

哈尔滨工业大学计算机科学与技术学院

2017 年秋季学期《软件工程》

Lab 4: 代码评审与程序性能优化

姓名	学号	联系方式
钟宇宏	1151200119	nanua.zqz@gmail.com/18846832803
王巍	1150340114	1450648595@qq.com/15546631826

目 录

第 1 章 实验要求	1
第 2 章 在 IntelliJ 中配置代码审查与分析工具	2
2.1 Checkstyle	2
2.2 PMD	2
2.3 FindBugs	2
2.4 VisualVM	5
第 3 章 本次实验所评审的代码	7
第 4 章 代码 review 记录	8
第 5 章 Checkstyle 所发现的代码问题清单及原因分析.....	9
第 6 章 PMD 所发现的代码问题清单及原因分析	11
第 7 章 FindBugs 所发现的代码问题清单及原因分析	13
第 8 章 VisualVM 性能分析结果	14
8.1 执行时间的统计结果与原因分析	14
8.1.1 生成与展示有向图	14
8.1.2 查询桥接词	15
8.1.3 根据桥接词生成新文本.....	16
8.1.4 计算两个单词之间的最短路径.....	17
8.1.5 随机游走.....	18
8.2 内存占用的统计结果与原因分析	19
8.2.1 生成与展示有向图	19
8.2.2 查询桥接词	22
8.2.3 根据桥接词生成新文本.....	25
8.2.4 计算两个单词之间的最短路径.....	25
8.2.5 随机游走.....	25
8.3 代码改进之后的执行时间统计结果.....	25
8.3.1 随机游走.....	25
8.4 内存占用的统计结果与原因分析	26
8.4.1 生成与展示有向图	26
8.4.2 查询桥接词	29

8.4.3 根据桥接词生成新文本.....	32
8.4.4 计算两个单词之间的最短路径.....	32
8.4.5 随机游走.....	37
8.5 代码改进之后的执行时间统计结果.....	40
8.6 代码改进之后的内存占用统计结果.....	40
第 9 章 利用 Git/GitHub 进行协作的过程	46
9.1 第一部分	46
9.2 第二部分	46
第 10 章 评述	49
10.1 对代码规范方面的评述	49
10.2 对代码性能方面的评述	49
第 11 章 计划与实际进度	50
第 12 章 小结	52

第 1 章 实验要求

本次实验的要求为对 Lab1 所完成的代码进行代码的评审以及性能分析，并从性能角度对代码进行优化。其中，代码的评审包括静态分析以及动态分析两个方面，并且在实验中逐一使用 Checkstyle，FindBugs，PMD，VisualVM 四个工具对代码进行评审以及性能分析。

在本次实验中，采用 Lab1 的分组方式（即两人一组），并随机分配另一组作为本组的评审和分析对象，并要求实验期间不能与原作者进行沟通。

第 2 章 在 IntelliJ 中配置代码审查与分析工具

在本章中，将通过文字以及图片来对在 IntelliJ 中配置 Checkstyle, PMD, FindBugs, VisualVM 四种工具的步骤逐一进行叙述。

2.1 Checkstyle

在本节中，将描述在 IntelliJ 中安装以及配置 Checkstyle 的方法。

首先，在 IntelliJ 中打开项目，然后在顶端的 File 选项栏中选择 Setting 选项，并在弹出的窗口中的左侧边栏中选择 Plugins 选项卡。然后，在输入栏中输入“Checkstyle”，如图2-1所示。

然后，点击其中的 Search in repositories，在弹出的窗口中点击 Install，如图2-2所示。等待安装完成后，由此即完成了 IntelliJ 上 Checkstyle 的安装。

在安装完成后，我们在 Setting 中左边栏选择 Other Settings, Checkstyle。在出现的页面中的 Configuration File 选项中选择默认的 Sun Checks 作为风格检查的标准，如图2-3所示，然后点击 OK 进行确定。由此，即完成了 Checkstyle 的配置。

2.2 PMD

在本节中，将描述在 IntelliJ 中安装以及配置 PMD 的方法。

与 Checkstyle 的安装方式基本一致，首先我们在 Setting 中的 Plugins 选项卡内输入“PMD”，点击 Search in repositories，在出现的窗口中选择 PMDPlugin，点击 Install 进行安装，等待安装完成。安装完后如图2-4所示。

由于本次实验中我们将直接使用 PMD 默认的配置，因此不需要进行额外的配置。

2.3 FindBugs

在本节中，将描述在 IntelliJ 中安装 FindBugs 的方法。

与 Checkstyle 的安装方式基本一致，首先我们在 Setting 中的 Plugins 选项卡内输入“FindBugs”，点击 Search in repositories，在出现的窗口中选择 FindBugs-IDEA，点击 Install 进行安装，等待安装完成。安装完后如图2-5所示。

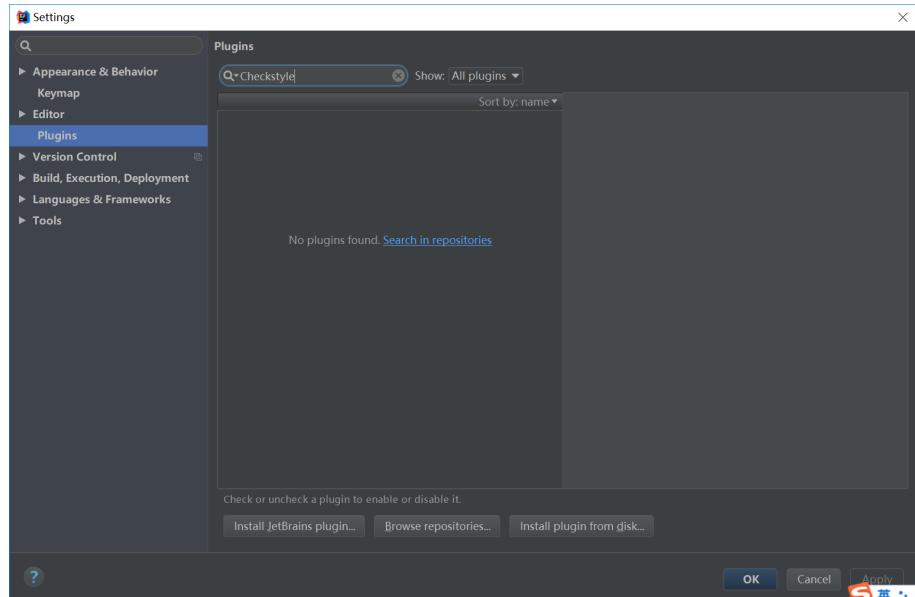


图 2-1 Plugins 选项卡

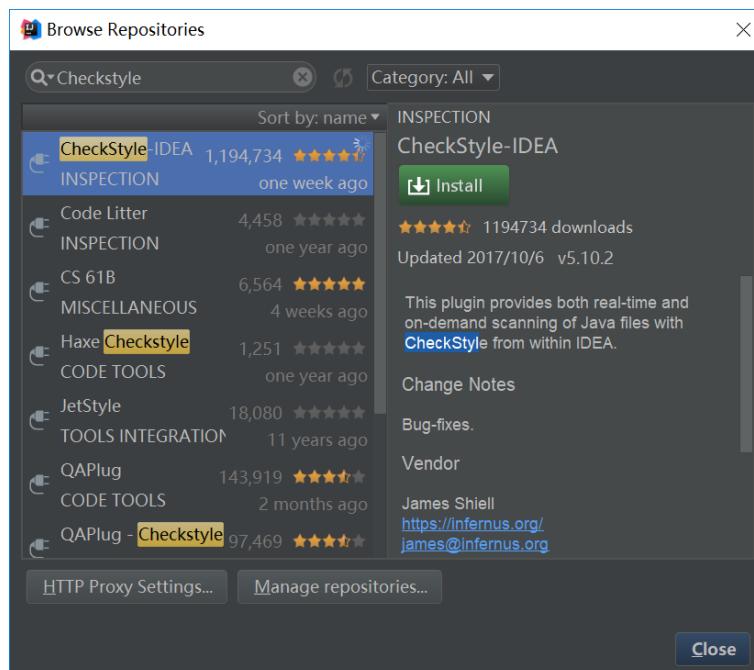


图 2-2 Browse Repositories 界面

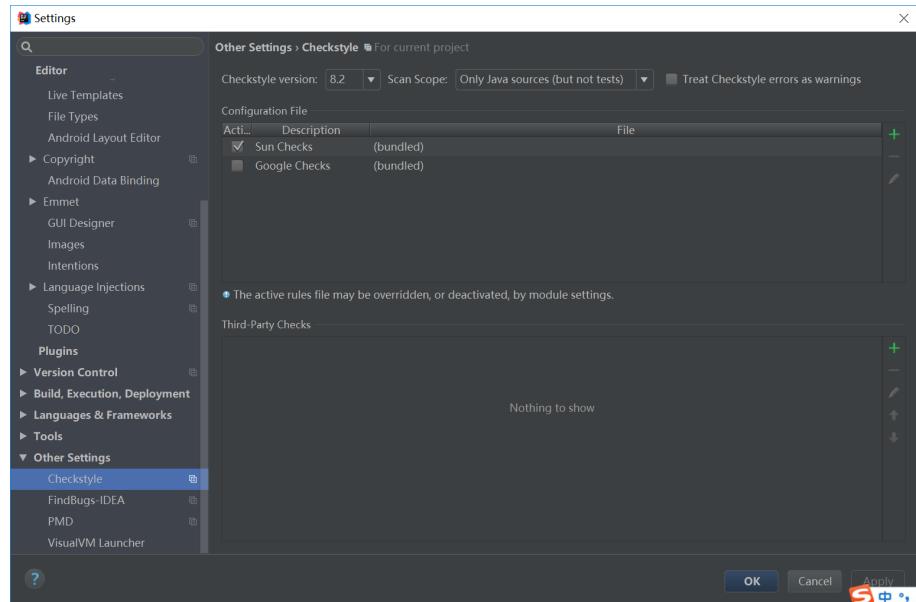


图 2-3 设置检查标准

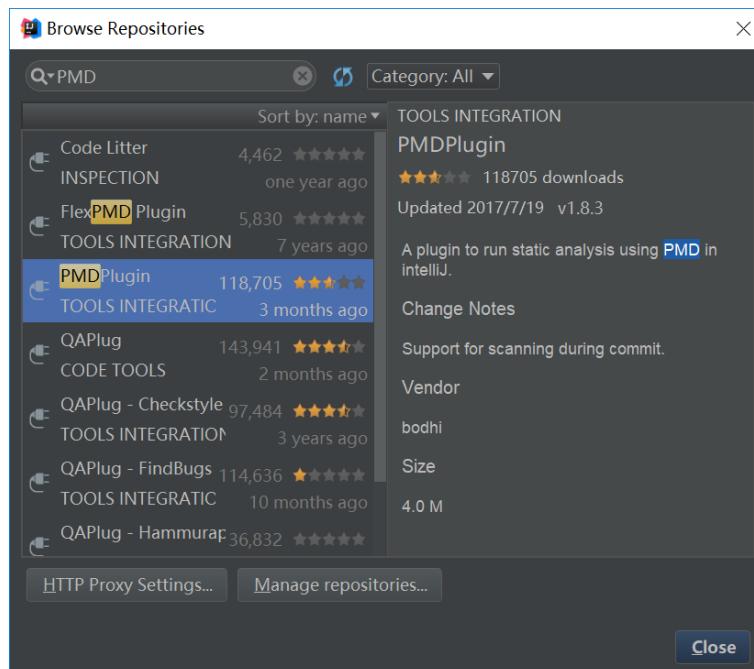


图 2-4 安装 PMD

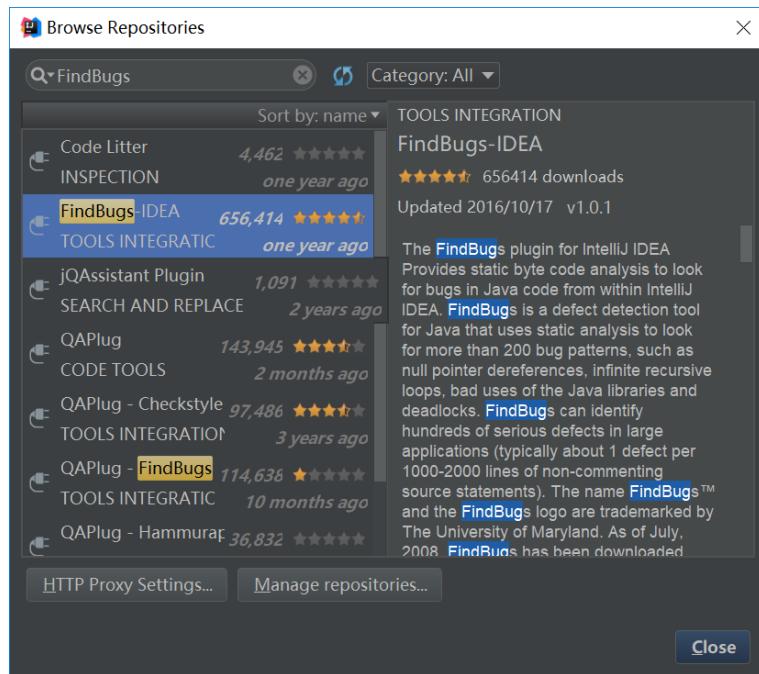


图 2-5 安装 FindBugs

2.4 VisualVM

在本节中，将描述在 IntelliJ 中安装 VisualVM 的方法。

由于 JDK 中自带 VisualVM 软件，因此不需要额外安装 VisualVM。不过，为了能够在 IntelliJ 中方便地使用 VisualVM，我们选择安装 VisualVM Launcher 插件。该插件的安装过程与 Checkstyle 的安装方式基本一致，首先我们在 Setting 中的 Plugins 选项卡内输入“VisualVM”，点击 Search in repositories，在出现的窗口中选择 VisualVM Launcher，点击 Install 进行安装，等待安装完成。安装完后如图2-6所示。

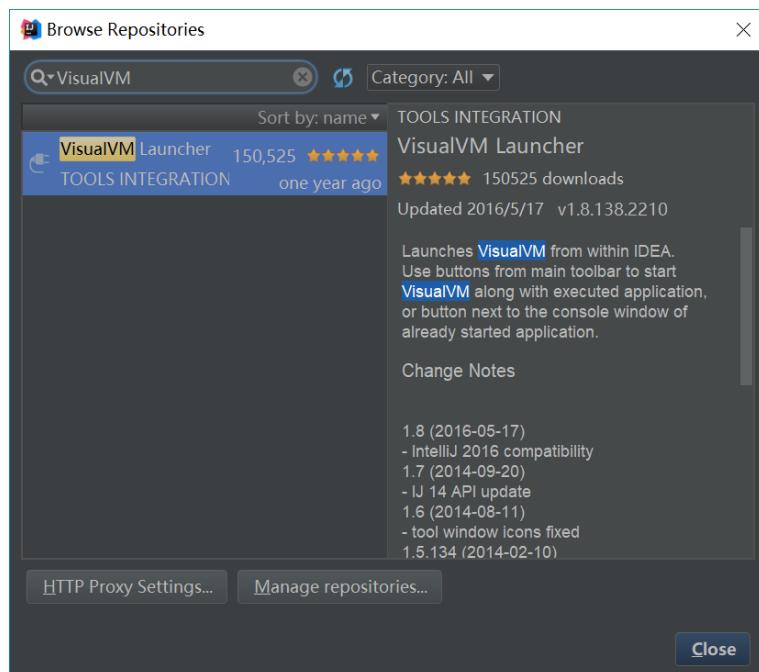


图 2-6 VisualVM Launcher 安装

第 3 章 本次实验所评审的代码

姓名：张冠华

学号：1150310323

Github 地址：<https://github.com/miuws>

姓名：王珊

学号：1150310302

Github 地址：<https://github.com/arthua196>

miuws second commit		Latest commit 847e16d 7 days ago
.idea	first commit	8 days ago
config	first commit	8 days ago
out/production/JavaLab1	first commit	8 days ago
src	first commit	8 days ago
temp	first commit	8 days ago
DotGraph.jpg	first commit	8 days ago
Graph.java	second commit	7 days ago
JavaLab1.iml	first commit	8 days ago
dotsource.dot	first commit	8 days ago
test.txt	first commit	8 days ago

项目清单

miuws first commit		Latest commit c65c103 8 days ago
..		
Edge.java		
Graph.java		
GraphViz.java		
MainPage.java		
TextMaker.java		

源码清单

第 4 章 代码 review 记录

问题描述	类型	所在代码行号	修改方式
在点击退出按钮后 GUI 界面关闭 但程序仍在运行	程序退出控制	MainPage.java : 467	增加代码 setDefaultCloseOperation (WindowConstants. EXIT_ON_CLOSE)
没有对 Graphviz 关键字特别处理 展示图失败	外部方法调用	TextMaker.java : 168	使用引号 包围所有关键词
对文本逐字符处理 效率极低	运行效率	TextMaker.java : 53	使用一行正则表达式 完成文本处理
重新打开对话框 以前输入的文字混乱	GUI 界面管理	MainPage.java : 322	每次打开对话框 初始化新的对话框
使用 HIDE_ON_CLOSE 关闭对话框	GUI 资源释放	MainPage.java : 365	改为 DISPOSE_ON_CLOSE

第 5 章 Checkstyle 所发现的代码问题清单及原因分析

(使用 Sun Checks 规则)

编号	问题描述	类型	所在代码行号	修改策略
1	类缺少 JavaDoc	文档缺失	Edge.java :3	补充 Edge 类文档
2	大括号应位于类、方法定义同一行	类、方法定义格式	Edge.java :4	将大括号放在类、方法定义同一行
3	,	空格格式	Edge.java :5	补充空格
4	应避免在字表达式中赋值	赋值格式	Edge.java :10	将赋值拆分成多行
5	参数 xxx 应定义为 final 的	只读参数、变量用法	Edge.java :12	为变量声明增加 final 关键字
6	{' 后应换行	避免使用一行定义方法	Edge.java :33	为方法定义规范换行
7	if/else 结构必须使用大括号	控制结构可读性	Edge.java :36	为 if 结构添加大括号
8	数组大括号位置错误	数组定义规范	Graph.java :121	数组中括号移至变量类型后
9	xxx 是一个魔术数字	直接常数	Graph.java :135	将数字赋值给常量
10	不应以.* 形式导入 xxx	import 规范	MainPage.java :1	GUI 使用了大部分包中的类，不修改
11	xxx 应为 private 并配置访问方法	访问权限规范	MainPage.java :13	访问权限改为 private 添加访问方法
12	名称必须匹配表达式 xxx	命名规范	MainPage.java :22	refactor 修改命名
13	本行字符数 xxx 最多 xxx	行长度规范	MainPage.java :80	拆成多行

(使用 Google 规则集的不同之处)

编号	问题描述	类型	所在代码行号	修改策略
1	缩进空格应为两个	缩进格式	TextMaker.java :13	(和规则集有关，不修改)
2	包名导入顺序错误	import 顺序错误	MainPage.java :3	更改包导入顺序
3	注释中的空行 应该在 <p> 标签后	缩进格式	Graphviz.java :23	(和规则集有关，不修改)
4	其它 if,for 等 缩进空格数	缩进格式	Graph.java :35	(和规则集有关，不修改)

小结：

Sun 和 Google 规则集大致相同。它们主要对这些问题进行检查：

- 1、缩进
- 2、有利于可读性的空格
- 3、代码块是否正确地被大括号包围

不同的之处有：

- 1、Google 对缩进的要求是 Sun 规则集的一般
- 2、Google 规则集对 JavaDoc 的格式检查更严格
- 3、Google 规则集检查包名的导入顺序
- 4、Sun 规则集会检查部分内部逻辑

第 6 章 PMD 所发现的代码问题清单及原因分析

优先级按照 <https://pmd.github.io/pmd-5.8.1/pmd-java/rules/java> 文档定义

优先级	问题描述	违反的规则集合	代码行号	修改策略
3	变量、参数名过短 不易于理解	naming	Edge.java : 5	使用 refactor 更改命名
3	缺少包定义	naming	Edge.java : 3	为类编写文档
4	布尔型返回值 方法命名错误	naming	Edge.java : 33	用 is、has、can 等 命名此类方法
3	没有'{' 的 if 语句	braces	Edge.java : 46	为 if 结构增加'{'
3	类中方法过多	codesize	TextMaker.java : 10	方法过多的类 应该重构
3	控制流程语句 过于复杂	codesize	MainPage.java : 69	重构以较少控制分支
2	缺少注释	comments	Edge.java : 3	添加注释
3	多个 return 的方法	controversial	Edge.java : 46	将返回值复制给变量 使用一个 return 返回
3	硬编码字面量	controversial	Edge.java : 46	赋予有意义的常量名
3	发现 final 局部变量	controversial	Graph.java : 51	将 final 局部变量 写成类的域
3	在操作数中赋值	controversial	Graphviz.java : 317	拆成多行，单独赋值
3	应显式指定访问权限	controversial	MainPage.java : 13	显示指定访问权限
3	使用具体实现的类 限制了功能的实现	coupling	Graph.java : 13	使用接口名 定义对象引用

优先级	问题描述	违反的规则集合	代码行号	修改策略
3	只在初始化时赋值的变量应声明为 final	design	Edge.java : 4	在域中初始化并声明为 final
3	使用了 size=0 判断集合是否为空	design	Graph.java : 155	使用 isEmpty 方法替代
3	发现 God Class (过于复杂的类)	design	Graph.java : 1	重构类
3	使用了'==' 比较对象	design	MainPage.java : 74	替换为 equals 方法
2	发现空的 catch 代码块	empty	TextMaker : 136	抛出 RuntimeException 或处理异常
3	发现调用 System.Exit	j2ee	MainPage.java : 216	检查逻辑 System.Exit 使用无误
3	发现可以声明为 final 的参数	optimizations	Edge.java : 14	未在方法中修改的参数声明为 final
3	发现重复的字面量	string	Graph.java : 249	将字面量值赋予常量
3	发现未使用的参数	unusedcode	Graph.java : 85	删除参数

小结：

PMD 的各个规则集都要它们自己的要检查的规则

如 naming 规则集检查命名问题， design 规则集检查代码的结构逻辑

要根据规则集关于其规则的描述，决定是否应该使用规则集

第 7 章 FindBugs 所发现的代码问题清单及原因分析

问题描述	类型	所在代码行号	修改策略
不是所有的代码路径都关闭了流	IO 资源释放	Graphviz.java : 315	try with resource 管理资源
try catch 语句 catch 未处理	异常处理	Graphviz.java : 48	直接抛出异常或在方法内处理异常
使用 \n 作为换行符 符合平台特性	格式化符号	Graph.java : 249	将 \n 替换为%n
发现调用 System.exit	System.exit 用法错误	Graph.java : 216	检查逻辑 System.Exit 使用无误
BufferedWriter 未指定编码	对默认编码依赖	Graphviz.java : 254	针对平台 指定编码参数
使用硬编码 引用绝对路径	使用硬编码	TextMaker.java : 268	使用相对路径 并赋值给常量

第 8 章 VisualVM 性能分析结果

在本章中，将利用 VisualVM 工具对项目的执行时间以及内存占用情况进行分析，并根据分析得出的结果对项目代码进行改进。最后将测试改进后的代码的执行时间以及内存占用情况，并与为改进之前的情况进行对比。

8.1 执行时间的统计结果与原因分析

在本节中，将分别分析项目在生成与展示有向图，查询桥接词，根据桥接词生成新文本，计算两个单词之间的最短路径，以及随机游走这五个功能方面的耗时，并对耗时的原因进行分析。

需要说明的是，虽然在 Lab1 的原始要求中“读取并生成有向图”以及“展示有向图”为两个功能，但是在该项目中完成文件的读取以及有向图的生成后即自动展示了有向图，因此在此将原始要求中的两个功能需求合并为“读取并展示有向图”一个功能，并对这一个功能进行分析。

8.1.1 生成与展示有向图

在生成与展示有向图的时间测试中，我们采用 Lab1 验收时的测试数据作为输入进行测试，耗时结果的统计见图8-1。

需要首先特别说明的是，MenuListener.actionPerformed() 的耗时为用户选择所要读取的文件时的耗时，因此不算在程序的运行时间内。

在耗时图像中，GraphViz.get_img_stream() 为调用外部程序 GraphViz

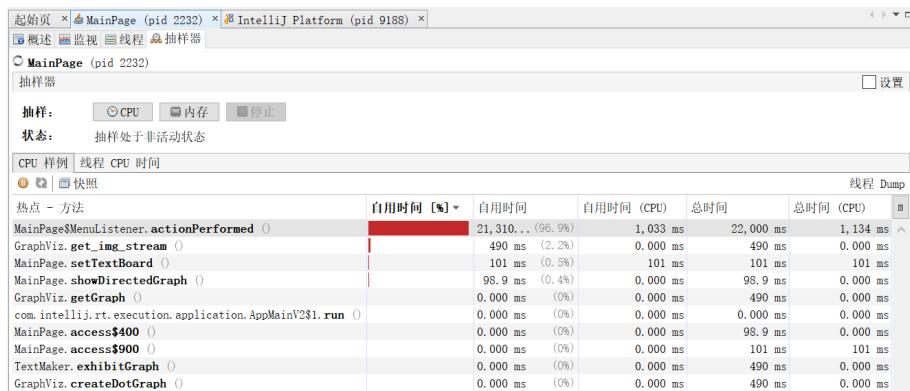


图 8-1 生成与展示有向图耗时

绘制图像，并从 GraphViz 读取其返回的图像的二进制串的函数，GraphViz.createDotGraph() 为接受 GraphViz 源程序作为输入，并最终将生成的图片写入到文件中的函数，TextMaker.exhibitGraph() 为生成程序中图的结构对应的 GraphViz 源程序，并最终将生成的图片写入到文件中的函数， MainPage.setTextBoard() 为配置 GUI 左侧展示文本框的函数，MainPage.showDirectedGraph() 为配置 GUI 中展示有向图部分的 GUI 元素的函数。

由此可见，除去用户操作的时间外，主要占用时间的函数为 GraphViz.get_img_stream(), MainPage.setTextBoard(), 以及 MainPage.showDirectedGraph() 这三个函数。由于 GraphViz.get_img_stream() 为与 GraphViz 进行交互的函数，在不修改 GraphViz 的前提下没有更多优化的余地，并且 MainPage.setTextBoard() 与 MainPage.showDirectedGraph() 为控制 GUI 元素的函数，因此在使用同一 GUI 框架的前提下也无法进行优化，因此在生成与展示有向图的功能中无法通过耗时分析来进行这部分功能的优化。

8.1.2 查询桥接词

在查询桥接词的测试中，我们同样采用 Lab1 验收的测试数据作为输入，即进行查询：

1. 不存在的词的情况：first, second
2. 不存在桥接词的情况：time, by
3. 存在一个桥接词的情况：important, trends
4. 存在多个桥接词的情况：the, of

耗时分析结果见图8-2。

在耗时图像中，GraphViz.get_img_stream() 为调用外部程序 GraphViz 绘制图像，并从 GraphViz 读取其返回的图像的二进制串的函数，MainPage.showDirectedGraph() 为配置 GUI 中展示有向图部分的 GUI 元素的函数，MainPage\$ButtonListener.actionPerformed() 为处理按钮点击事件的函数，TextMaker.exhibitGraph() 为生成程序中图的结构对应的 GraphViz 源程序，并最终将生成的图片写入到文件中的函数，GraphViz.createDotGraph() 为接受 GraphViz 源程序作为输入，并最终将生成的图片写入到文件中的函数，GraphViz.getGraph()

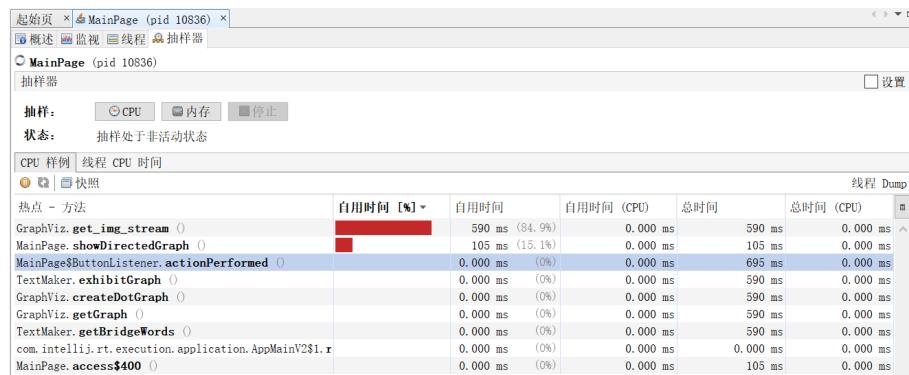


图 8-2 查询桥接词耗时分析

为接受 GraphViz 源程序作为输入，并将生成的图片的二进制返回的函数，TextMaker.getBridgeWords() 为查询有向图中桥接词的函数。

由以上分析可见，在查询桥接词的过程之中，占用时间的主要是 GraphViz.get_img_stream()， MainPage.showDirectedGraph() 这两个函数。与生成并展示有向图时的情况相同，我们没法在不改变 GUI 框架以及 GraphViz 实现原理的情况下对这两个函数进行优化。

8.1.3 根据桥接词生成新文本

根据桥接词生成新文本的测试中，我们同样采用 Lab1 验收的测试数据作为输入，其输入的文本分别为“*In the big time servitization becomes the of the world.*”，以及“*In the big time, servitization one of the important trends-of the IT world.*”。测试得出的耗时图像见图8-3。

在耗时图像中，GraphViz.get_img_stream() 为调用外部程序 GraphViz 绘制图像，并从 GraphViz 读取其返回的图像的二进制串的函数，

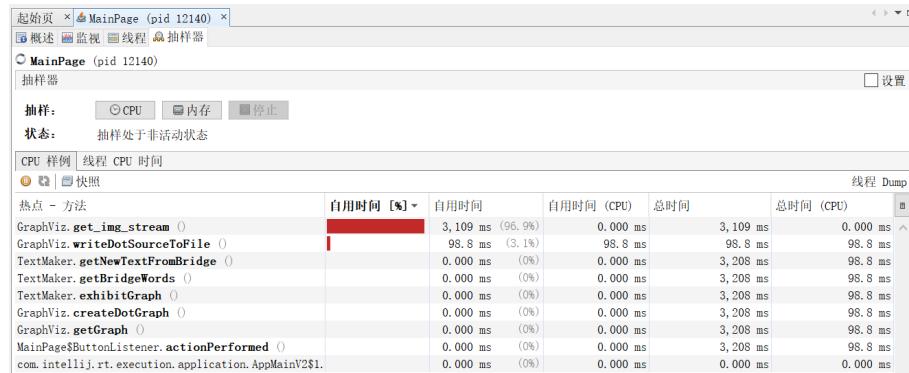


图 8-3 根据桥接词生成新文本耗时分析

GraphViz.writeDotSourceToFile() 为接受 GraphViz 源代码作为输入，将源代码写入 GraphViz 源文件中的函数， MainPage\$ButtonListener.actionPerformed() 为处理按钮点击事件的函数， TextMaker.exhibitGraph() 为生成程序中图的结构对应的 GraphViz 源程序，并最终将生成的图片写入到文件中的函数， GraphViz.createDotGraph() 为接受 GraphViz 源程序作为输入，并最终将生成的图片写入到文件中的函数， GraphViz.getGraph() 为接受 GraphViz 源程序作为输入，并将生成的图片的二进制返回的函数， TextMaker.getBridgeWords() 为查询有向图中桥接词的函数， TextMaker.getTextFromBridge() 为根据输入的文本的桥接词生成新文本的函数。

由以上分析可知，占用时间的主要为 GraphViz.get_img_stream(), GraphViz.writeDotSourceToFile() 这两个函数。与之前分析的情况类似，在不改变通过 GraphViz 来生成图像的前提下，无法进一步从耗时方面提高程序的性能。

8.1.4 计算两个单词之间的最短路径

在计算两个单词之间的最短路径的测试中，我们同样采用 Lab1 验收的测试数据作为输入，即测试：

1. 有一条最短路径的情况： time, word
2. 两个单词不可达的情况： this, study
3. 有多条最短路径的情况： the, word
4. 输入一个单词查询单源最短路径的情况： in

测试得出的耗时图像见图8-4。

在耗时图像中， GraphViz.get_img_stream() 为调用外部程序 GraphViz 绘制图像，并从 GraphViz 读取其返回的图像的二进制串的函数， MainPage.showInformation() 为配置 GUI 中显示结果提示信息部分的 GUI 元素的函数， MainPage.showDirectedGraph() 为配置 GUI 中展示有向图部分的 GUI 元素的函数， TextMaker.exhibitGraph() 为生成程序中图的结构对应的 GraphViz 源程序，并最终将生成的图片写入到文件中的函数， GraphViz.createDotGraph() 为接受 GraphViz 源程序作为输入，并最终将生成的图片写入到文件中的函数， GraphViz.getGraph() 为接受 GraphViz 源程序作为输入，并将生成的图片的二进

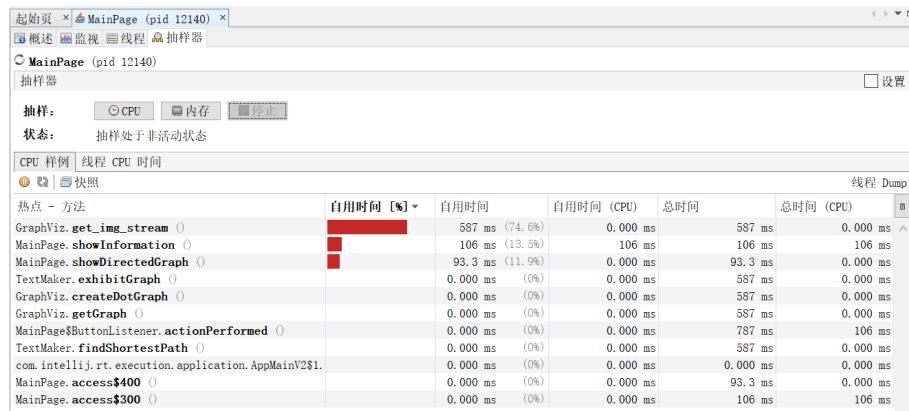


图 8-4 计算两个单词之间的最短路径耗时分析

制返回的函数， MainPage\$ButtonListener.actionPerformed() 为处理按钮点击事件的函数， TextMaker.findShortestPath() 为查询单源最短路径的函数。

由以上分析可知，占用时间主要是 GraphViz.get_img_stream()， MainPage.showInformation()， MainPage.showDirectedGraph() 这三个函数。与之前的情况相同，在不改变 GUI 框架以及使用 GraphViz 进行绘图的前提下无法从耗时方面对计算两个单词之间最短路劲的性能进行优化。

8.1.5 随机游走

在随机游走的测试中不需要输入，其耗时图像见图8-15。

在耗时图像中， GraphViz.get_img_stream() 为调用外部程序 GraphViz 绘制图像，并从 GraphViz 读取其返回的图像的二进制串的函数， MainPage.showDirectedGraph() 为配置 GUI 中展示有向图部分的 GUI 元素的函数， MainPage.setTextBoard() 为配置 GUI 左侧展示文本框的函数，

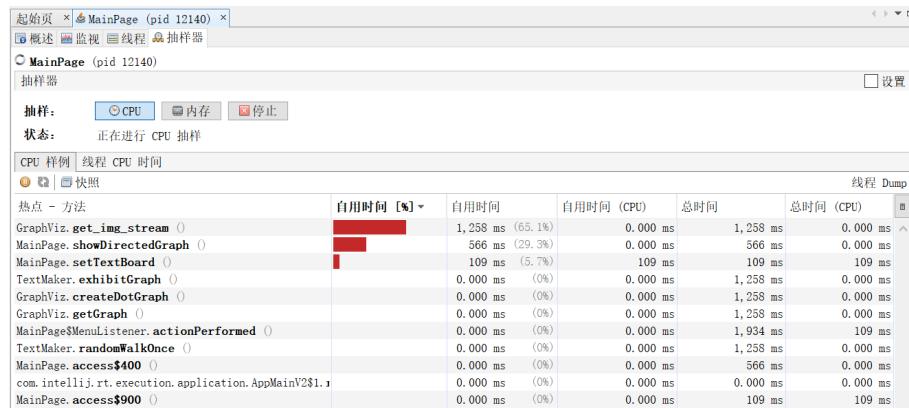


图 8-5 随机游走耗时分析

TextMaker.exhibitGraph() 为生成程序中图的结构对应的 GraphViz 源程序，并最终将生成的图片写入到文件中的函数，GraphViz.createDotGraph() 为接受 GraphViz 源程序作为输入，并最终将生成的图片写入到文件中的函数，GraphViz.getGraph() 为接受 GraphViz 源程序作为输入，并将生成的图片的二进制返回的函数， MainPage\$ButtonListener.actionPerformed() 为处理按钮点击事件的函数，TextMaker.findShortestPath() 为查询单源最短路径的函数，TextMaker.randomWalkOnce() 为随机游走一次的函数。

由以上分析可知，占用时间的主要原因是 GraphViz.get_img_stream()，MainPage.showDirectedGraph()，MainPage.setTextBoard() 三个函数。与之前的情况相同，在不改变 GUI 框架以及使用 GraphViz 进行绘图的前提下无法从耗时方面对计算两个单词之间最短路劲的性能进行优化。

8.2 内存占用的统计结果与原因分析

在本节中，将分别分析项目在生成与展示有向图，查询桥接词，根据桥接词生成新文本，计算两个单词之间的最短路径，以及随机游走这五个功能方面的内存占用情况，并对占用内存的原因进行分析。

8.2.1 生成与展示有向图

在生成与展示有向图部分，我们将主要考查多次读取输入文件时，是否发生了内存泄漏的情况。测试的方法为分别记录读取一次文本，十次文本，二十次文本，三十次文本时的堆的使用情况，并进行比较。其中，每次读取使用相同的文本文件（也即 Lab1 的测试文件），并在每次记录数据前执行垃圾回收操作。由于程序每次读取文本时实际上在更新其内的有向图，并把原始文本以及预处理后的文本累加输出到 GUI 相应的文本框中，因此若无内存泄漏，则可预计 java.lang.String 在堆中的占用增加，其他类的占用应基本保持不变。

第一，二，三，四次记录的数据分别见图8-16，图8-17，图8-18，图8-19。读取三十次文本时记录的数据与读取一次文本时记录的数据的差异比较见图8-20。

由图8-16，图8-17，图8-18，图8-19可见，每次记录数据时主要占用堆空间的类为 char[] 以及 java.lang.String，合起来每次记录数据时都占用了超过 40% 的堆空间。另外，从图8-16到图8-19的变化中可以看出，每次记录数据时占用堆空间较多的类的实例数以及大小并没有发生较大的改变。由读取三十次文本与读取一次文本时堆的比较可见看出，char[] 的实例数增加了 5%，java.lang.String 的实例数增加

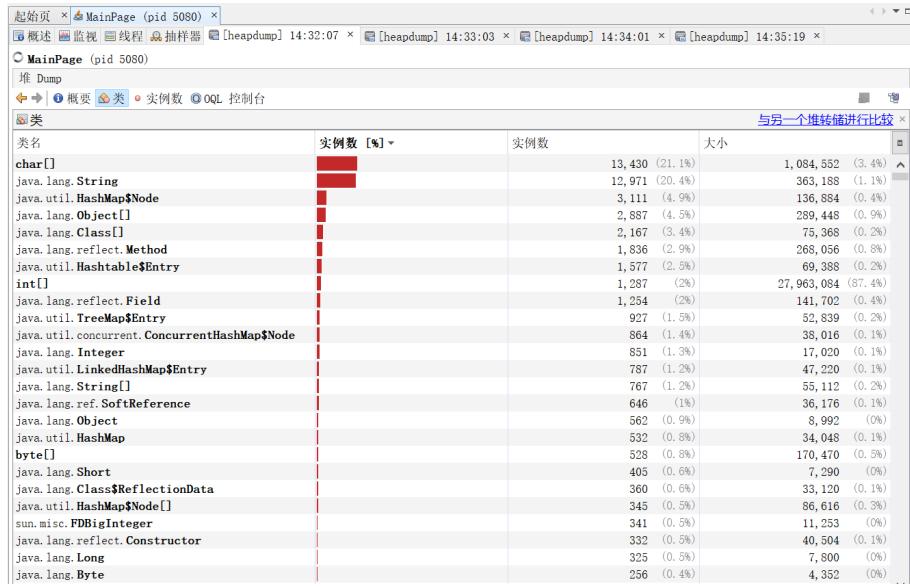


图 8-6 读取一次文本时堆情况

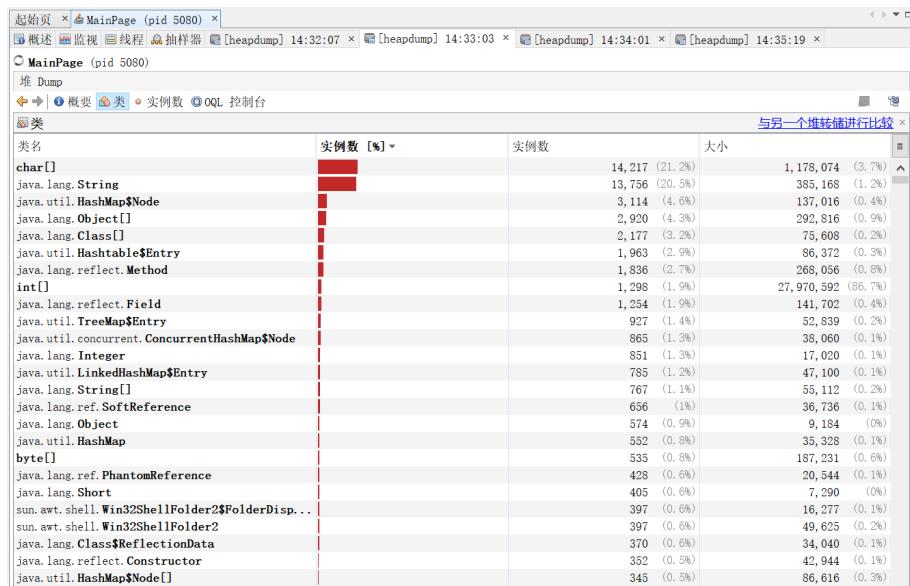


图 8-7 读取十次文本时堆情况

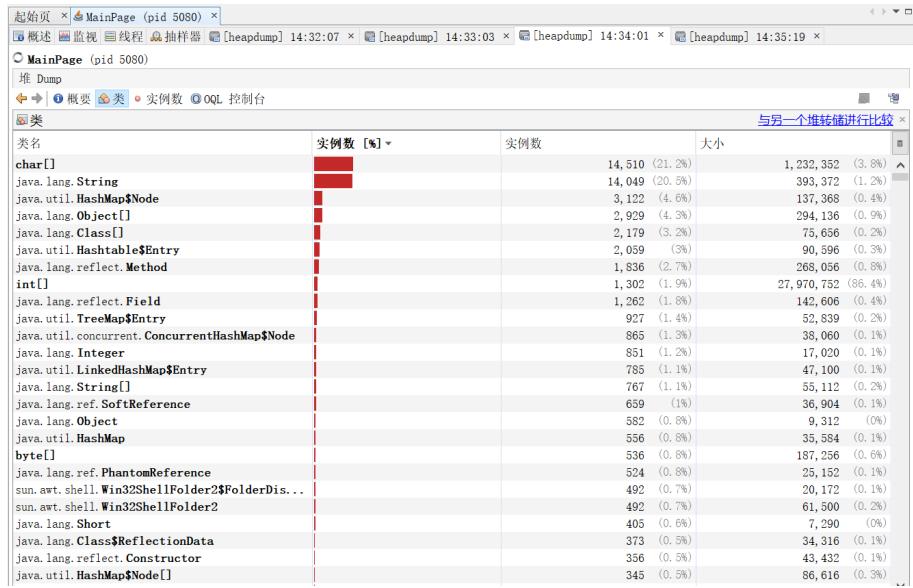


图 8-8 读取二十次文本时堆情况

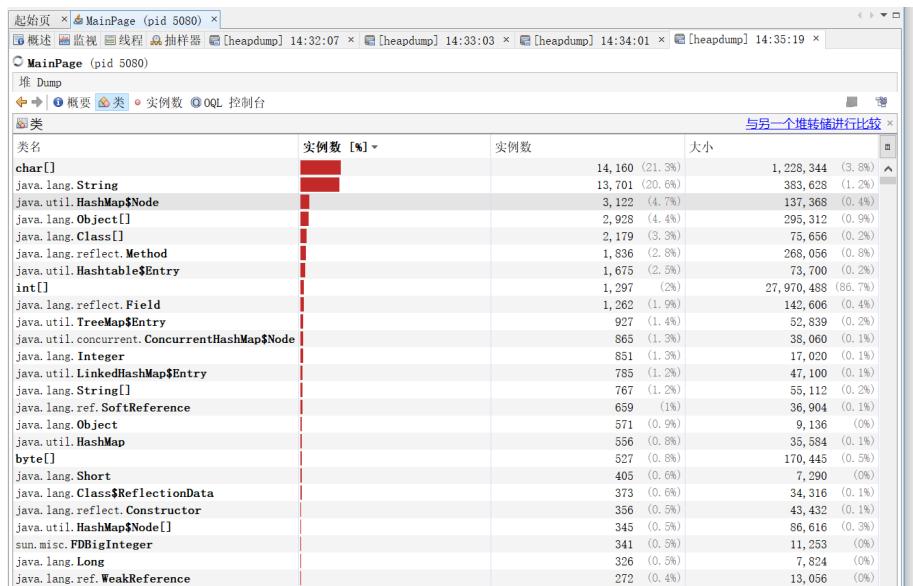


图 8-9 读取三十次文本时堆情况

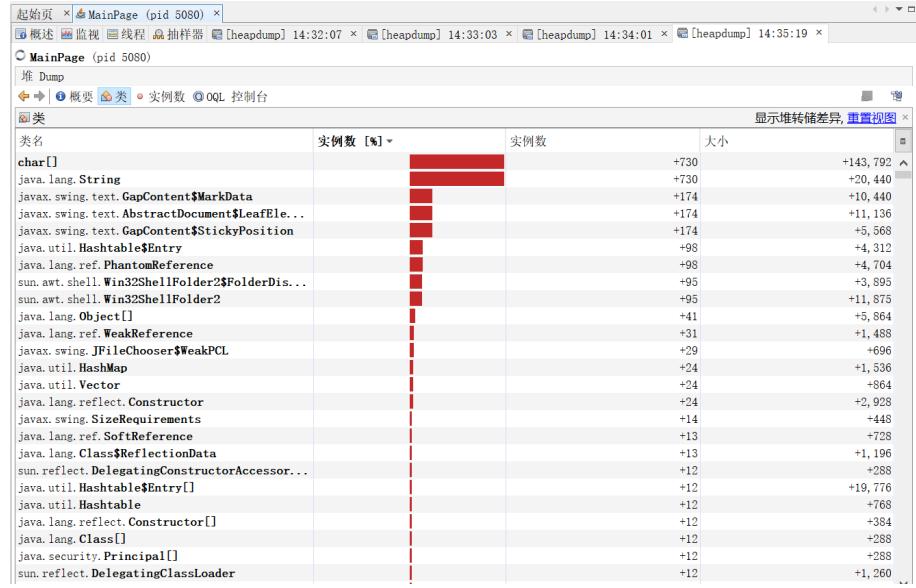


图 8-10 读取三十次文本与读取一次文本比较

了 5%，HashMap\$Node 的实例数增加了 0.3%，即只有 char[] 以及 java.lang.String 两个类的占用发生了小幅度的提升，其他类的堆占用基本保持不变。这与无内存泄漏的预期结果相符合，因此可以推断此过程中未发生内存泄漏。

8.2.2 查询桥接词

在查询桥接词的功能中，我们主要测试多次查询桥接词时是否会出现内存泄漏的情况。测试方法为读入 Lab1 测试数据中“读取并生成有向图”部分的文件数据，然后在此有向图上进行一次，十次，二十次查询有多个桥接词的两个单词 the, of 之间的桥接词，由此分析是否在查询桥接词的过程中发生了内存泄漏。其中，每次记录堆的数据占用情况前都进行垃圾回收。由于程序每次完成桥接词的查询之后通过弹出对话框的方式显示查询到的桥接词，关闭对话框后查询结果即消失，因此若无内存泄漏，则可预计所有类的堆占用情况皆不会发生较大的改变。

第一，二，三次记录的数据分别见图8-21，图8-22，图8-23。查询二十次桥接词时记录的数据与查询一次桥接词时记录的数据的差异比较见图8-24。

由图8-21，图8-22，图8-23，每次记录数据时主要占用堆空间的类同样为 char[] 以及 java.lang.String。同样的，从图8-21到图8-23的变化中可以看出，每次记录数据时占用堆空间较多的类的实例数以及大小并没有发生较大的改变。由查询二十次桥接词与查询一次桥接词时堆的比较可见看出，char[] 的实例数增加了 1%，java.lang.String 的实例数增加了 1%，HashMap\$Node 的实例数增加了 0.3%，基

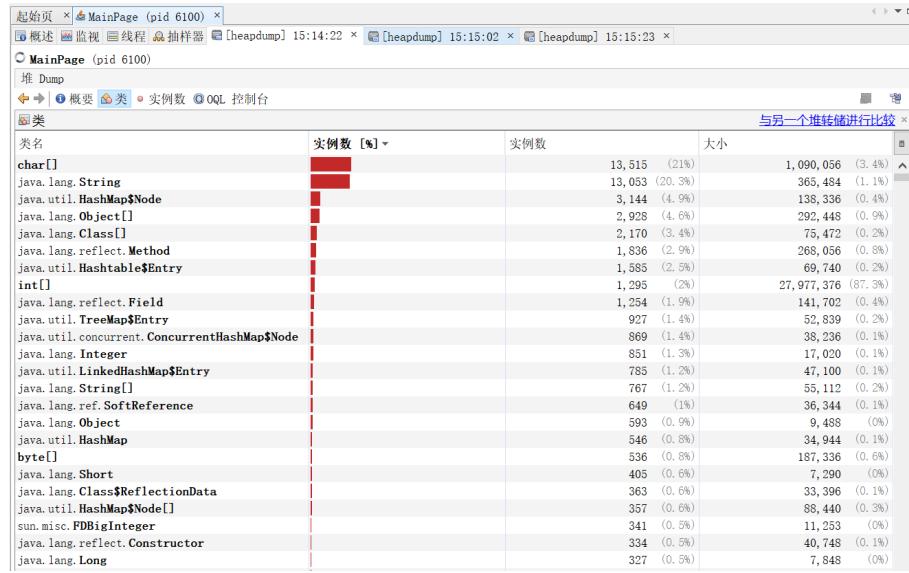


图 8-11 查询一次桥接词时堆情况

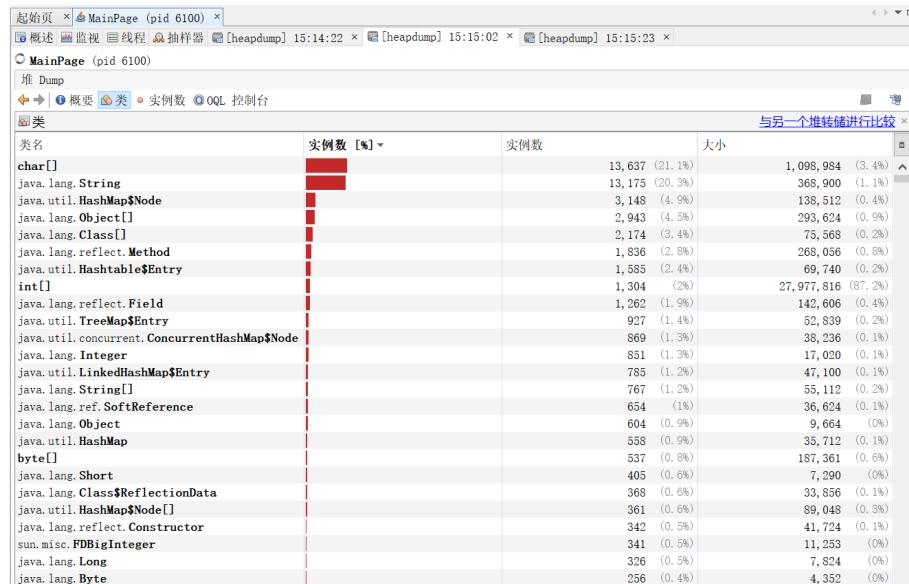


图 8-12 查询十次桥接词时堆情况

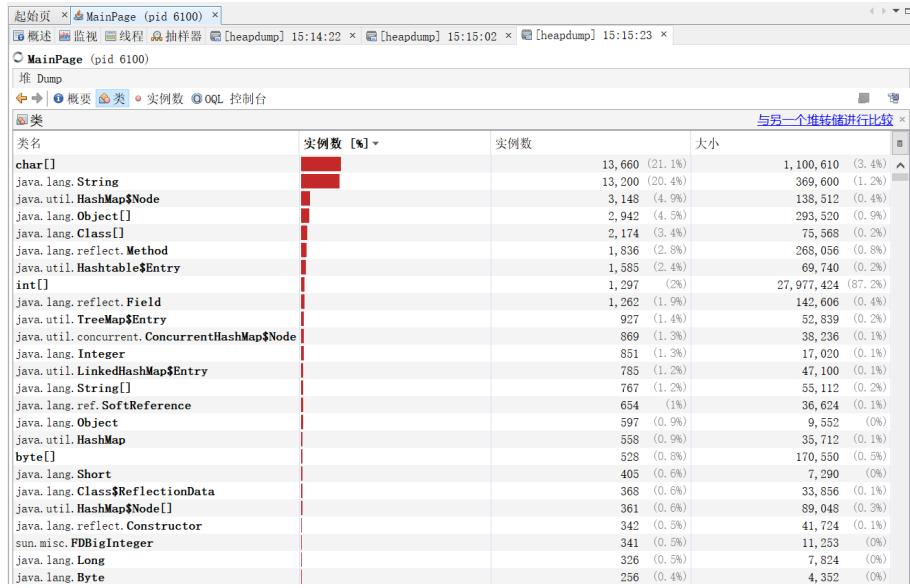


图 8-13 查询二十次桥接词时堆情况

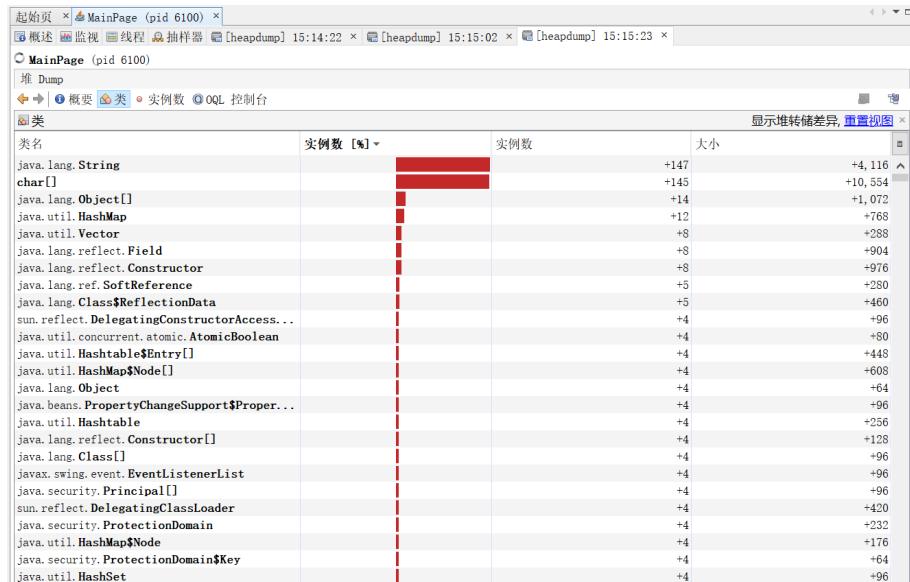


图 8-14 查询一次与查询二十次堆比较

本可视作没有发生变化，与无内存泄漏时的预期基本一致，因此可以推断出在查询桥接词时为发生内存泄漏。

8.2.3 根据桥接词生成新文本

8.2.4 计算两个单词之间的最短路径

8.2.5 随机游走

8.3 代码改进之后的执行时间统计结果

在不改变 GUI 框架以及使用 GraphViz 进行绘图的前提下无法从耗时方面对计算两个单词之间最短路劲的性能进行优化。

8.3.1 随机游走

在随机游走的测试中不需要输入，其耗时图像见图8-15。

在耗时图像中，GraphViz.get_img_stream() 为调用外部程序 GraphViz 绘制图像，并从 GraphViz 读取其返回的图像的二进制串的函数， MainPage.showDirectedGraph() 为配置 GUI 中展示有向图部分的 GUI 元素的函数，MainPage.setTextBoard() 为配置 GUI 左侧展示文本框的函数，TextMaker.exhibitGraph() 为生成程序中图的结构对应的 GraphViz 源程序，并最终将生成的图片写入到文件中的函数，GraphViz.createDotGraph() 为接受 GraphViz 源程序作为输入，并最终将生成的图片写入到文件中的函数，GraphViz.getGraph() 为接受 GraphViz 源程序作为输入，并将生成的图片

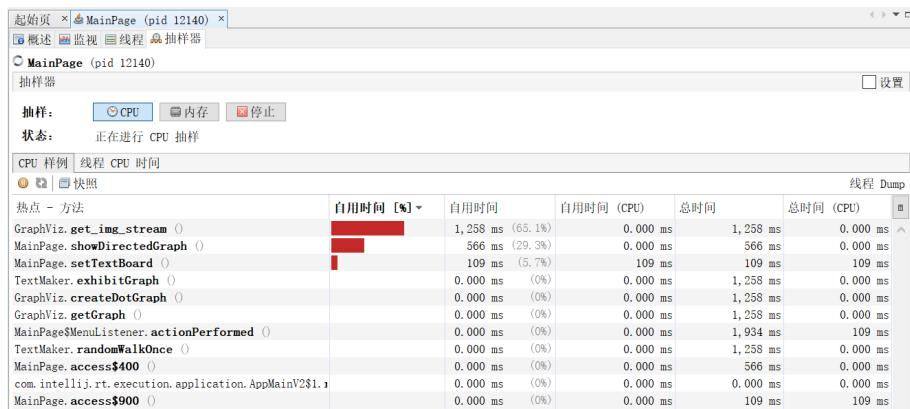


图 8-15 随机游走耗时分析

的二进制返回的函数， MainPage\$ButtonListener.actionPerformed() 为处理按钮点击事件的函数， TextMaker.findShortestPath() 为查询单源最短路径的函数， TextMaker.randomWalkOnce() 为随机游走一次的函数。

由以上分析可知，占用时间的主要有 GraphViz.get_img_stream()， MainPage.showDirectedGraph()， MainPage.setTextBoard() 三个函数。与之前的情况相同，在不改变 GUI 框架以及使用 GraphViz 进行绘图的前提下无法从耗时方面对计算两个单词之间最短路劲的性能进行优化。

8.4 内存占用的统计结果与原因分析

在本节中，将分别分析项目在生成与展示有向图，查询桥接词，根据桥接词生成新文本，计算两个单词之间的最短路径，以及随机游走这五个功能方面的内存占用情况，并对占用内存的原因进行分析。

8.4.1 生成与展示有向图

在生成与展示有向图部分，我们将主要考查多次读取输入文件时，是否发生了内存泄漏的情况。测试的方法为分别记录读取一次文本，十次文本，二十次文本，三十次文本时的堆的使用情况，并进行比较。其中，每次读取使用相同的文本文件（也即 Lab1 的测试文件），并在每次记录数据前执行垃圾回收操作。由于程序每次读取文本时实际上在更新其内的有向图，并把原始文本以及预处理后的文本累加输出到 GUI 相应的文本框中，因此若无内存泄漏，则可预计 java.lang.String 在堆中的占用增加，其他类的占用应基本保持不变。

第一，二，三，四次记录的数据分别见图8-16，图8-17，图8-18，图8-19。读取三十次文本时记录的数据与读取一次文本时记录的数据的差异比较见图8-20。

由图8-16，图8-17，图8-18，图8-19可见，每次记录数据时主要占用堆空间的类为 char[] 以及 java.lang.String，合起来每次记录数据时都占用了超过 40% 的堆空间。另外，从图8-16到图8-19的变化中可以看出，每次记录数据时占用堆空间较多的类的实例数以及大小并没有发生较大的改变。由读取三十次文本与读取一次文本时堆的比较可见看出，char[] 的实例数增加了 5%，java.lang.String 的实例数增加了 5%，HashMap\$Node 的实例数增加了 0.3%，即只有 char[] 以及 java.lang.String 两个类的占用发生了小幅度的提升，其他类的堆占用基本保持不变。这与无内存泄漏的预期结果相符合，因此可以推断此过程中未发生内存泄漏。

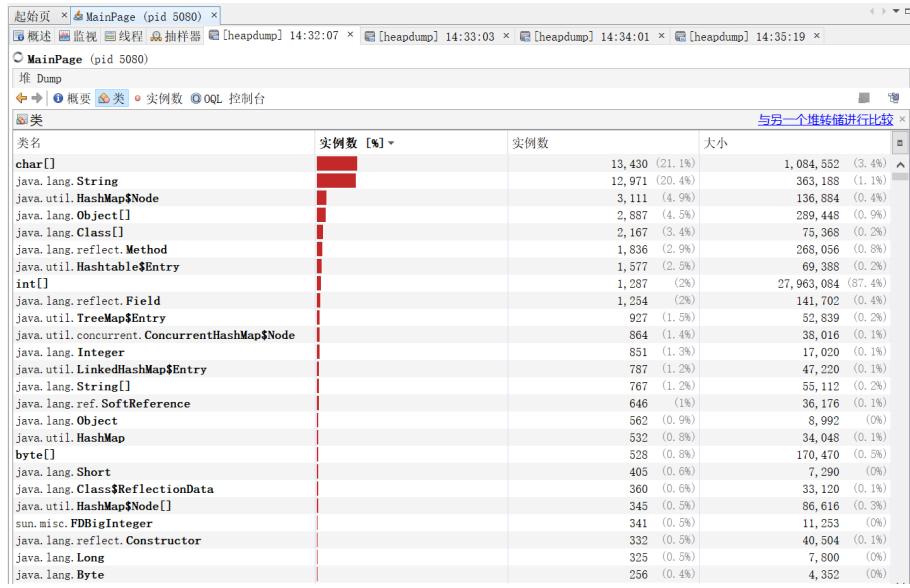


图 8-16 读取一次文本时堆情况

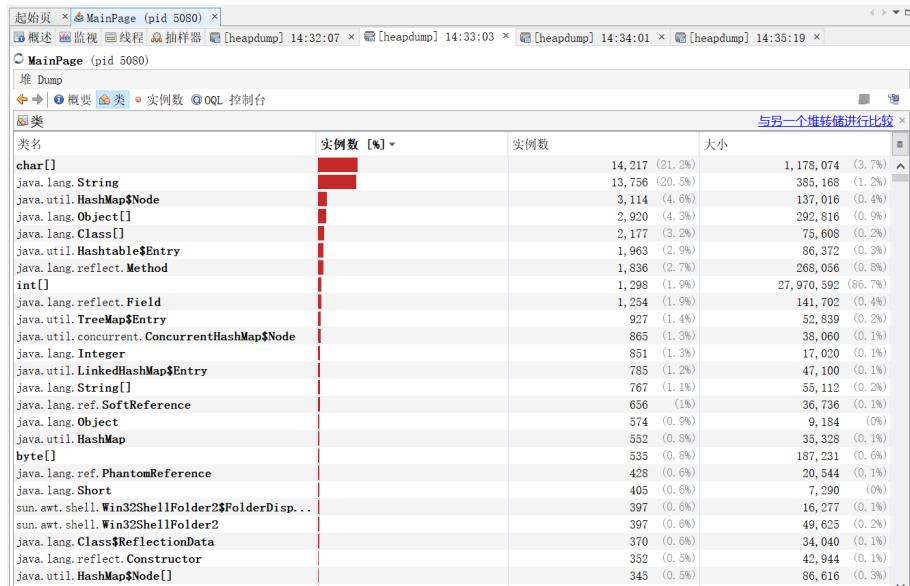


图 8-17 读取十次文本时堆情况

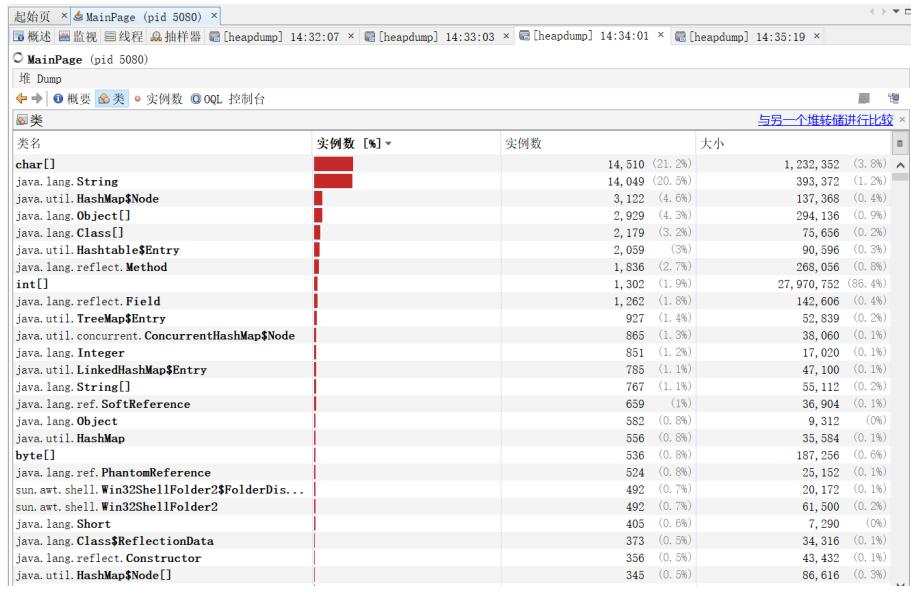


图 8-18 读取二十次文本时堆情况

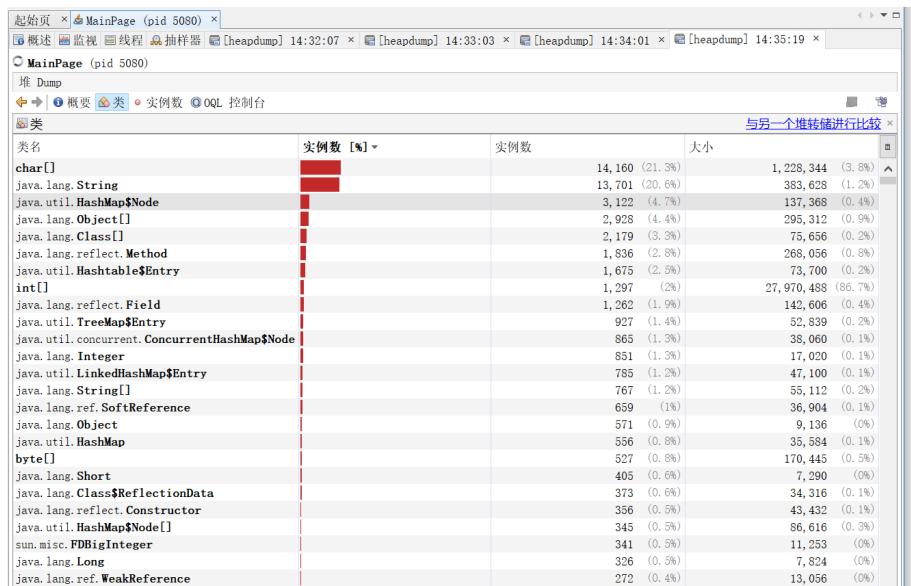


图 8-19 读取三十次文本时堆情况

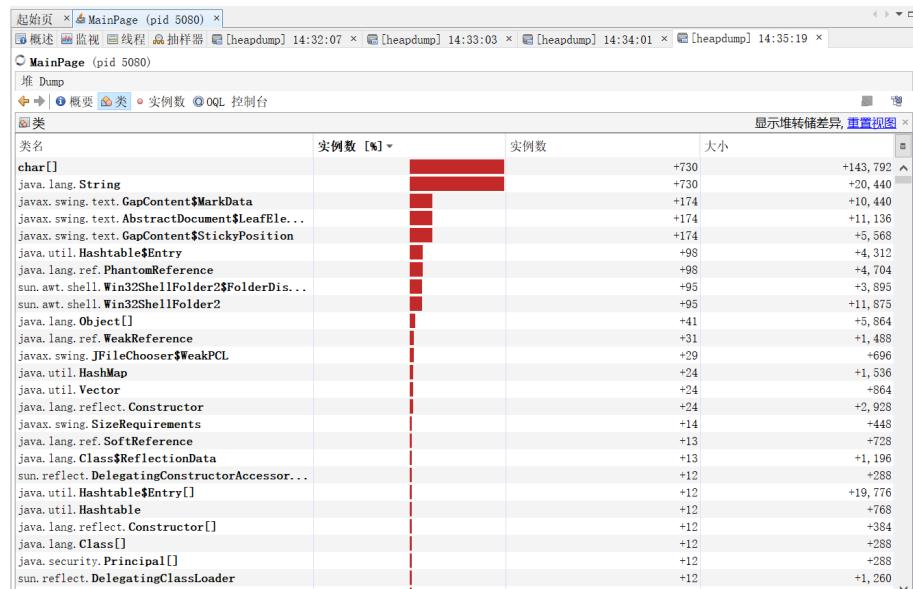


图 8-20 读取三十次文本与读取一次文本比较

8.4.2 查询桥接词

在查询桥接词的功能中，我们主要测试多次查询桥接词时是否会出现内存泄漏的情况。测试方法为读入 Lab1 测试数据中“读取并生成有向图”部分的文件数据，然后在此有向图上进行一次，十次，二十次查询有多个桥接词的两个单词 the, of 之间的桥接词，由此分析是否在查询桥接词的过程中发生了内存泄漏。其中，每次记录堆的数据占用情况前都进行垃圾回收。由于程序每次完成桥接词的查询之后通过弹出对话框的方式显示查询到的桥接词，关闭对话框后查询结果即消失，因此若无内存泄漏，则可预计所有类的堆占用情况皆不会发生较大的改变。

第一，二，三次记录的数据分别见图8-21，图8-22，图8-23。查询二十次桥接词时记录的数据与查询一次桥接词时记录的数据的差异比较见图8-24。

由图8-21，图8-22，图8-23，每次记录数据时主要占用堆空间的类同样为 char[] 以及 java.lang.String。同样的，从图8-21到图8-23的变化中可以看出，每次记录数据时占用堆空间较多的类的实例数以及大小并没有发生较大的改变。由查询二十次桥接词与查询一次桥接词时堆的比较可见看出，char[] 的实例数增加了 1%，java.lang.String 的实例数增加了 1%，HashMap\$Node 的实例数增加了 0.3%，基本可视作没有发生变化，与无内存泄漏时的预期基本一致，因此可以推断出在查询桥接词时未发生内存泄漏。

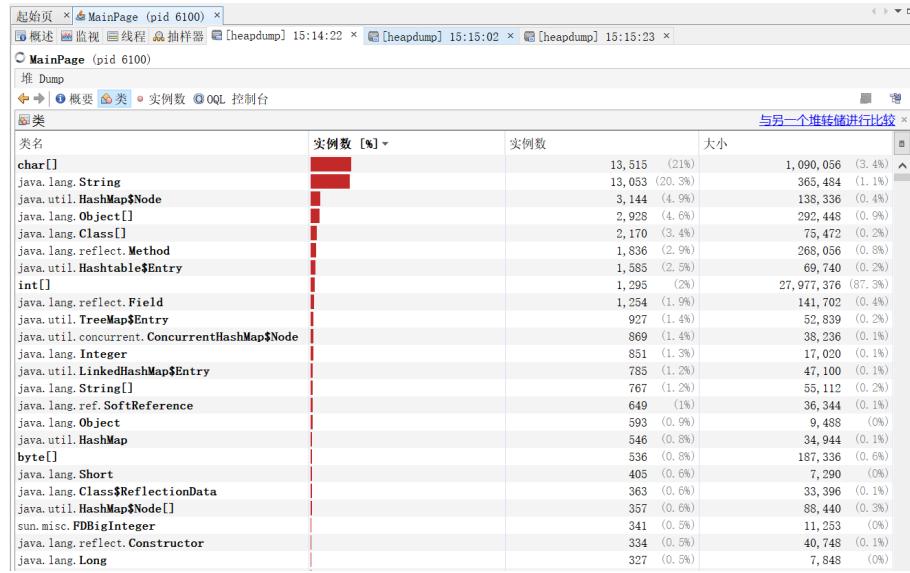


图 8-21 查询一次桥接词时堆情况

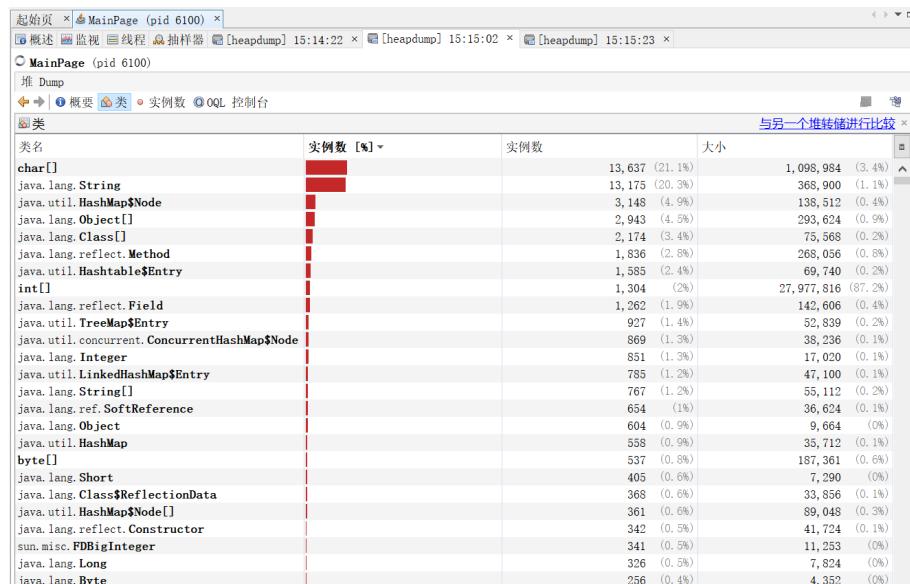


图 8-22 查询十次桥接词时堆情况

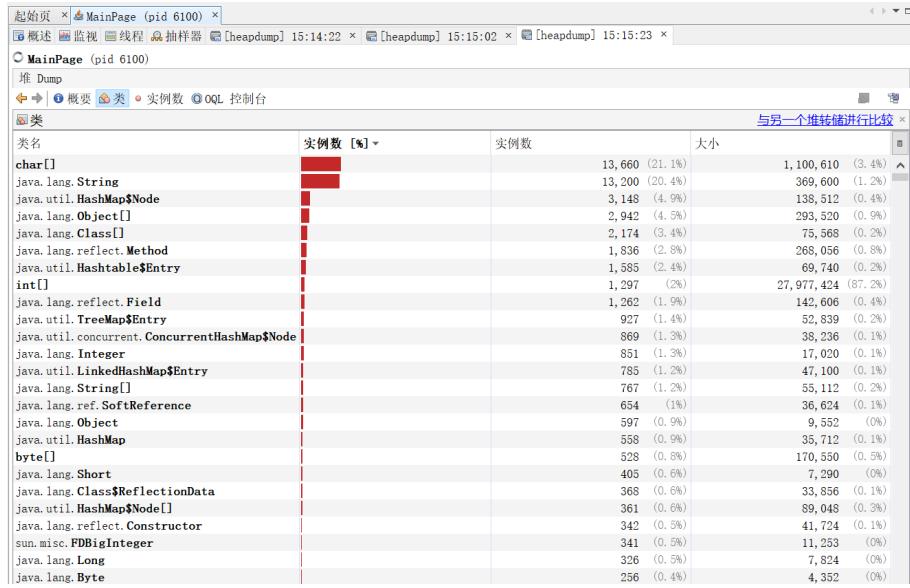


图 8-23 查询二十次桥接词时堆情况

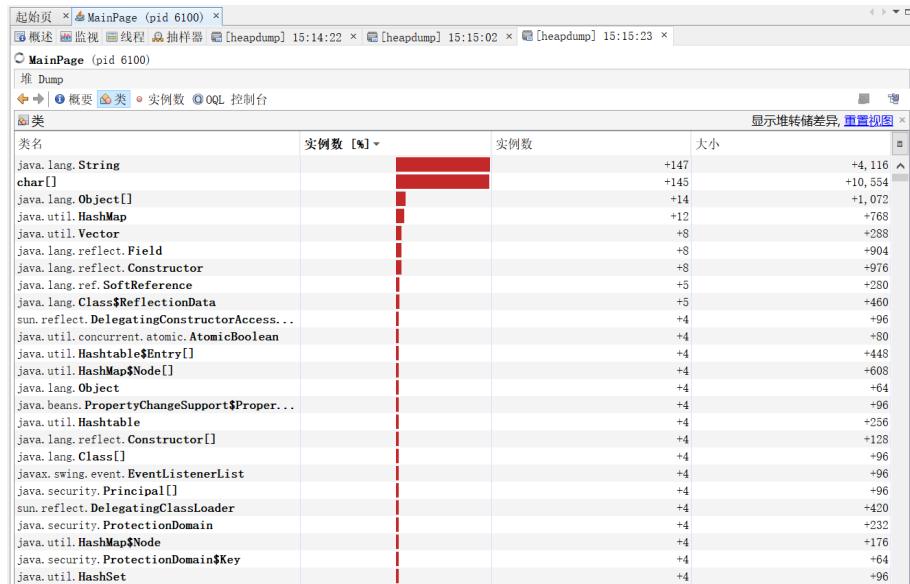


图 8-24 查询一次与查询二十次堆比较

8.4.3 根据桥接词生成新文本

在根据桥接词生成新文本的功能中，我们主要测试多次根据桥接词生成新文本时是否会出现内存泄漏的情况。测试方法为读入 Lab1 测试数据中“读取并生成有向图”部分的文件数据，然后在此有向图上进行一次，十次，二十次对同一句子（采用 Lab1 验收时的句子）来根据桥接词生成新文本，由此分析是否在由桥接词生成新文本的过程中发生了内存泄漏。其中，每次记录堆的数据占用情况前都进行垃圾回收。由于程序每次完成由桥接词生成新文本之后通过弹出对话框的方式显示生成的文本，关闭对话框后查询结果即消失，因此若无内存泄漏，则可预计所有类的堆占用情况皆不会发生较大的改变。

第一，二，三次记录的数据分别见图8-25，图8-26，图8-27。执行二十次由桥接词生成新文本与执行一次由桥接词生成新文本时记录的数据的差异比较见图8-24。

由图8-25，图8-26，图8-27，每次记录数据时主要占用堆空间的类同样为 char[] 以及 java.lang.String。同样的，从图8-25到图8-27的变化中可以看出，每次记录数据时占用堆空间较多的类的实例数以及大小并没有发生较大的改变。由执行二十次由桥接词生成新文本与执行一次由桥接词生成新文本时堆的比较可见看出，char[] 的实例数增加了 0.7%，java.lang.String 的实例数增加了 0.7%，HashMap\$Node 的实例数增加了 0%，因此各个类的实例数基本可视作没有发生变化，与无内存泄漏时的预期基本一致，因此可以推断出在由桥接词生成新文本时未发生内存泄漏。

8.4.4 计算两个单词之间的最短路径

在计算两个单词之间的最短路径的功能中，我们主要测试多次计算同一节点的单源最短路径时是否会出现内存泄漏的情况。测试方法为读入 Lab1 测试数据中“读取并生成有向图”部分的文件数据，然后在此有向图上进行一次，十次，二十次对同一单词来计算单源最短路径，由此分析是否在计算最短路径的过程中发生了内存泄漏。其中，每次记录堆的数据占用情况前都进行垃圾回收。由于程序每次完成最短路径的计算之后通过弹出对话框的方式显示计算得到的最短路径，关闭对话框后查询结果即消失，因此若无内存泄漏，则可预计所有类的堆占用情况皆不会发生较大的改变。

第一，二，三次记录的数据分别见图8-29，图8-30，图8-31。执行二十次计算单源最短路径与执行一次计算单源最短路径时记录的数据的差异比较见图8-32。

由图8-29，图8-30，图8-31可知，与之前类似的，每次记录数据时主要占用堆空间的类同样为 char[] 以及 java.lang.String。同样的，从图8-29到图8-31的变化中

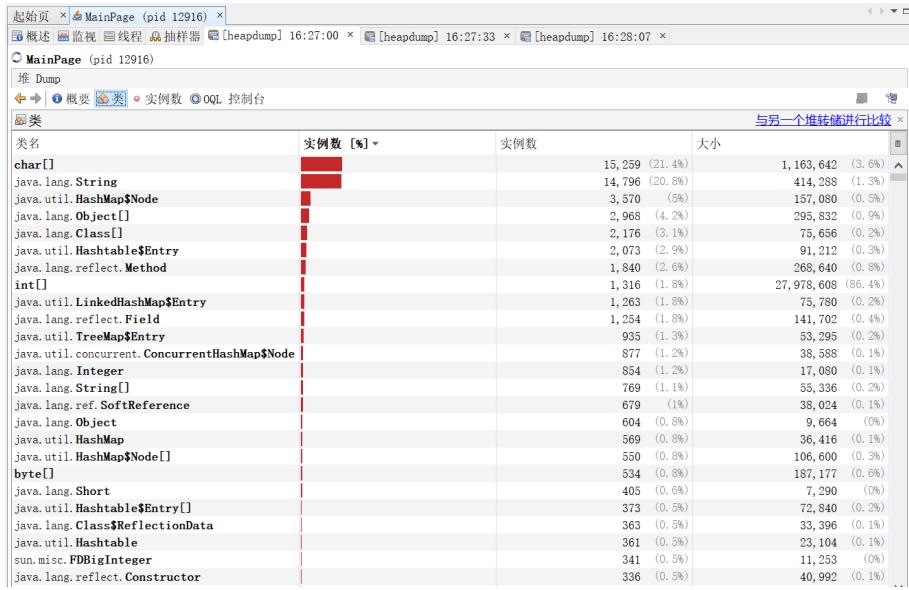


图 8-25 执行一次时堆情况

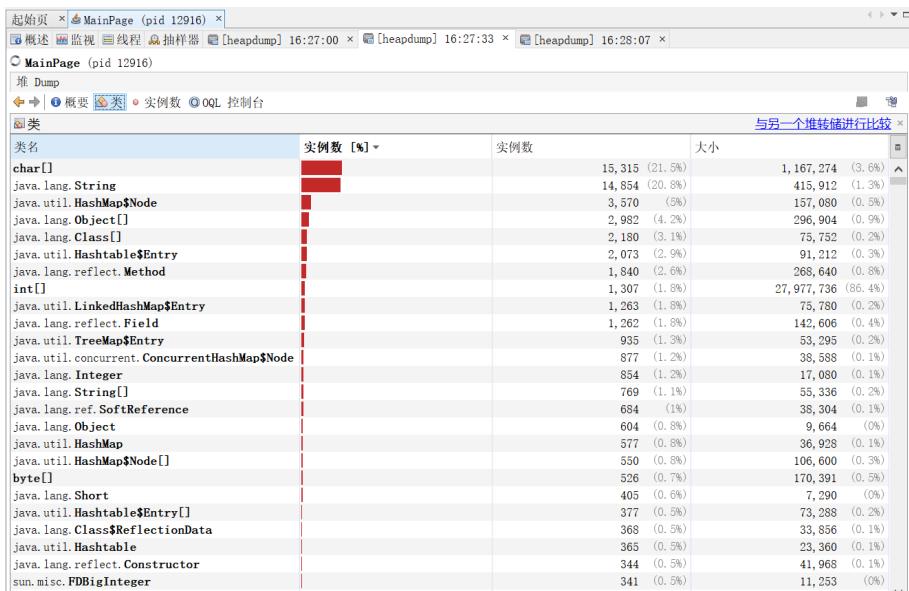


图 8-26 执行十次时堆情况

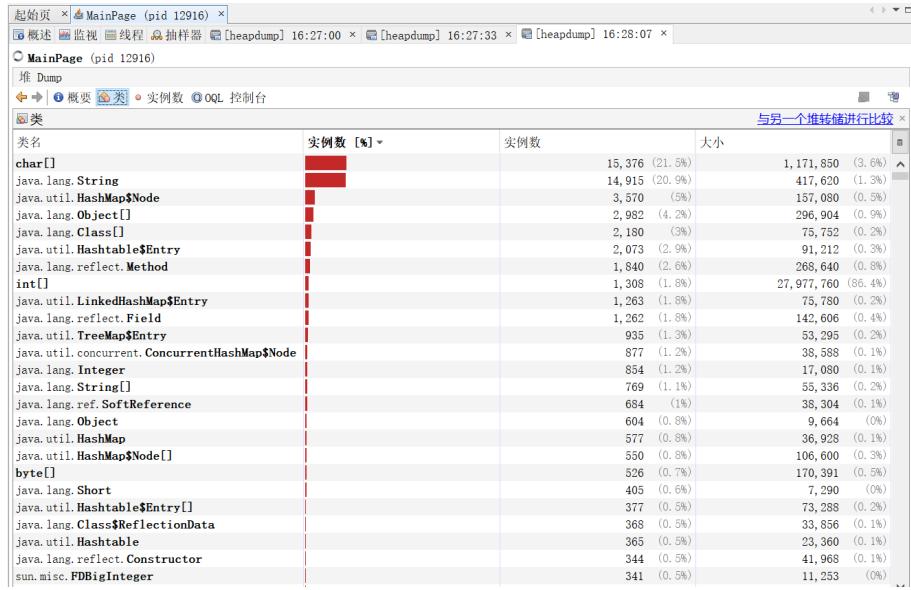


图 8-27 执行二十次时堆情况

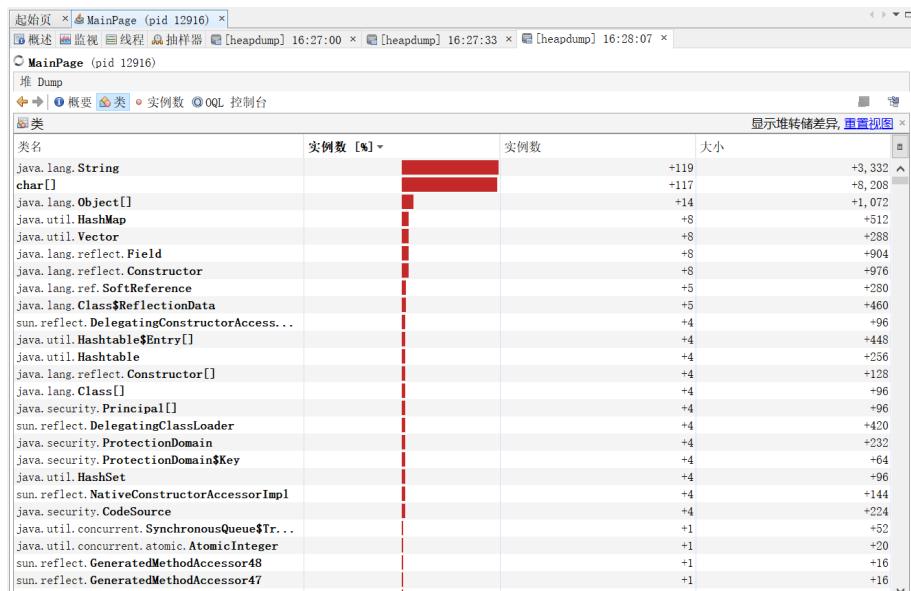


图 8-28 执行一次与执行二十次堆比较

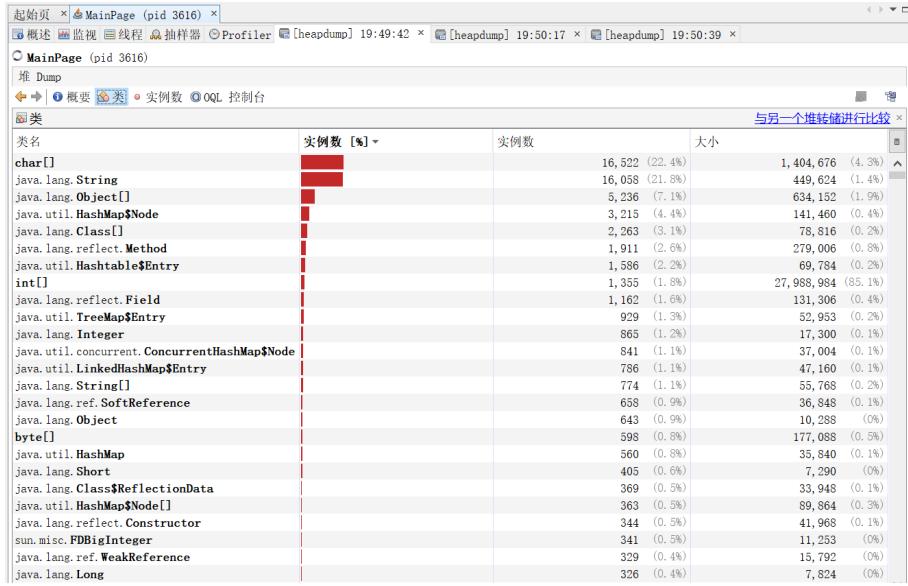


图 8-29 执行一次时堆情况

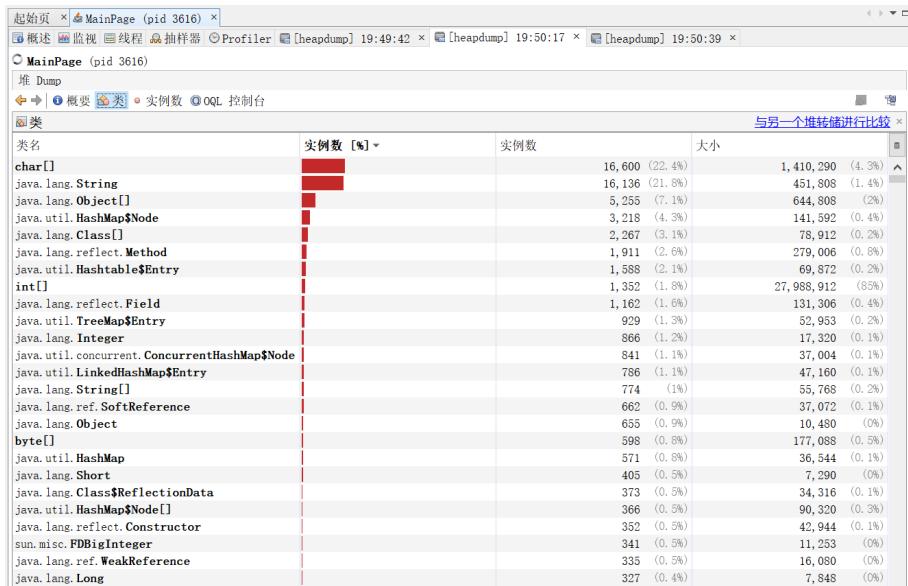


图 8-30 执行十次时堆情况

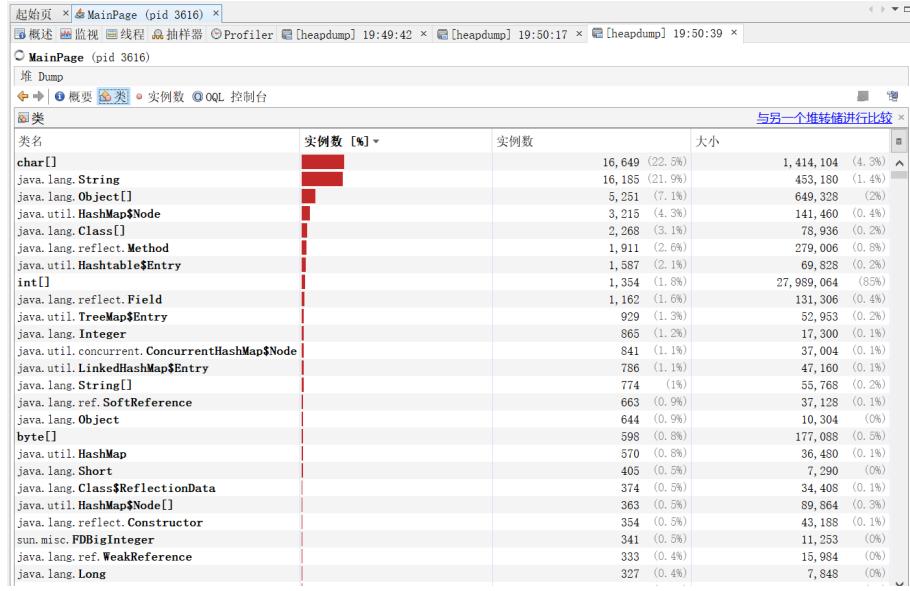


图 8-31 执行二十次时堆情况

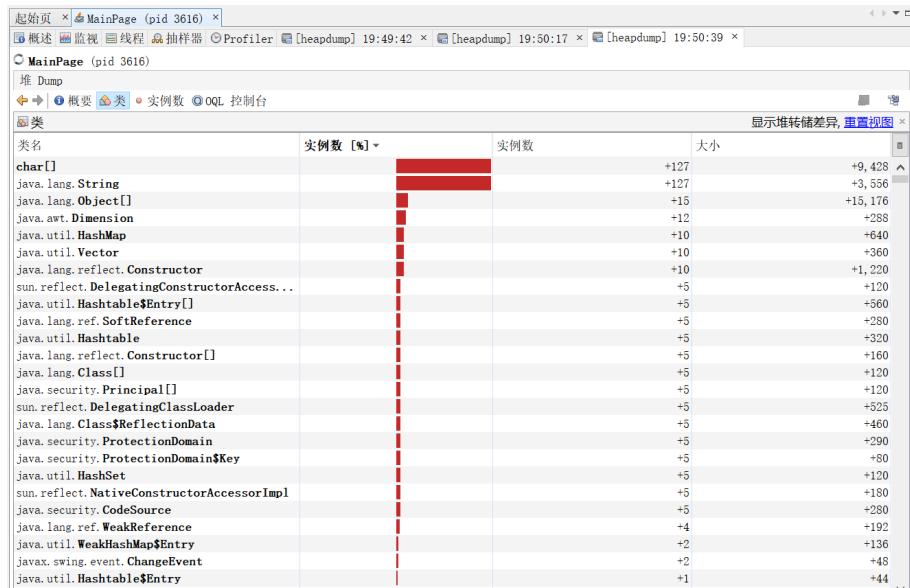


图 8-32 执行一次与执行二十次堆比较

可以看出，每次记录数据时占用堆空间较多的类的实例数以及大小并没有发生较大的改变。由执行二十次单源最短路径的计算与执行一次单源最短路径的计算时堆的比较可见看出，`char[]` 的实例数增加了 0.7%，`java.lang.String` 的实例数增加了 0.7%，`java.lang.Object[]` 的实例数增加了 0.2%，`HashMap$Node` 的实例数增加了 0.3%，因此各个类的实例数基本可视作没有发生变化，与无内存泄漏时的预期基本一致，因此可以推断出在计算最短路径时未发生内存泄漏。

8.4.5 随机游走

在测试随机游走时，我们采用同样的方法测试在随机游走的过程中是否出现了内存泄漏。测试的方法为读入 Lab1 测试数据中“读取并生成有向图”部分的文件数据，然后在此有向图上进行一次，十次，二十次随机游走并记录堆的信息，每一次随机游走皆进行到没有节点可以继续游走为止，由此分析是否在计算最短路径的过程中发生了内存泄漏。其中，每次记录堆的数据占用情况前都进行垃圾回收。由于每一次随机游走之间没有相互关联，GUI 中也不保留随机游走的数据，因此在没有发生内存泄漏的前提下，可预计所有类的堆占用情况皆不会发生较大的改变。

第一，二，三次记录的数据分别见图8-33，图8-34，图8-35。执行二十次随机游走与执行一次随机游走时记录的数据的差异比较见图8-36。

由图8-33，图8-34，图8-35可知，与之前类似的，每次记录数据时主要占用堆空间的类同样为`char[]` 以及`java.lang.String`。同样的，从图8-33到图8-35的变化中可以看出，每次记录数据时占用堆空间较多的类的实例数以及大小并没有发生较大的改变。由执行二十次随机游走与执行一次随机游走时堆的比较可见看出，`char[]` 的实例数增加了 3%，`java.lang.String` 的实例数增加了 3%，`HashMap$Node` 的实例数增加了 0.06%，因此各个类的实例数基本可视作没有发生变化，与无内存泄漏时的预期基本一致，因此可以推断出在随机游走时未发生内存泄漏。

最后，值得一提的是，在上述五种功能的分析中，所有的功能的堆信息中占用堆最多的都是`char[]` 以及`java.lang.String` 两个类，并且两个类的实例个数非常相近。这是因此在`java.lang.String` 类中，即有`char[]` 作为其实例变量来存储`java.lang.String` 表示的字符串中的字符，因此任何一个`java.lang.String` 的对象都会有一个`char[]` 变量，故其实例数量相近。

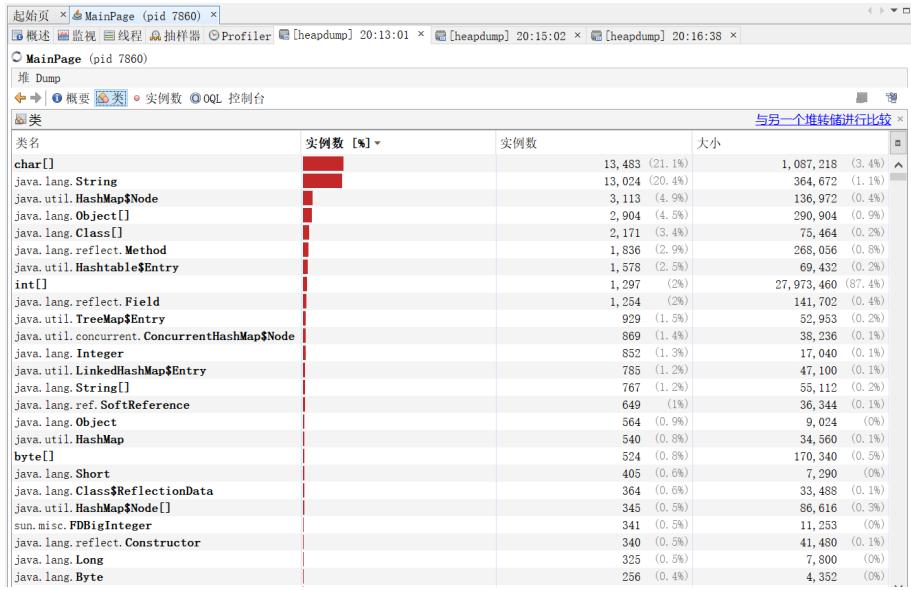


图 8-33 执行一次时堆情况

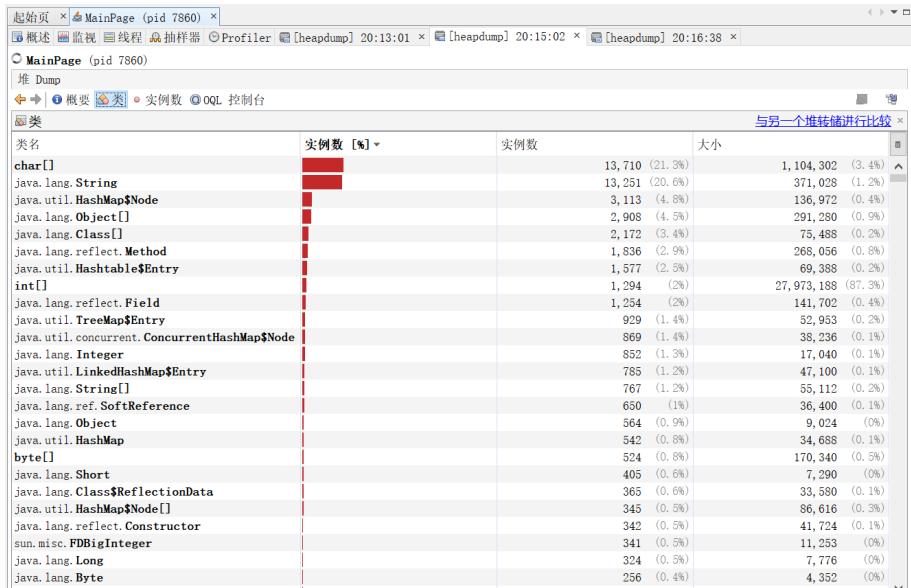


图 8-34 执行十次时堆情况

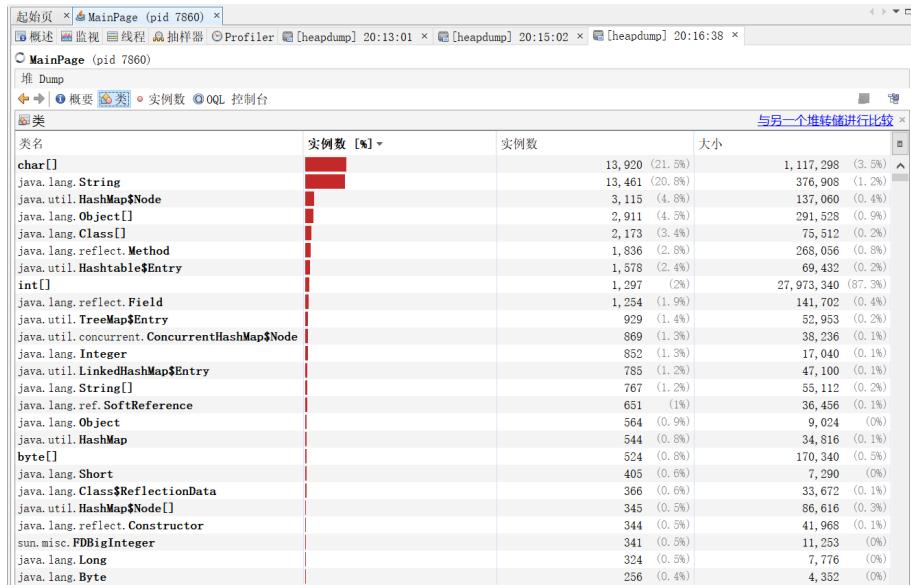


图 8-35 执行二十次时堆情况

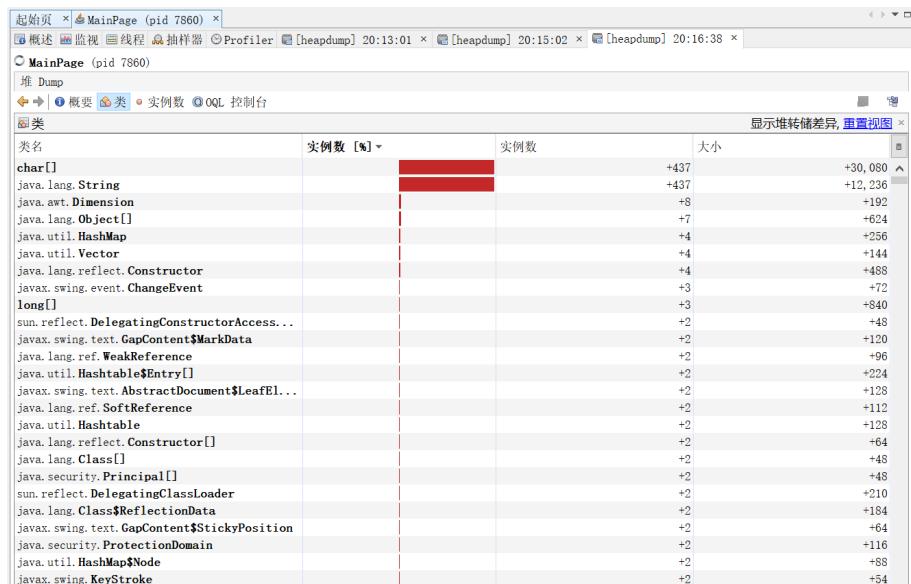


图 8-36 执行一次与执行二十次堆比较

8.5 代码改进之后的执行时间统计结果

在本节中将给出代码改进之后的执行时间统计结果。由使用 VisualVM 进行耗时分析以及内存分析的结果可知，在不改变 GUI 框架以及使用 GraphViz 进行绘图的前提下无法从耗时方面对程序的性能进行优化，也因为没有出现内存泄漏而无法通过消除内存泄漏来改进代码。因此本节中采用的修改后的代码为进行代码 review, checkstyle, FindBugs, PMD 进行检测与修正后的代码，并不包含利用 VisualVM 进行代码改进的部分。

对改进代码进行时间测试的方法为：依次执行“生成并展示有向图”，“查询桥接词”，“由桥接词生成新文本”，“计算最短路径”，“随机游走”五个功能，其中“生成并展示有向图”输入数据为 Lab1 的验收文件，“查询桥接词为”为查询从 the 到 of 的桥接词，“由桥接词生成新文本”中的文本为“In the big time, servitization one of the important trends-of the IT world.”，“计算最短路径”为计算单词 in 的单源最短路径，“随机游走”为游走到没有可选节点为止。测试时分别对每个功能执行时各个函数的耗时利用 VisualVM 进行记录。

“生成并展示有向图”耗时图像为图8-37，“查询桥接词”耗时图像为图8-38，“由桥接词生成新文本”耗时图像为图8-39，“计算最短路径”耗时图像为图8-40，“随机游走”耗时图像为图8-41。

8.6 代码改进之后的内存占用统计结果

在本节中将给出代码改进之后的内存占用统计结果。与上一节相同，本节中采用的修改后的代码为进行代码 review, checkstyle, FindBugs, PMD 进行检测与修正后的代码，并不包含利用 VisualVM 进行代码改进的部分。

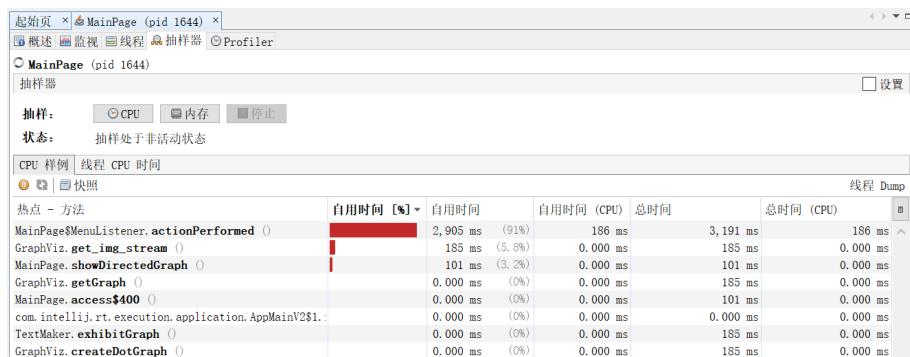


图 8-37 生成并展示有向图耗时

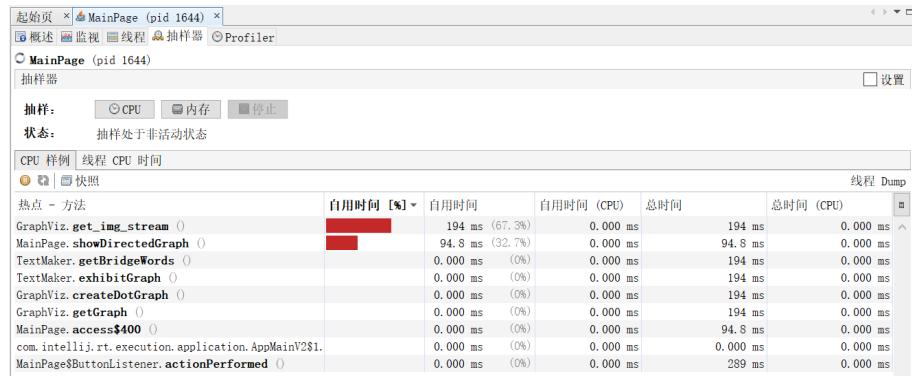


图 8-38 查询桥接词耗时

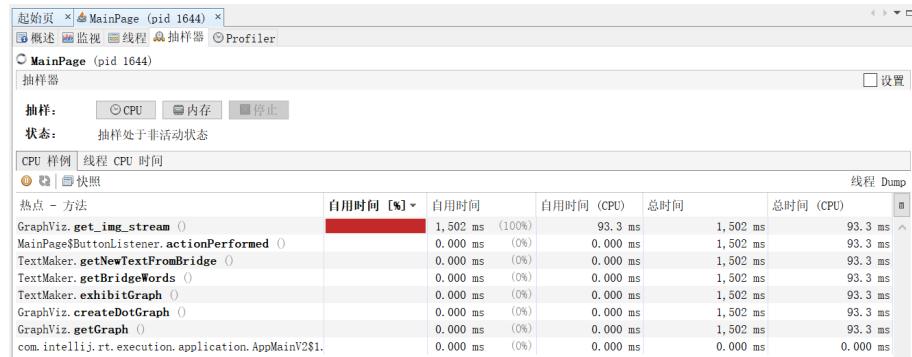


图 8-39 由桥接词生成新文本耗时



图 8-40 计算最短路径耗时

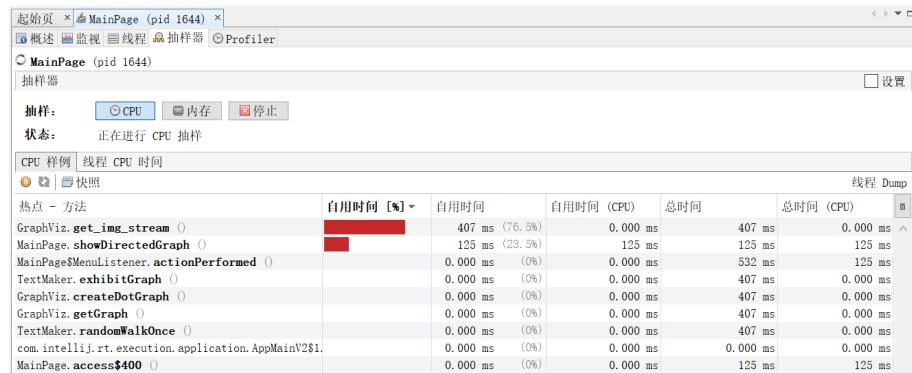


图 8-41 随机游走耗时

对改进代码进行内存测试的方法为：依次执行“生成并展示有向图”，“查询桥接词”，“由桥接词生成新文本”，“计算最短路径”，“随机游走”五个功能，其中“生成并展示有向图”输入数据为 Lab1 的验收文件，“查询桥接词为”为查询从 the 到 of 的桥接词，“由桥接词生成新文本”中的文本为“In the big time, servitization one of the important trends-of the IT world.”，“计算最短路径”为计算单词 in 的单源最短路径，“随机游走”为游走到没有可选节点为止。测试时分别对每个功能执行时的堆使用情况利用 VisualVM 进行记录。

“生成并展示有向图”的堆使用情况图像为图8-42，“查询桥接词”的堆使用情况图像为图8-43，“由桥接词生成新文本”的堆使用图像为图8-44，“计算最短路径”的堆使用图像为图8-45，“随机游走”的堆使用图像为图8-46。

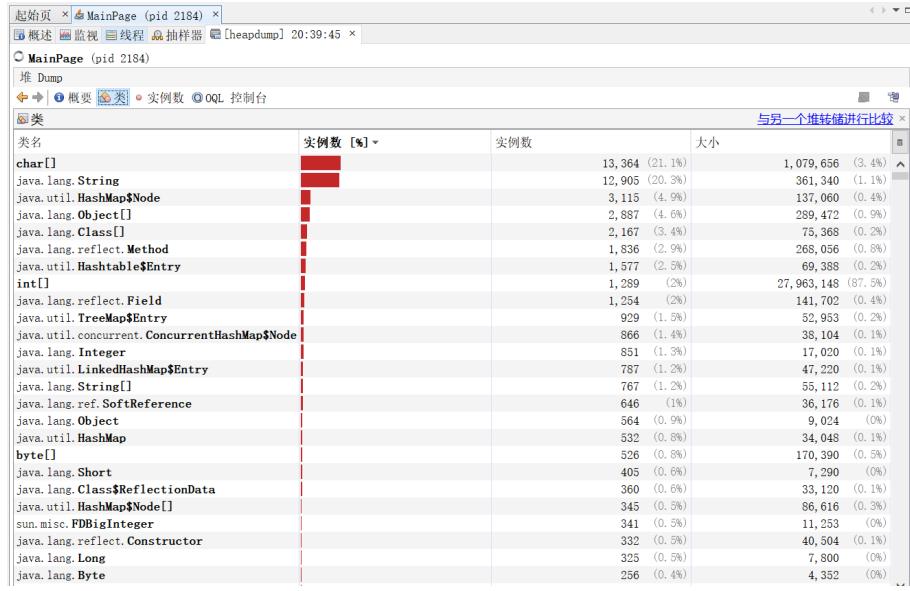


图 8-42 生成并展示有向图的堆使用

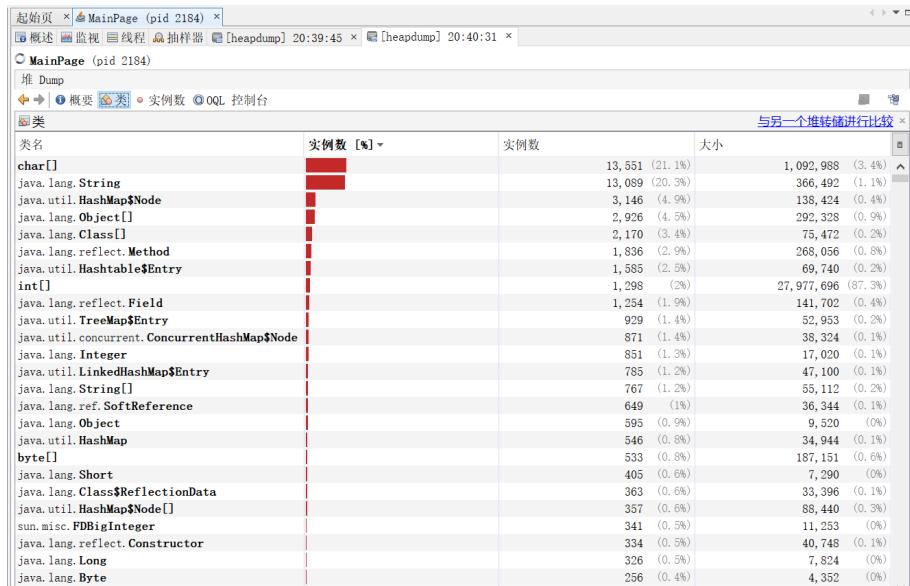


图 8-43 查询桥接词的堆使用

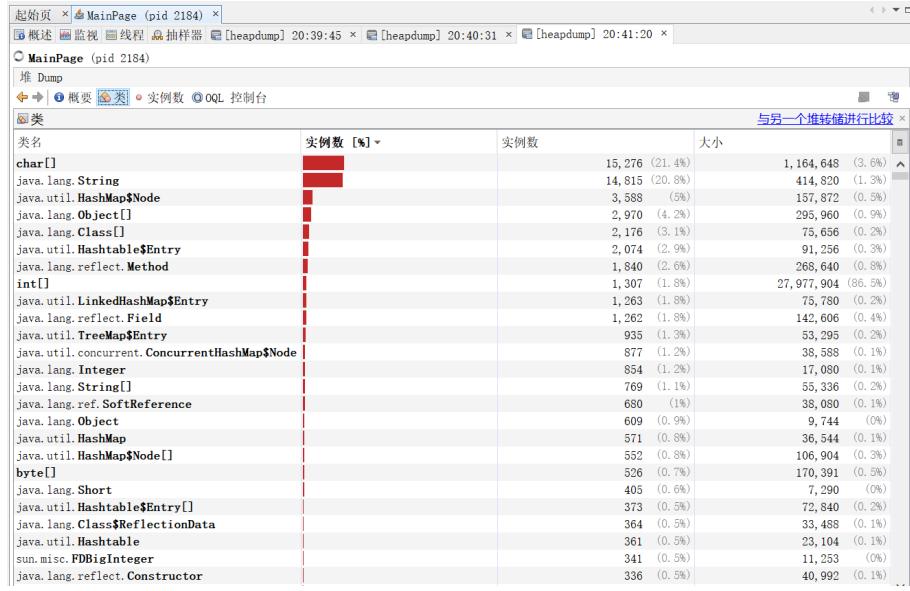


图 8-44 由桥接词生成新文本的堆使用

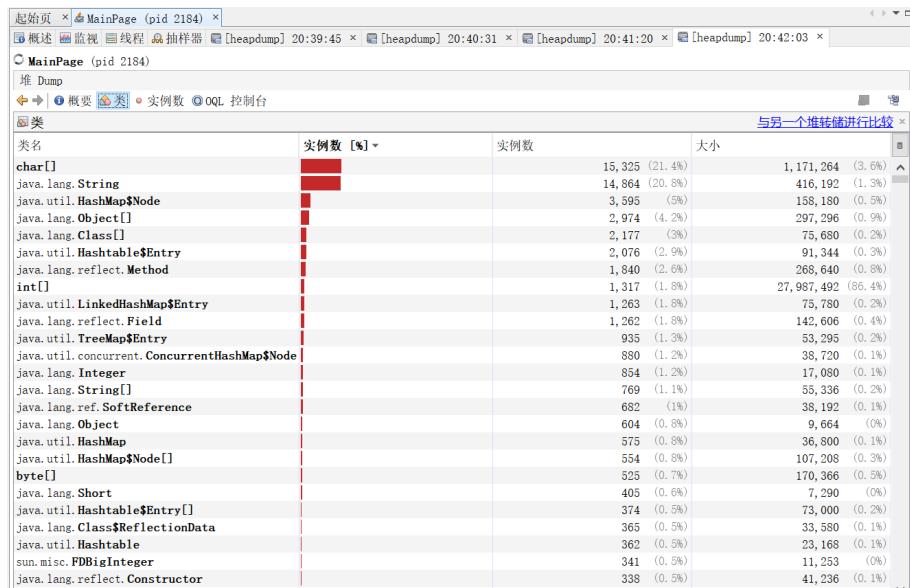


图 8-45 计算最短路径的堆使用

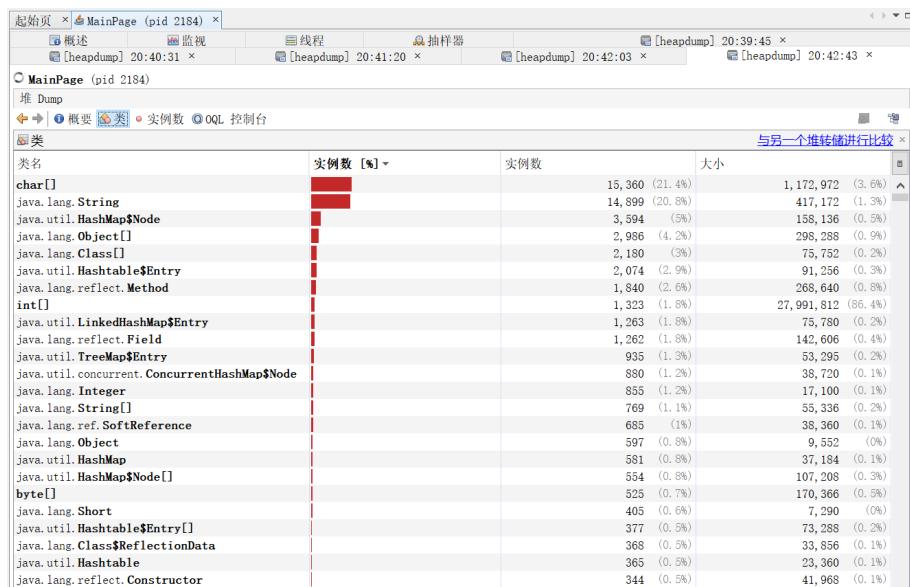


图 8-46 随机游走的堆使用

第 9 章 利用 Git/GitHub 进行协作的过程

本章中将描述在本次实验的第一，第二部分中利用 Git 与 GitHub 进行协作的截图与相应描述。

9.1 第一部分

首先，需要在 GitHub 上找到待评审代码，fork 至本组 GitHub 仓库内并重命名为 Lab4，完成后截图见图9-1。

然后，将 fork 得到的仓库 Lab4 clone 至本组的本地仓库 Lab4，完成后截图见图9-2。

完成 clone 之后，人工 review 和工具 review 之后将修改提交至本地仓库，完成后截图见图9-3。

提交至本地仓库后，将所有修改历史 push 到 GitHub 上本组仓库 Lab4，完成后截图见图9-4。

Push 至远程仓库后，在 Github 上将最新的提交 Pull Request 到原作者的 Lab1 仓库，完成后截图见图9-5。

由此，即完成了利用 Git 与 GitHub 进行协作的第一部分。

9.2 第二部分

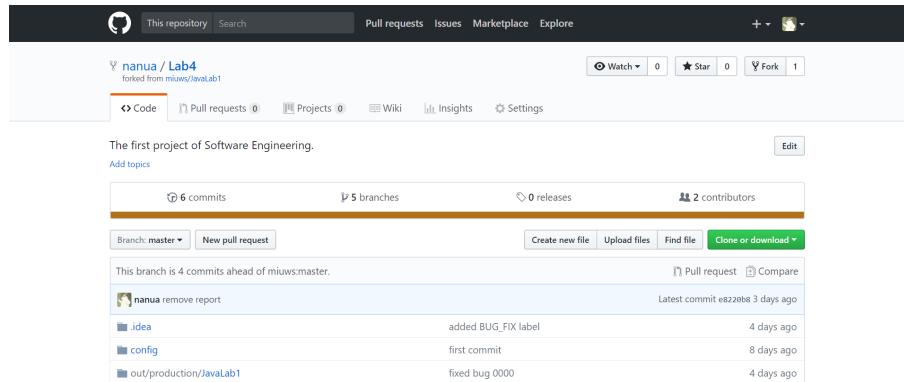
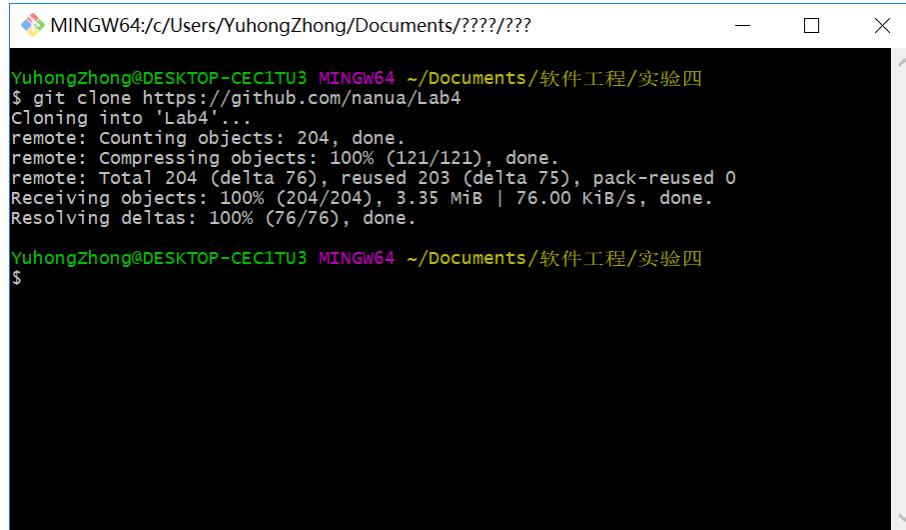


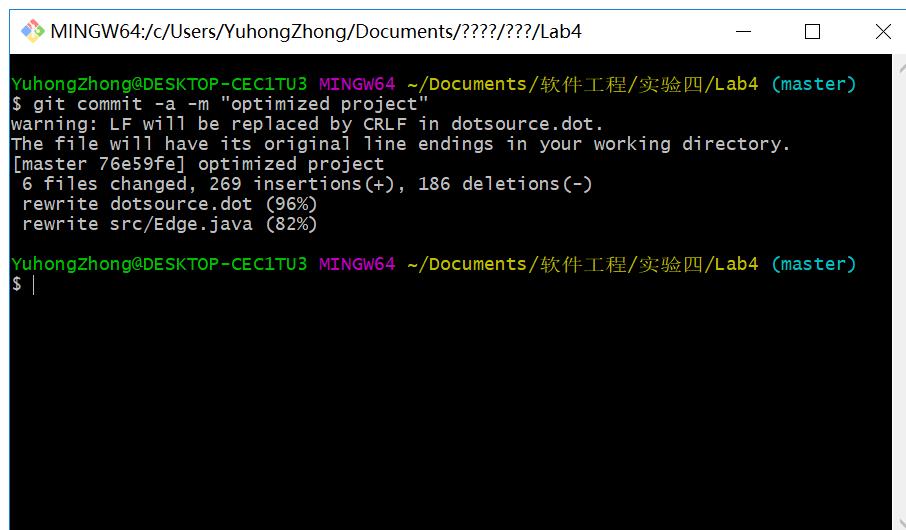
图 9-1 Fork 代评审代码至本组仓库



```
MINGW64:/c/Users/YuhongZhong/Documents/????/??? YuhongZhong@DESKTOP-CEC1TU3 MINGW64 ~/Documents/软件工程/实验四
$ git clone https://github.com/nanua/Lab4
Cloning into 'Lab4'...
remote: Counting objects: 204, done.
remote: Compressing objects: 100% (121/121), done.
remote: Total 204 (delta 76), reused 203 (delta 75), pack-reused 0
Receiving objects: 100% (204/204), 3.35 MiB | 76.00 KiB/s, done.
Resolving deltas: 100% (76/76), done.

YuhongZhong@DESKTOP-CEC1TU3 MINGW64 ~/Documents/软件工程/实验四
$
```

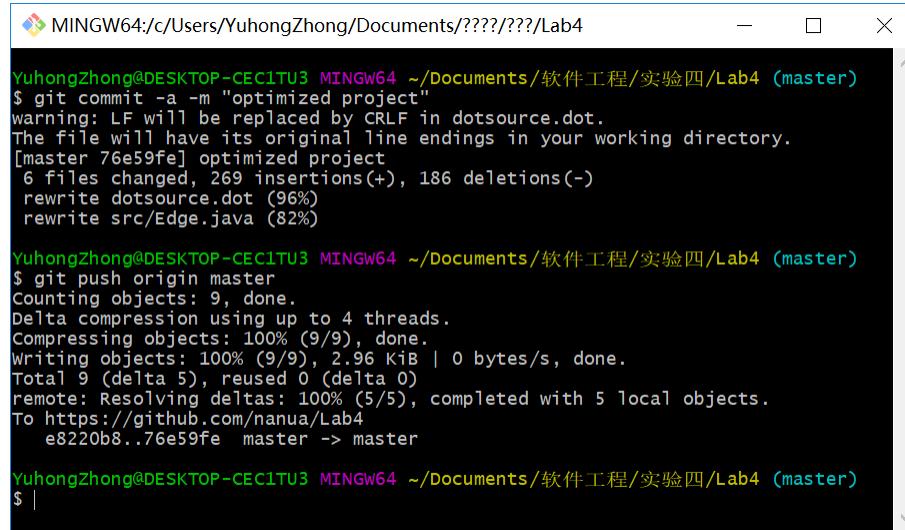
图 9-2 将远程仓库 clone 到本地



```
MINGW64:/c/Users/YuhongZhong/Documents/????/???/Lab4 YuhongZhong@DESKTOP-CEC1TU3 MINGW64 ~/Documents/软件工程/实验四/Lab4 (master)
$ git commit -a -m "optimized project"
warning: LF will be replaced by CRLF in dotsource.dot.
The file will have its original line endings in your working directory.
[master 76e59fe] optimized project
 6 files changed, 269 insertions(+), 186 deletions(-)
 rewrite dotsource.dot (96%)
 rewrite src/Edge.java (82%)

YuhongZhong@DESKTOP-CEC1TU3 MINGW64 ~/Documents/软件工程/实验四/Lab4 (master)
$ |
```

图 9-3 将修改提交至本地仓库

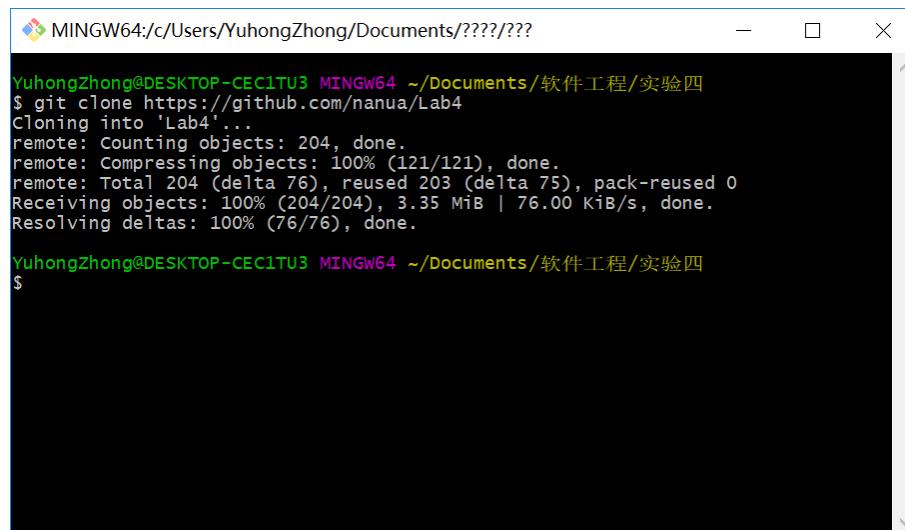


```
MINGW64:/c/Users/YuhongZhong/Documents/????/???/Lab4 (master)
$ git commit -a -m "optimized project"
warning: LF will be replaced by CRLF in datasource.dot.
The file will have its original line endings in your working directory.
[master 76e59fe] optimized project
 6 files changed, 269 insertions(+), 186 deletions(-)
 rewrite datasource.dot (96%)
 rewrite src/Edge.java (82%)

YuhongZhong@DESKTOP-CEC1TU3 MINGW64 ~/Documents/软件工程/实验四/Lab4 (master)
$ git push origin master
Counting objects: 9, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (9/9), 2.96 KiB | 0 bytes/s, done.
Total 9 (delta 5), reused 0 (delta 0)
remote: Resolving deltas: 100% (5/5), completed with 5 local objects.
To https://github.com/nanua/Lab4
  e8220b8..76e59fe master -> master

YuhongZhong@DESKTOP-CEC1TU3 MINGW64 ~/Documents/软件工程/实验四/Lab4 (master)
$ |
```

图 9-4 将本地仓库内容 push 至远程仓库



```
MINGW64:/c/Users/YuhongZhong/Documents/????/???
YuhongZhong@DESKTOP-CEC1TU3 MINGW64 ~/Documents/软件工程/实验四
$ git clone https://github.com/nanua/Lab4
Cloning into 'Lab4'...
remote: Counting objects: 204, done.
remote: Compressing objects: 100% (121/121), done.
remote: Total 204 (delta 76), reused 203 (delta 75), pack-reused 0
Receiving objects: 100% (204/204), 3.35 MiB | 76.00 KiB/s, done.
Resolving deltas: 100% (76/76), done.

YuhongZhong@DESKTOP-CEC1TU3 MINGW64 ~/Documents/软件工程/实验四
$ |
```

图 9-5 提交 Pull Request

第 10 章 评述

10.1 对代码规范方面的评述

10.2 对代码性能方面的评述

第 11 章 计划与实际进度

在本节中将描述本次实验中各个环节的任务名称，计划时间长度，实际耗费时间，以及提前或延期的原因分析。

任务名称	计划时间长度（分钟）	实际耗费时间（分钟）	提前或延期原因分析
在 IntelliJ 下配置工具	30	60	IntelliJ 插件下载出现异常
在 GitHub 上找到待评审代码并 fork	10	10	
将远程仓库 clone 至本地	5	5	
进行代码走查并消除问题	120	240	在使用时发现代码错误较多，花了一定时间上网查询解决方案与修改
提交代码至本地 Git 仓库	5	5	
使用 Checkstyle 检查并修正代码风格	60	120	代码风格错误较多，花了较多时间修复
使用 PMD 检查并修正代码错误	120	240	PMD 发现的错误较多，花了较多时间修复
使用 FindBugs 检查并修正代码错误	120	60	FindBugs 发现的代码错误较少

任务名称	计划时间长度(分钟)	实际耗费时间(分钟)	提前或延期原因分析
使用 VisualVM 测试并提高代码性能	360	300	未通过 VisualVM 发现严重影响代码性能的缺陷
提交代码至本地 Git 仓库	5	5	
将本地仓库 push 至本组远程仓库	5	1	Git 使用较熟练

第 12 章 小结