

From Space:
capacity = 6815744 (6.5MB)                //survivor1区容量
used     = 6814552 (6.498683220214844MB)  //survivor1已使用容量
free     = 1192 (0.00113677978515625MB)   //survivor1剩余容量
99.98251108022836% used                  //survivor1使用比例

To Space:
capacity = 8912896 (8.5MB)                //survivor2区容量
used     = 0 (0.0MB)                      //survivor2区已使用容量
free     = 8912896 (8.5MB)                //survivor2剩余容量
0.0% used                                //survivor2使用比例

PS Old Generation
capacity = 88080384 (84.0MB)              //OldGen大小
used     = 19030208 (18.14862060546875MB) //OldGen已使用大小
free     = 69050176 (65.85137939453125MB) //OldGen剩余大小
21.60550072079613% used                  //OldGen使用比例

3360 interned Strings occupying 267520 bytes.

```

3.3.4 使用 jmap -histo 命令行工具 (可选)

num	#instances	#bytes	class	name
1:	1384141	54410872	[C	
2:	1384137	33219288	java.lang.String	
3:	11899	8612504	[I	
4:	80485	4510968	[Ljava.lang.Object;	
5:	159764	3834336	java.util.ArrayList	
6:	83913	2685216	java.util.HashMap\$Node	
7:	79874	2555968	edge.NetworkConnection	
8:	35	1043696	[Ljava.util.HashMap\$Node;	
9:	3897	124704	java.util.regex.Pattern\$Curly	
10:	1363	93816	java.util.regex.Pattern	
11:	3677	88248	java.util.regex.Pattern\$Ctype	
12:	1299	83136	java.util.regex.Matcher	
13:	766	79528	java.lang.Class	
14:	3030	72720	java.util.regex.Pattern\$GroupHead	
15:	3030	72720	java.util.regex.Pattern\$GroupTail	
16:	1296	72576	[Ljava.util.regex.Pattern\$GroupHead;	
17:	2383	57192	java.util.regex.Pattern\$Single	
18:	21	49840	[B	
19:	1731	41544	java.util.regex.Pattern\$Slice	
20:	1081	34592	java.util.regex.Pattern\$BnM	
21:	1000	24000	vertex.Computer	
22:	1000	24000	vertex.Router	
23:	1000	24000	vertex.Server	
24:	781	18744	java.util.LinkedList\$Node	
25:	391	12512	java.util.LinkedList	
26:	217	8680	java.util.HashMap\$KeyIterator	
27:	253	8096	java.util.concurrent.ConcurrentHashMap\$Node	
28:	181	7688	[Ljava.lang.String;	
29:	216	6912	java.util.AbstractList\$Itr	
30:	93	5952	java.net.URL	
31:	82	5904	java.lang.reflect.Field	
32:	222	5328	java.util.regex.Pattern\$Start	
33:	219	5256	java.util.Arrays\$ArrayList	
34:	327	5232	java.lang.Integer	
35:	216	5184	[Lvertex.Vertex;	
36:	216	5184	java.util.regex.Pattern\$Bound	
37:	216	5184	java.util.regex.Pattern\$TreeInfo	
38:	143	4576	java.util.Hashtable\$Entry	

```

306:           1          16 sun.util.locale.provider.AuxLocaleProviderAdapter$NullProvider
307:           1          16 sun.util.locale.provider.SPILocaleProviderAdapter
308:           1          16 sun.util.resources.LocaleData
309:           1          16 sun.util.resources.LocaleData$LocaleDataResourceBundleControl
Total      3215899      111983144

```

采用命令行 jmap -histo PID，可以查看一些当前装载进内存的各类的实例数目及内存占用情况。

3.3.5 使用 jmap -permstat 命令行工具（可选）

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。

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

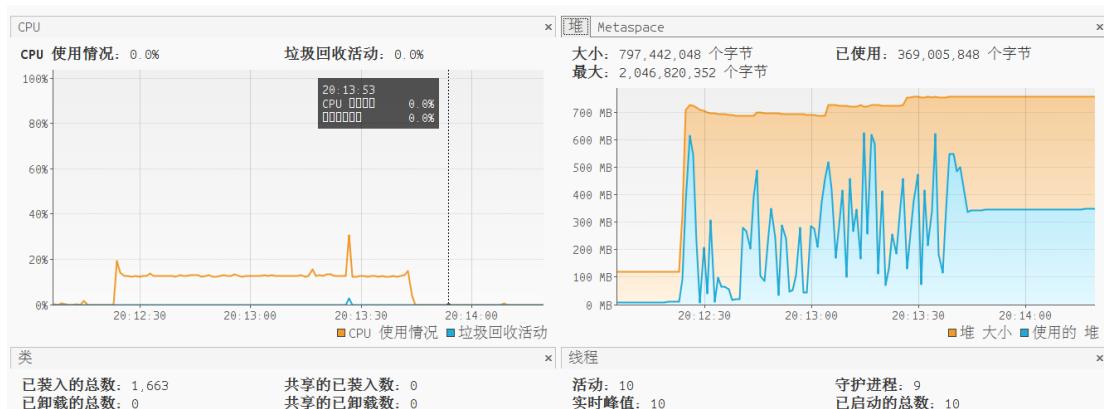
```

C:\Users\W24>jmap -clstats 19772
Attaching to process ID 19772, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.172-b11
finding class loader instances ..done.
computing per loader stat ..done.
please wait.. computing liveness.liveness analysis may be inaccurate ...
class_loader    classes   bytes   parent_loader   alive?   type
<bootstrap>      854     1644063    null        live    <internal>
0x0000000082299ea0      0         0 0x00000000820f94d0    dead    java/util/ResourceBundle$RBClassLoader@0x00000001000571f8
0x00000000820f94d0      47      47154 0x00000000820f9530    live    sun/misc/Launcher$AppClassLoader@0x000000010000f8d0
0x00000000820f9530      4      4551    null        live    sun/misc/Launcher$ExtClassLoader@0x000000010000fc78
total = 4      905     1695768    N/A      alive=3, dead=1    N/A

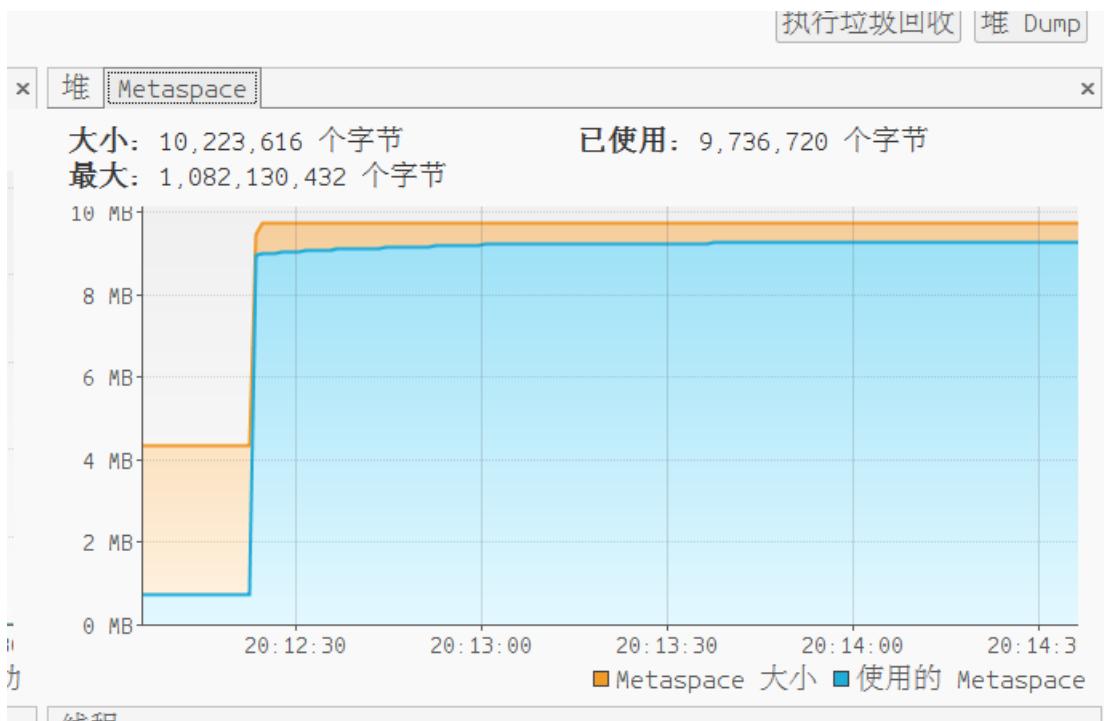
```

这里显示的是一次读取文件时装载的 class 和 method 相关信息。

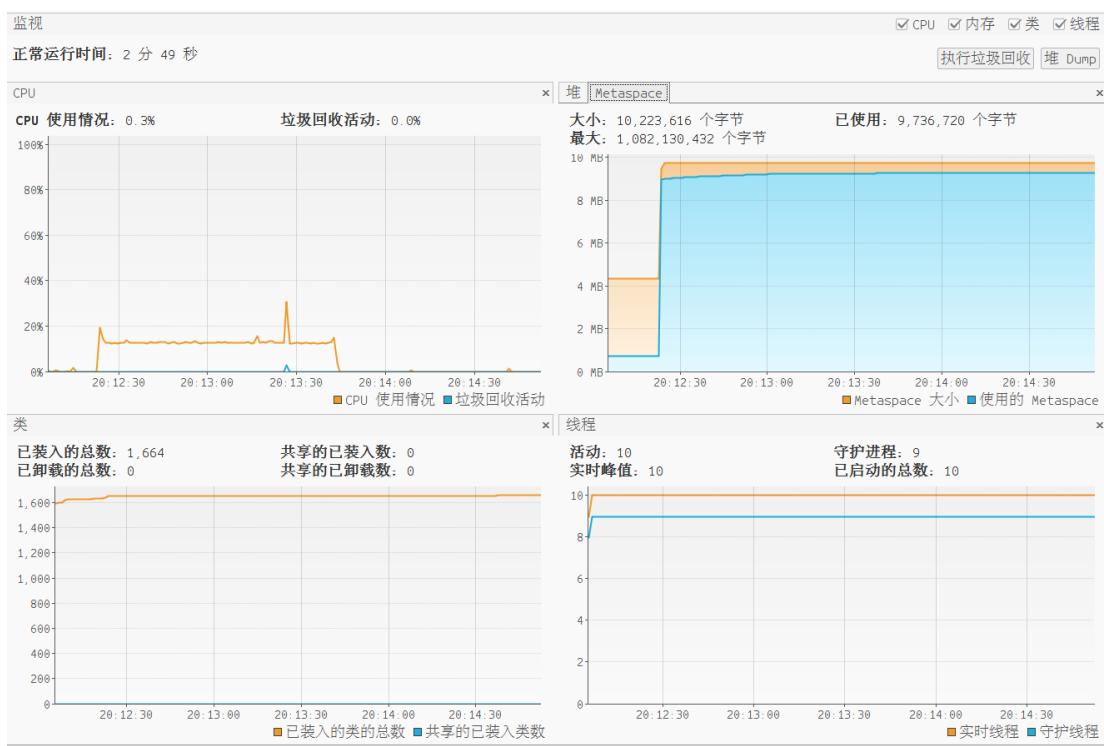
3.3.6 使用 jconsole 或 VisualVM 工具



可见, 当堆出现急速下降的时候, 也就是进行了 FullGC 的时候。



MetaSpace 接近于 10MB 左右，并且很稳定，没有骤升和骤降。



装载的时候，装载的类总数几乎没有变化。

3.3.7 分析垃圾回收过程是否正常、异常

通过 `jstat -gc` 和 `jstat -gcutil` 命令执行结果可以看到一次 Full GC 之后, OldGen (O) 区的内存被回收, 从 81.3% 变为了 13.01%。关于 Minor GC 与 Full GC 的时间在 `-gc` 指令中已经分析过。Metaspace (M) 区的占用率始终保持在 95% 左右, 没有内存突变, 比较正常。通过 VisualVM 中 Metaspace 占用空间的变化也可以看出, Metaspace 占用的区域基本上没有发生变化。如果 Minor GC 与 Full GC 都能够正常发生, 且都能有效回收内存, 且 M 区内有发生内存突变, 说明内存释放正常, 垃圾回收及时, 内存泄漏的几率大大降低。

3.3.8 配置 JVM 参数并发现最优参数配置

1. 随着测试可以发现, 如果增加堆的大小, 也就是 OldGen 的区域的大小增大, FULL GC 的次数会明显的降低, 由于 FULL GC 消耗的时间比较长, 所以适当的增加堆空间是可以有效地减少垃圾回收地时间的。所以调优主要是减少 FULL GC 的次数。

综上: 可以为 JVM 设置参数: (当然不考虑其他因素, 为虚拟机分配越大的资源, 得到的效果越好, 但是资源达到程度之后, 带来的效益提升会越来越低)

`-Xmx1000m`

`-Xms1000m`

`-Xmn500m`

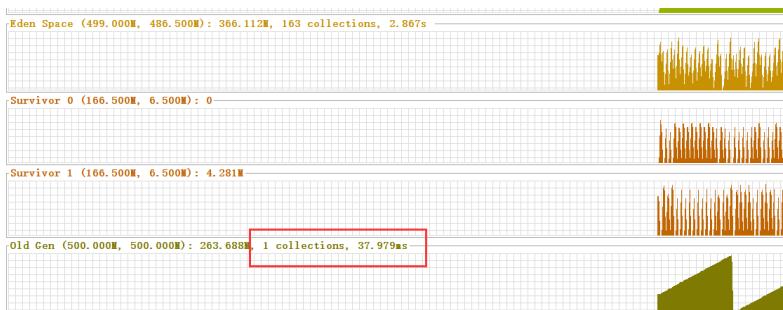
还可以设置新生代与堆大小的比例。

-XX:NewRatio=

新生代 Survivor 区与 Eden 区的比例

-XX:SurvivorRatio=4 (由于 Survivor 有两个区，所以为 2: 4)

由于增大了 OldGen 区域的大小，FULL GC 的次数明显的降低，从七次变为了一次。



3.4 Dynamic Program Profiling

3.4.1 使用 Visual VM 进行 CPU Profiling

Visual VM 能够监控应用程序在一段时间的 CPU 的使用情况，显示 CPU 的使用率、方法的执行效率。

我们进行如下的操作：

- 读取 file1.txt
- 添加一个新的顶点
- 添加一条新的边
- 删除符合正则表达式 regex 的顶点
- 删除符合正则表达式 regex 的边
- 修改顶点的信息
- 修改边的信息

```

input command(input "exit" to select function or "help" to check):
vertex --add "newvertex" "Computer"
complete!
edge --add "newedge" "NetworkConnection" weighted=Y 100 directed=N "newvertex" "Server1"
complete!
vertex --delete "Computer1"
complete!
edge --delete "S84"
complete!
vertex --modify oldlabel=newvertex newlabel=newvertex1 args="192.168.52.11"
complete!
edge --modify newedge label=Y newlabel weight=Y 200 direct=N
complete!

```

1. 可以通过监控界面来看执行各操作的 CPU 使用率



其中 1 为读取文件并创建图，2 为加入一个新的顶点，3 为加入一条新的边，
4 为删除符合 regex 的顶点，5 为删除符合 regex 的边（花费的时间比较长），
6 为更改顶点信息，7 为更改边的信息。

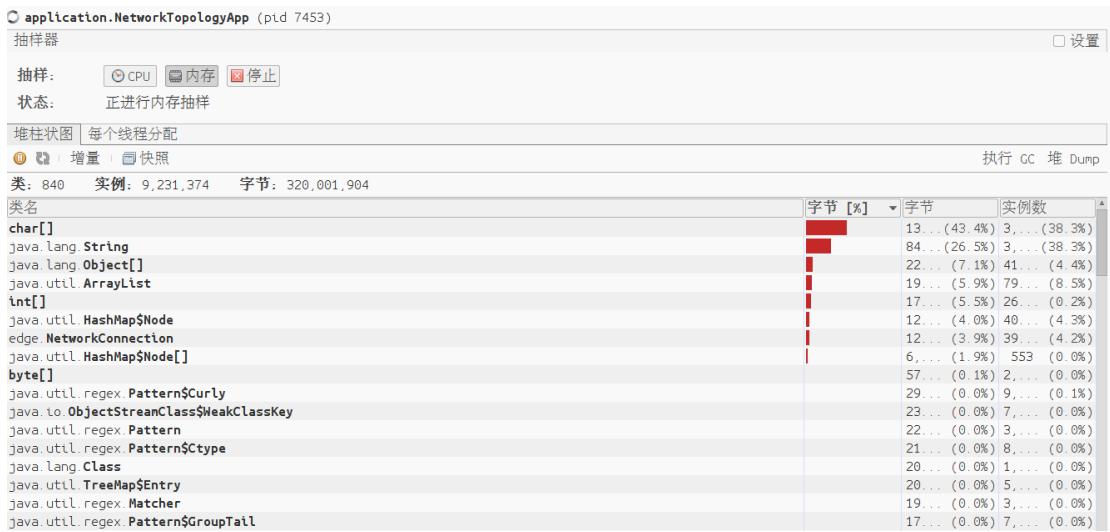
分析：

可以看出，删除符合正则表达式 regex 的边的操作 CPU 消耗相对较大，其他的操作 CPU 消耗并不大。

2. 将 List 替换成为 HashSet 可以更快的进行遍历。

3.4.2 使用 Visual VM 进行 Memory profiling

有如下的截图：



首先可以看到, char[]数组占据内存很大可以理解, 因为读文件、fillInfo 等操作是利用 char[]数组来进行的, 在刚刚读完文件并建完图之后, String 类和 List 类占的内存很大, 因为我是使用 String 类来进行存储文件的每一行, 并进行正则表达式的解析, 由于 String 是 immutable 类型的变量, 所以每读进来一行都会重新 new 一个对象来存储读进来的内容, 所以不难理解为什么 String 类会占据这么大内存。其次是 ArrayList, 因为我将所有的 Vertex 和所有的 Edge 都存在 List 中, 所以会占据较大内存。至于 Edge 类, 一定会占据很大内存的, 因为生成了很多 Edge 类的对象。所以, 综上所述, 这样的结果还是很符合我的代码的。

3.5 Memory Dump Analysis and Performance Optimization

3.5.1 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。

① 概述

基本信息：

生成的日期: Fri Jun 01 21:22:36 CST 2018
文件: /tmp/visualvm.dat/localhost_6953/heapdump-1527859355834.hprof
文件大小: 152 MB

字节总数: 141,584,174
类总数: 1,829
实例总数: 2,835,469
类加载器: 79
垃圾回收根节点: 1,574
等待结束的暂挂对象数: 0

环境：

操作系统: Linux (4.13.0-43-generic)
体系结构: amd64 64bit
Java 主目录: /opt/java/jre
Java 版本: 1.8.0_162
JVM: Java HotSpot(TM) 64-Bit Server VM (25.162-b12, mixed mode)
Java 供应商: Oracle Corporation

系统属性：

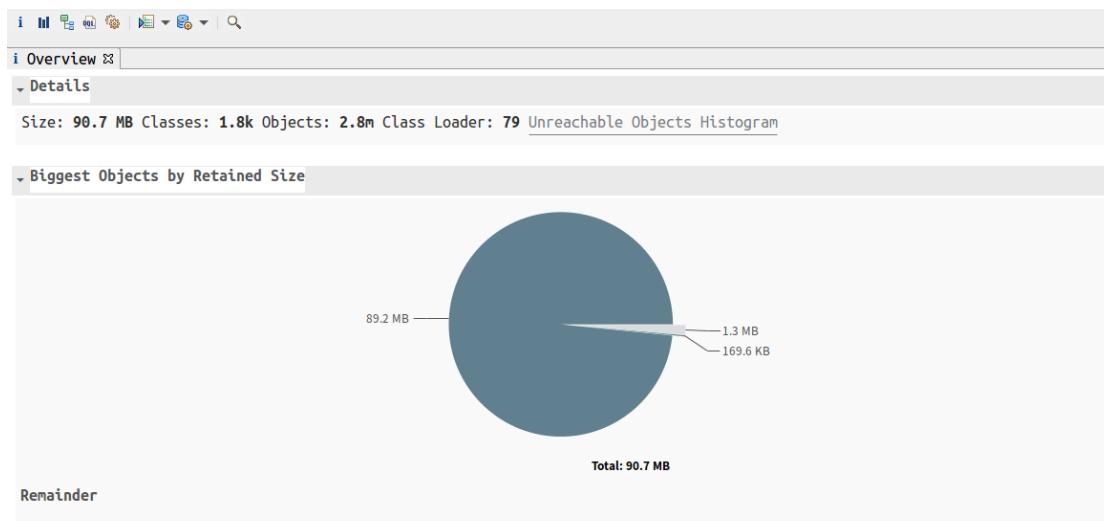
[显示系统属性](#)

堆转储上的线程：

[显示线程](#)

3.5.2 使用 MAT 分析内存导出文件

1. OverView



分析：灰绿色区域代表已经被分配的内存，空白的是剩余内存。

2.Histogram

从类的角度出发，进行分析

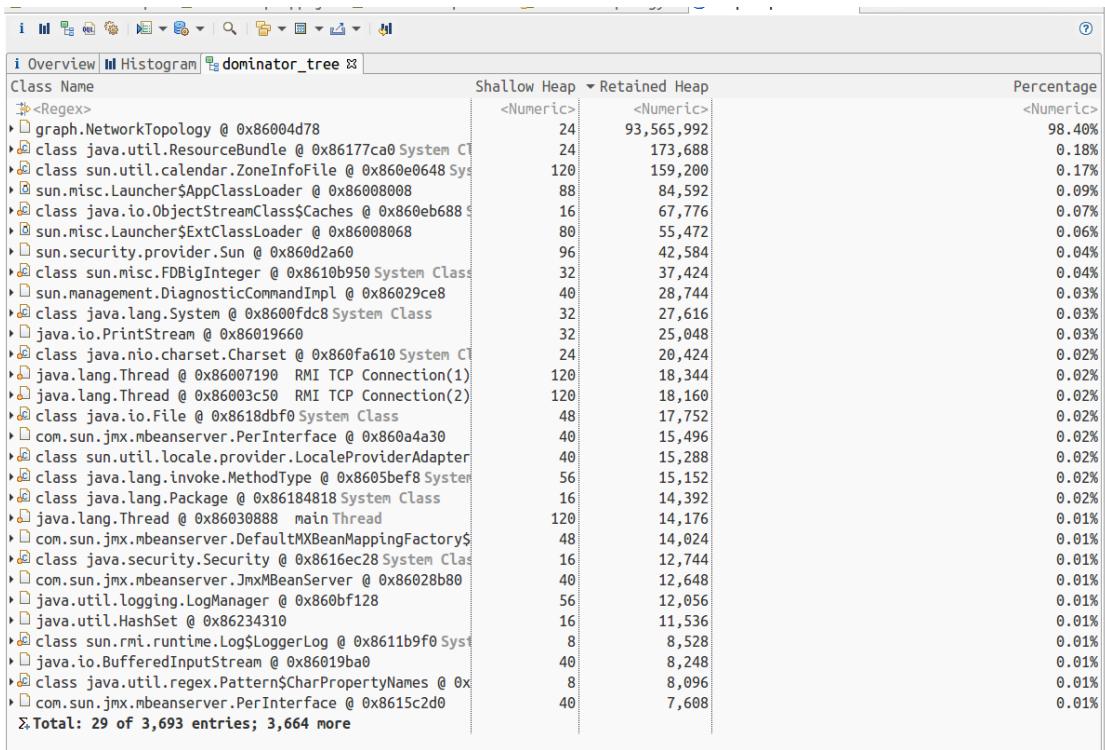
Class Name	Objects	Shallow Heap
	<Numeric>	<Numeric>
<Regex>	397,384	22,238,936
java.lang.Object[]	792,941	19,030,584
java.util.ArrayList	409,583	13,369,704
char[]	401,375	12,844,000
java.util.HashMap\$Node	396,418	12,685,376
edge.NetworkConnection	409,550	9,829,200
java.lang.String	213	4,242,792
java.util.HashMap\$Node[]	495	182,440
byte[]	495	43,560
java.lang.reflect.Method	481	39,768
int[]	841	33,640
java.util.LinkedHashMap\$Entry	1,000	24,000
vertex.Router	1,000	24,000
vertex.Server	1,000	24,000
vertex.Computer	619	23,808
java.lang.String[]	840	20,160
java.util.LinkedList\$Node	411	19,728
java.util.HashMap	592	18,944
java.util.concurrent.ConcurrentHashMap\$Node	1,776	16,568
java.lang.Class	422	13,504
java.util.LinkedList	489	11,872
java.lang.Class[]	341	10,912
sun.misc.FDBigInteger	242	9,680
java.lang.ref.SoftReference	125	9,576
java.util.Hashtable\$Entry[]	298	9,536
java.util.Hashtable\$Entry	121	7,744
java.net.URL	321	7,704
java.lang.Long	226	7,232
java.lang.invoke.LambdaForm\$Name	208	6,656
Σ Total: 29 of 1,767 entries; 1,738 more	2,827,934	95,086,264

Shallow Heap: 对象自身占用的内存大小，不包括它引用的对象。

在这里可以看出来顶点数、边数。

3. Dominator tree:

Dominator tree 意味着支配树，更善于去分析对象的引用关系。



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

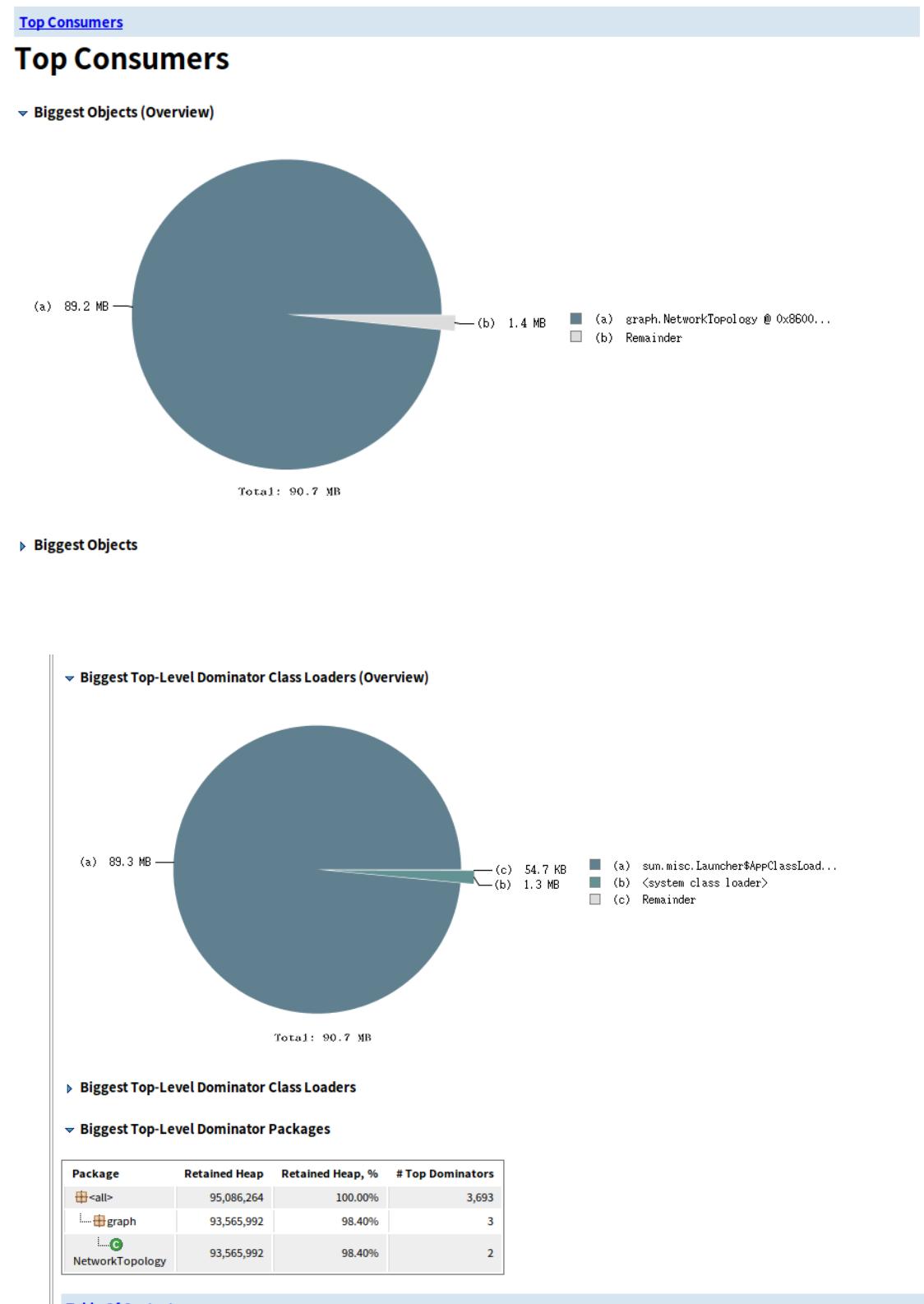
象，如果直接支配对象中出现了不该有的对象，说明发生了内存泄漏。可以

通过顶部的<Regex>来加入过滤条件。

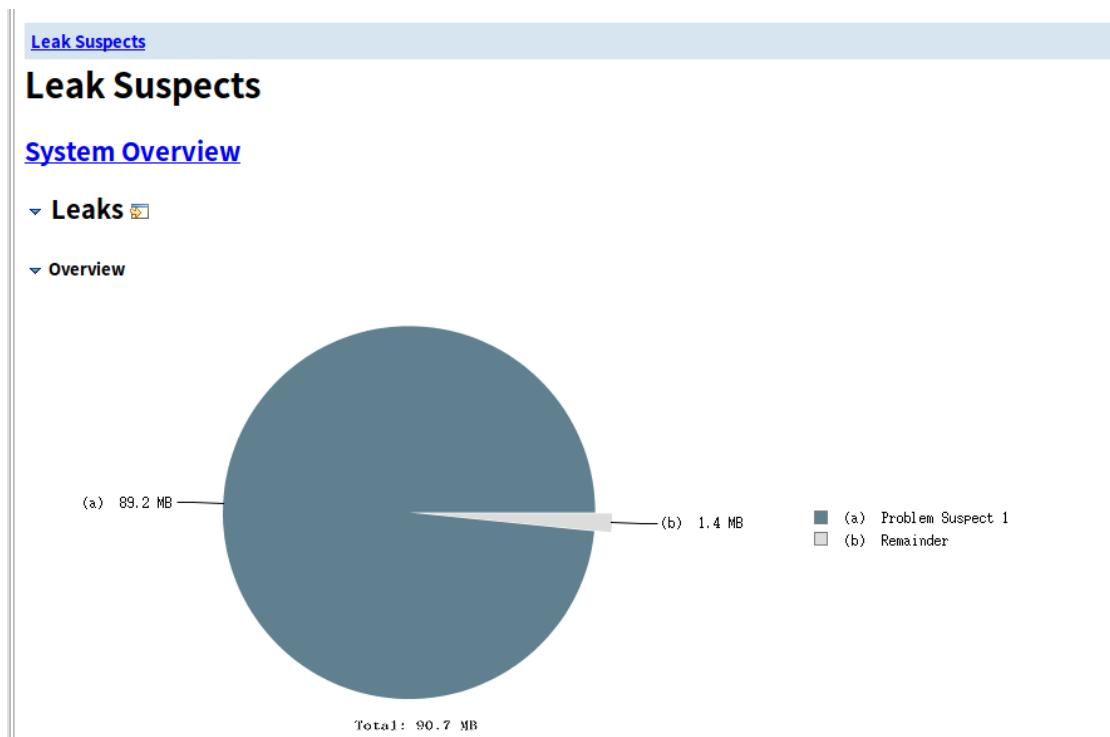
4. Top Consumers

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

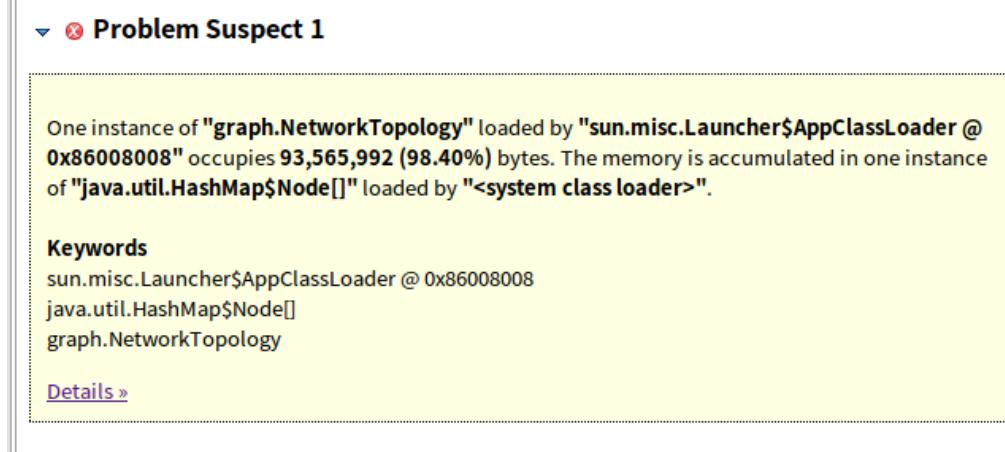
classLoader 是哪些。



5. Leak Suspects



其中，深色区域是被怀疑的地方，怀疑有内存泄漏，具体分析信息如下：



图中深颜色的区域被怀疑有内存泄漏，而下面是内存泄漏的猜想，而通过点击每个内存泄漏猜想的 details 可以看到更深入的分析清理情况。

▼ Shortest Paths To the Accumulation Point ☰

Class Name	Shallow Heap	Retained Heap
java.util.HashMap\$Node@1048576 @ 0x8b631600	4,194,320	92,998,336
└ table java.util.HashMap @ 0x86173220	48	92,998,384
└ map java.util.HashSet @ 0x86173210	16	92,998,400
└ allEdges graph.NetworkTopology @ 0x86004d78	24	93,565,992
└ <Java Local> java.lang.Thread @ 0x86030888 main Thread	120	14,176
└ graph class factory.NetworkTopologyFactory @ 0x86004b68 »	16	264
└ Σ Total: 2 entries		

▼ Accumulated Objects in Dominator Tree ☰

Class Name	Shallow Heap	Retained Heap	Percentage
graph.NetworkTopology @ 0x86004d78	24	93,565,992	98.40%
└ table java.util.HashSet @ 0x86173210	16	92,998,400	97.80%
└ map java.util.HashMap @ 0x86173220	48	92,998,384	97.80%
└ java.util.HashMap\$Node@1048576 @ 0x8b631600	4,194,320	92,998,336	97.80%
└ java.util.HashMap\$TreeNode @ 0x8b377c00	56	1,736	0.00%
└ java.util.HashMap\$Node @ 0x86f63bb8	32	1,344	0.00%
└ java.util.HashMap\$Node @ 0x87322c00	32	1,344	0.00%
└ java.util.HashMap\$Node @ 0x86bba660	32	1,344	0.00%
└ java.util.HashMap\$Node @ 0x87e82f40	32	1,344	0.00%
└ java.util.HashMap\$Node @ 0x8655df48	32	1,120	0.00%
└ java.util.HashMap\$Node @ 0x867ae580	32	1,120	0.00%
└ java.util.HashMap\$Node @ 0x86889be0	32	1,120	0.00%
└ java.util.HashMap\$Node @ 0x86aad248	32	1,120	0.00%
└ java.util.HashMap\$Node @ 0x862b8808	32	1,120	0.00%
└ java.util.HashMap\$Node @ 0x868d0928	32	1,120	0.00%
└ java.util.HashMap\$Node @ 0x86d91118	32	1,120	0.00%
└ java.util.HashMap\$Node @ 0x8798e598	32	1,120	0.00%
└ java.util.HashMap\$Node @ 0x873c69e8	32	1,120	0.00%
└ java.util.HashMap\$Node @ 0x8827e0c8	32	1,120	0.00%

▼ Accumulated Objects by Class in Dominator Tree

Label	Number of Objects	Used Heap Size	Retained Heap Size
java.util.HashMap\$Node First 10 of 331,375 objects	331,375	10,604,000	88,802,280
java.util.HashMap\$TreeNode All 1 objects	1	56	1,736
Σ Total: 2 entries	331,376	10,604,056	88,804,016

▼ All Accumulated Objects by Class

Class Name	Objects	Shallow Heap
java.lang.Object[] First 10 of 396,418 objects	396,418	22,199,408
java.util.ArrayList First 10 of 792,836 objects	792,836	19,028,064
char[] First 10 of 396,418 objects	396,418	12,691,592
edge.NetworkConnection First 10 of 396,418 objects	396,418	12,685,376
java.util.HashMap\$Node First 10 of 396,411 objects	396,411	12,685,152
java.lang.String First 10 of 396,418 objects	396,418	9,514,032
java.util.HashMap\$Node[] All 1 objects	1	4,194,320
java.util.HashMap\$TreeNode All 7 objects	7	392
Σ Total: 8 entries	2,774,927	92,998,336

这个地方提示内存泄漏的原因是，在读取完文件后，所建的图的信息已经保存在了堆中，还没有进行其他的操作。根据 Details 中的信息，除了 NetworkConnection 和 String 类型(char)的变量，并没有其他类型的数据。在 Dominator Tree 中也只是看到了 NetworkConnection 的对象。所以，在读取文件的过程中，并不存在内存泄漏。

3.5.3 发现热点/瓶颈并改进、改进前后的性能对比分析

1. 在进行建图的时候，加边的时候，首先要先去进行遍历存着所有边的 List，去看有没有 label 重复的边，然后去遍历存着所有顶点的 List，去找到边中需要的两个顶点，然后确定这两个顶点之间没有多重边，最后去执行加边操作，也就是将 Edge 对象加入到存着所有边的 List 中，实现了加边，这样操作，会耗费大量时间，经常出现半个小时还没有建完图的情况。

改进： 经过查询资料之后，选择采取 HashSet 来进行存储所有的边和顶点，这

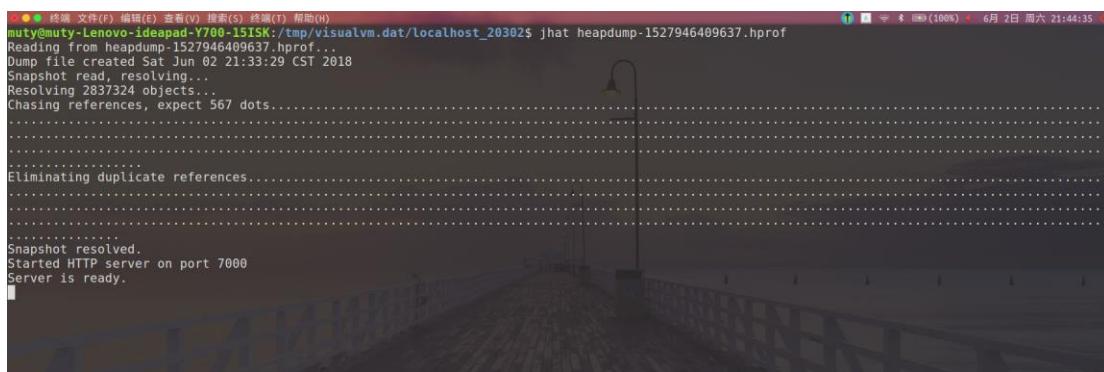
样遍历的时候会相当快，因为 List 和 HashSet 的存储机制不同，所以在进行遍历的时候，List 是有多少元素就一直挨个访问，直到找到我们想要的元素，但是针对我们的图，每次应该遍历都找不到我们想要的元素，才能正常实现加边，所以每次要遍历所有元素，但是 HashSet 会根据 hashCode 值去进行遍历和查找，所以会少遍历很多元素，很节省时间，所以将存储着所有顶点和边的数据结构改为 HashSet，节省了大部分时间。

```
protected final Set<L> allVertices = new HashSet<>();
protected final Set<E> allEdges = new HashSet<>();
```

2. 对于实验要求理解有误，SocialNetwork 中我之前以为在建图的时候，每加入一条边都要进行一次遍历，在我进行建图的时候，发现，竟然一晚上都建不完图，因为到最后计算量实在是太大了。所以再次确定了实验需求之后，进行了修改 addEdge () 方法，最终实现了在一分钟之内读图建图。

3.5.4 jhat 和 OQL 查询内存导出（可选）

首先，用 jhat 命令来解析.prof 文件，有如下的结果



```
muty@muty-Lenovo-ideapad-V700-15ISK:/tmp/visualvm.dat/localhost_2030$ jhat heapdump-1527946409637.hprof
Reading from heapdump-1527946409637.hprof..
Dump file created Sat Jun 02 21:33:29 CST 2018
Snapshot read, resolving...
Resolving 2837324 objects...
Chasing references, expect 567 dots...
...
Eliminating duplicate references...
...
Snapshot resolved.
Started HTTP server on port 7000
Server is ready.
```

然后，这时候可以打开 chrome，在地址栏输入 localhost: 7000，然后就可以看到如下界面：

All Classes (excluding platform)

Package IOStrategy
`class IOStrategy.InputStream [0x86061460]
 class IOStrategy.StreamInputMethod [0x860613f8]`

Package <Arrays>
`class [Lvertex.Vertex; [0x86061188]`

Package application
`class application.NetworkTopologyApp [0x86017ee0]`

Package com.sun.jmx.remote.internal
`class com.sun.jmx.remote.internal.PRef [0x8602ade8]`

Package com.sun.jmx.remote.protocol.iiop
`class com.sun.jmx.remote.protocol.iiop.ProxyStub [0x8602ad10]`

Package edge
`class edge.Edge [0x86061050]
 class edge.NetworkConnection [0x86060f80]
 class edge.UndirectedEdge [0x86060fe8]`

Package factory
`class factory.GraphFactory [0x86061600]
 class factory.NetworkTopologyFactory [0x8600ac00]`

Package graph

点击最下方的 OQL 查询，就可以看到下方搜索框



(1) 查找大于指定长度 n 的 String 对象(假设 n 为 60):

指令: select s from java.lang.String s where s.value.length >= 60

点击其中一个查找结果如下：

(2) 查找特定大小的任意类型对象实例(假设为 100 bytes):

指令如下: select s from instanceof java.lang.Object s where sizeof(s) >= 100

Object Query Language (OQL) query

[All Classes \(excluding.platform\)](#) [OQL Help](#)

```
select s from instanceof java.lang.Object s where sizeof(s) > 100
```

java.util.logging.Logger@0x8526f078
java.util.logging.Logger@0x8523a108
java.util.logging.Logger@0x85239ee8
java.util.logging.Logger@0x852386c8
java.util.logging.Logger@0x85237ed0
java.util.logging.Logger@0x8521ef30
java.util.logging.Logger@0x8521ed68
java.util.logging.Logger@0x8521ea38
java.util.logging.Logger@0x8521e980
java.util.logging.Logger@0x8521e448
java.util.logging.Logger@0x8521d190
java.util.logging.Logger@0x85207fa0
java.util.logging.Logger@0x85207678
java.util.logging.Logger@0x851fd1b0
java.util.logging.Logger@0x851fd110
java.util.logging.Logger@0x851fd070

(3) 查找 Vertex 及每个子类的对象实例的数量和总占用内存的大小

Object Query Language (OQL) query

[All Classes \(excluding.platform\)](#) [OQL Help](#)

```
select count(map(heap.objects('vertex.Vertex')))
```

3000.0

Object Query Language (OQL) query

[All Classes \(excluding.platform\)](#) [OQL Help](#)

```
sum(map(heap.objects('vertex.Vertex'), 'sizeof(it)'))
```

98000.0

3.5.5 jstack 导出 java 程序运行时的调用栈（可选）

(1) 加边

```
"main" #1 prio=5 os_prio=0 tid=0x00007f937c00c800 nid=0x1bf4 runnable [0x00007f9382ce5000]
  ↳ java.lang.Thread.State: RUNNABLE
    ↳ 网络      at java.io.FileInputStream.readBytes(Native Method)
    ↳ 网络      at java.io.FileInputStream.read(FileInputStream.java:255)
    ↳ 127 GB   at java.io.BufferedInputStream.read1(BufferedInputStream.java:284)
    ↳ Software- locked <0x0000000086011ea0> (a java.io.BufferedInputStream)
    ↳ 计算机    at sun.nio.cs.StreamDecoder.readBytes(StreamDecoder.java:284)
    ↳ 计算机    at sun.nio.cs.StreamDecoder.implRead(StreamDecoder.java:326)
    ↳ 文档     at sun.nio.cs.StreamDecoder.read(StreamDecoder.java:178)
    ↳ 系统     - locked <0x0000000086011e88> (a java.io.InputStreamReader)
    ↳ 连接到... at java.io.Reader.read(Reader.java:100)
    ↳ 网络      at java.util.Scanner.readInput(Scanner.java:804)
    ↳ 网络      at java.util.Scanner.findWithinHorizon(Scanner.java:1685)
    ↳ 网络      at java.util.Scanner.nextLine(Scanner.java:1538)
    ↳ 网络      at application.NetworkTopologyApp.main(NetworkTopologyApp.java:35)
```

可以看到，调用了 addEdge () 方法。

(2) 计算顶点中心度

```
"main" #1 prio=5 os_prio=0 tid=0x00007fca6000c800 nid=0x285c runnable [0x00007fca68fab000]
  ↳ java.lang.Thread.State: RUNNABLE
    ↳ Clos      at java.util.Collections$UnmodifiableCollection$1.hasNext(Collections.java:1041)
    ↳ Degra    at edu.uci.ics.jung.algorithms.scoring.BetweennessCentrality.computeBetweenness(BetweennessCentrality.java:122)
    ↳ Eccen    at edu.uci.ics.jung.algorithms.scoring.BetweennessCentrality.<init>(BetweennessCentrality.java:52)
    ↳ Graph    at helper.GraphMetrics.betweennessCentrality(GraphMetrics.java:63)
    ↳ Graph    at helper.BetweennessCentralityStrategy.centralityStrategy(BetweennessCentralityStrategy.java:9)
    ↳ InDegre  at helper.CentralityStrategyHelper.parseAndExecute(CentralityStrategyHelper.java:16)
    ↳ OutDegr  at helper.CentralityContext.setCentralityStrategy(CentralityContext.java:9)
    ↳ Pars     at helper.CentralityContext.setCentralityStrategy(CentralityContext.java:9)
    ↳ IOSTrea  at application.NetworkTopologyApp.main(NetworkTopologyApp.java:63)
```

可以看到，调用了 helper 包中的 centrality 相关方法。

(3) 计算图的中心度

```
"main" #1 prio=5 os_prio=0 tid=0x00007fca6000c800 nid=0x285c runnable [0x00007fca68fab000]
  ↳ java.lang.Thread.State: RUNNABLE
    ↳ Degre    at java.util.HashMap.hash(HashMap.java:339)
    ↳ Eccen    at java.util.HashMap.put(HashMap.java:612)
    ↳ Graph    at java.util.HashSet.add(HashSet.java:220)
    ↳ Graph    at edge.UndirectedEdge.vertices(UndirectedEdge.java:80)
    ↳ InDegre  at helper.GraphMetrics.degreeCentrality(GraphMetrics.java:118)
    ↳ OutDegr  at helper.GraphMetrics.degreeCentrality(GraphMetrics.java:129)
    ↳ Pars     at helper.GraphMetrics.degreeCentrality(GraphMetrics.java:129)
    ↳ IOSTrea  at application.NetworkTopologyApp.main(NetworkTopologyApp.java:77)
```

可以看到，调用了 helper 包中的 centrality 相关方法。

(4) 删除符合 regex 条件的边

```
[main" #1 prio=5 os_prio=0 tid=0x0000000002cae000 nid=0x4c84 runnable [0x00000000045ae000]
  java.lang.Thread.State: RUNNABLE
    at java.util.HashMap.hash(Unknown Source)
    at java.util.HashMap.put(Unknown Source)
    at java.util.HashSet.add(Unknown Source)
    at edge.DirectedEdge.vertices(DirectedEdge.java:99)
    at edge.DirectedEdge.equals(DirectedEdge.java:137)
    at edge.WordNeighborhood.equals(WordNeighborhood.java:58)
    at java.util.ArrayList.remove(Unknown Source)
    at graph.ConcreteGraph.removeEdge(ConcreteGraph.java:274)
    at graph.ConcreteGraph.removeEdge(ConcreteGraph.java:1)
    at helper.ParseCommandHelper.deleteEdge(ParseCommandHelper.java:909)
    at helper.ParseCommandHelper.parseAndExecuteCommand(ParseCommandHelper.java:151)
    at application.GraphPoetApp.main(GraphPoetApp.java:59)
```

Main 方法首先调用指令解析方法 parseAndExecuteCommand，由于这条指令当初实现时，将解析与实现放在了一起，所以在解析完，会直接调用 ConcreteGraph.edges()方法来获取边的集合，然后查找符合 label 的边，进行修改。

4 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 实验过程中遇到的困难与解决途径

在本次实验中遇到的最大的困难，应该是读取文件并建图时，消耗的时间非常巨大，在一开始没有修改以前代码的情况下，出现过几个小时建不完图的情况，找到原因是因为，每次向图中添加一天新边，都需要遍历以前添加过的边，来判断一些异常情况，后来的修改办法有两个，一是直接把该部分异常处理直接删除了，后来采用第二种修改办法，用 HashSet 来减少遍历边或点的集合时大量的时间消耗。

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。