

**2019年春季学期  
计算机学院《软件构造》课程**

**Lab 5实验报告**

|  |  |
| --- | --- |
| 姓名 | 张钟升 |
| 学号 | 1170300431 |
| 班号 | 1703004 |
| 电子邮件 | [1170300431@stu.hit.edu.cn](mailto:1170300431@stu.hit.edu.cn) |
| 手机号码 | 13029801568 |

**目录**

[1 实验目标概述 1](#_Toc10383059)

[2 实验环境配置 1](#_Toc10383060)

[3 实验过程 2](#_Toc10383061)

[3.1 Static Program Analysis 3](#_Toc10383062)

[3.1.1 人工代码走查（walkthrough） 3](#_Toc10383063)

[3.1.2 使用CheckStyle和SpotBugs进行静态代码分析 3](#_Toc10383064)

[3.2 Java I/O Optimization 3](#_Toc10383065)

[3.2.1 多种I/O实现方式 3](#_Toc10383066)

[3.2.2 多种I/O实现方式的效率对比分析 5](#_Toc10383067)

[3.3 Java Memory Management and Garbage Collection (GC) 6](#_Toc10383068)

[3.3.1 使用-verbose:gc参数 6](#_Toc10383069)

[3.3.2 用jstat命令行工具的-gc和-gcutil参数 8](#_Toc10383070)

[3.3.3 使用jmap -heap命令行工具 8](#_Toc10383071)

[3.3.4 使用jmap -clstats命令行工具 9](#_Toc10383072)

[3.3.5 使用jmap -histo命令行工具 9](#_Toc10383073)

[3.3.6 使用JMC/JFR、jconsole或VisualVM工具 10](#_Toc10383074)

[3.3.7 分析垃圾回收过程 10](#_Toc10383075)

[3.3.8 配置JVM参数并发现优化的参数配置 11](#_Toc10383076)

[3.4 Dynamic Program Profiling 11](#_Toc10383077)

[3.4.1 使用JMC或VisualVM进行CPU Profiling 11](#_Toc10383078)

[3.4.2 使用VisualVM进行Memory profiling 12](#_Toc10383079)

[3.5 Memory Dump Analysis and Performance Optimization 12](#_Toc10383080)

[3.5.1 内存导出 12](#_Toc10383081)

[3.5.2 使用MAT分析内存导出文件 13](#_Toc10383082)

[3.5.3 发现热点/瓶颈并改进、改进前后的性能对比分析 15](#_Toc10383083)

[3.5.4 在MAT内使用OQL查询内存导出 15](#_Toc10383084)

[3.5.5 观察jstack/jcmd导出程序运行时的调用栈 18](#_Toc10383085)

[3.5.6 使用设计模式进行代码性能优化 19](#_Toc10383086)

[3.6 Git仓库结构 19](#_Toc10383087)

[4 实验进度记录 20](#_Toc10383088)

[5 实验过程中遇到的困难与解决途径 20](#_Toc10383089)

[6 实验过程中收获的经验、教训、感想 20](#_Toc10383090)

[6.1 实验过程中收获的经验和教训 20](#_Toc10383091)

[6.2 针对以下方面的感受 20](#_Toc10383092)

# 实验目标概述

根本次实验通过对 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 SpotBugs）
* 动态代码分析（Java 命令行工具 jstat、jmap、jcmd、VisualVM、JMC、

JConsole 等）

* 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

在IDEA中配置实验环境.

* CheckStyle

在IDEA plugin market中安装CheckStyle-IDEA. (图1)

* SpotBugs

在IDEA plugin market中安装FindBugs-IDEA. (图1)

* VisualVM

在IDEA plugin market中安装VisualVM Launcher. (图1)

* MAT

下载独立的MAT程序: <http://www.eclipse.org/mat/downloads.php> (图2)

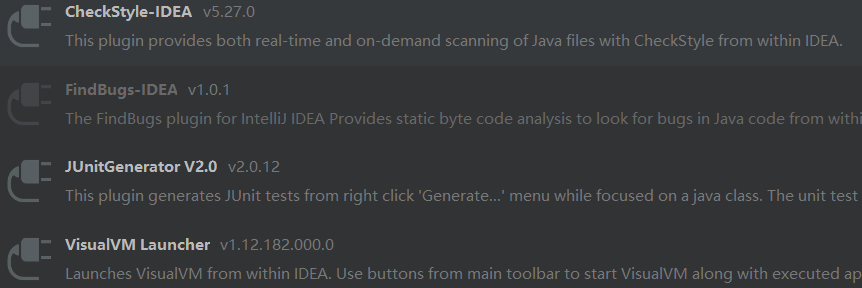


图 1 IDEA plugin market

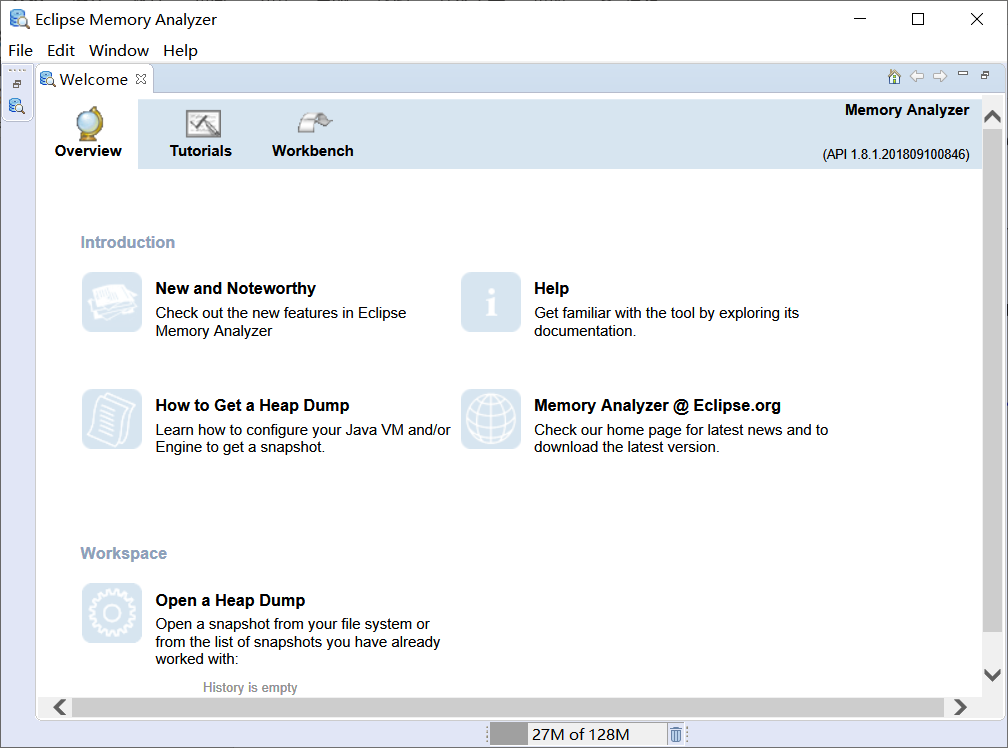


图 2 MAT screenshot

repo: <https://github.com/ComputerScienceHIT/Lab5-1170300431.git>

# 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）

* 更改类名: pair->Pair
* 更改方法名: MyMethod->myMethod

(不清楚包名应当符合要求还是符合规范 就没改...)

### 使用CheckStyle和SpotBugs进行静态代码分析

* 代码格式化, 包括代码缩进, 单行字数, 空行, block brace, 优化导入等.
* 每行只声明一个变量.
* 字段声明符合规范: OBJDIF1->objDIF1
* 局部变量声明规范: rSet->rtSet
* 为某些变量增加final.
* 顶级类应当在独立的文件: 使InfoParser为内部类
* switch必须有default: 为枚举switch添加default
* JavaDoc必须有描述, 且描述必须以句号结尾
* 为某些方法添加JavaDoc
* Inspections: 变量未赋值或为空/未使用的变量
* Inspections: 无法退出的递归调用或循环
* Inspections: 不可达的语句
* Inspections: 无效的调用(发生于repeat.clear(), 此时repeat为空)
* Inspections: 将可能为null的值传递给标注@NotNull的参数

CheckStyle和SpotBugs的区别在于, CheckStyle检查代码和注释的样式, 规范, 是对代码的检查; SpotBugs检查编译后的.class文件, 寻找可能存在的bug和不规范的写法.

## Java I/O Optimization

### 多种I/O实现方式

读文件: Reader, Channel, Scanner.

写文件: Writer, Channel, Stream.

使用setReadingMethod和setWritingMethod来切换读写IO的策略. 除了提供的3+3种IO策略, 用户可以自行构造Consumer或BiConsumer传递给setReadingMethod或setWritingMethod以实现更多的IO策略.

1. /\*\*
2. \* read a file, as a list of String separated by \n(\r\n). no blank line is included.
3. \*
4. \* @param path file path
5. \* @return list of String, separated by \n(\r\n).
6. \*/
7. @Nullable List<String> read(String path);
9. /\*\*
10. \* write a file, from a list of string.
11. \* @param path file path to write.
12. \* @param content list of string.
13. \* @return true if success.
14. \*/
15. **boolean** write(String path, List<String> content);
17. **public** **void** setReadingMethod(@NotNull Function<String, List<String>> readingMethod);
19. **void** setWritingMethod(BiPredicate<String, List<String>> writingMethod);
21. **public** **static** **final** Function<String, List<String>> readerStrategy;
23. **public** **static** **final** Function<String, List<String>> channelStrategy\_read;
25. **public** **static** **final** Function<String, List<String>> scannerStrategy;
27. **public** **static** **final** BiPredicate<String, List<String>> writerStrategy;
29. **public** **static** **final** BiPredicate<String, List<String>> channelStrategy\_write;
31. **public** **static** **final** BiPredicate<String, List<String>> streamStrategy;

### 多种I/O实现方式的效率对比分析

* 如何收集你的程序I/O语法文件的时间

1. @Test
2. **public** **void** testStellar\_channel() {
3. var cf = CircularOrbitFactory.getDefault();
4. cf.setWritingMethod(CircularOrbitFactory.channelStrategy\_write);
5. var start = System.currentTimeMillis();
6. cf.write("out/txt/stellar\_channel", sList);
7. var end = System.currentTimeMillis();
8. System.out.println(getMethodName() + ": " + (end - start) + "ms");
9. }

这个例子是收集channel方式写文件的时间. 运行时间由两次System.currentTimeMillis()的差得到.

* 表格方式对比不同I/O的性能

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 读文件 | | | | |
| Type | TestID | Reader | Channel | Scanner |
| StellarSystem | #1 | 178 | 233 | 841 |
| #2 | 236 | 247 | 628 |
| #3 | 114 | 392 | 564 |
| Average | 176 | 290.6667 | 677.6667 |
| SocialNetowrkCircle | #1 | 283 | 315 | 1446 |
| #2 | 320 | 344 | 1411 |
| #3 | 281 | 359 | 1391 |
| Average | 294.6667 | 339.3333 | 1416 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 写文件 | | | | |
| Type | TestID | Writer | Channel | Stream |
| StellarSystem | #1 | 108 | 115 | 69 |
| #2 | 93 | 117 | 66 |
| #3 | 95 | 113 | 64 |
| Average | 98.66667 | 115 | 66.33333 |
| SocialNetworkCircle | #1 | 156 | 97 | 117 |
| #2 | 153 | 93 | 132 |
| #3 | 148 | 86 | 99 |
| Average | 152.3333 | 92 | 116 |

* 图形对比不同I/O的性能

## Java Memory Management and Garbage Collection (GC)

### 使用-verbose:gc参数[[1]](#footnote-1)

1. Young GC和Full GC的频率

Young GC: 14次

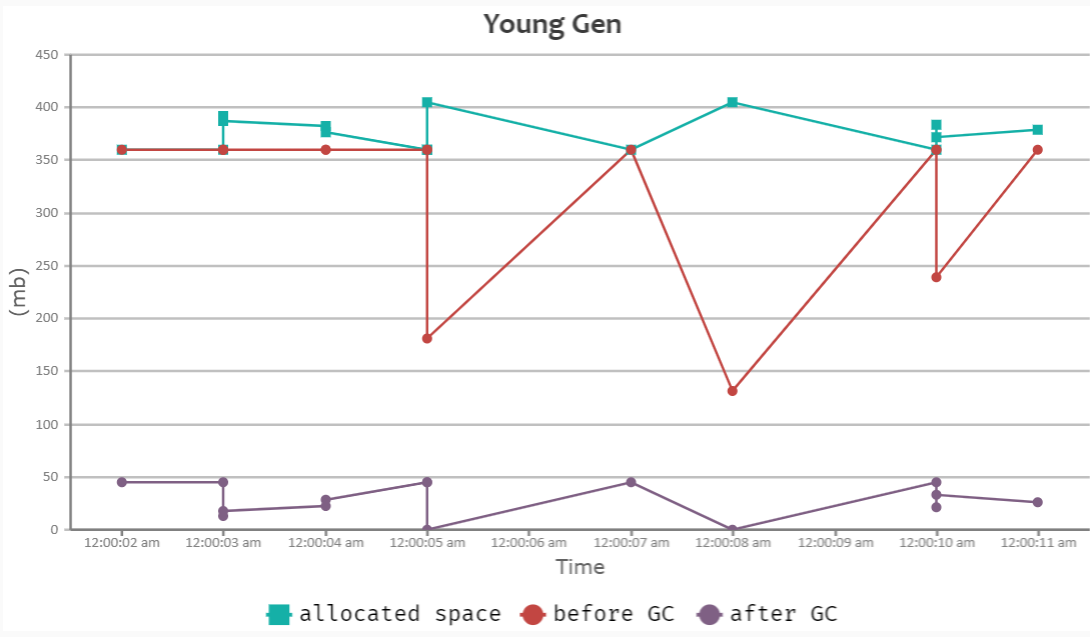
Full GC: 2次

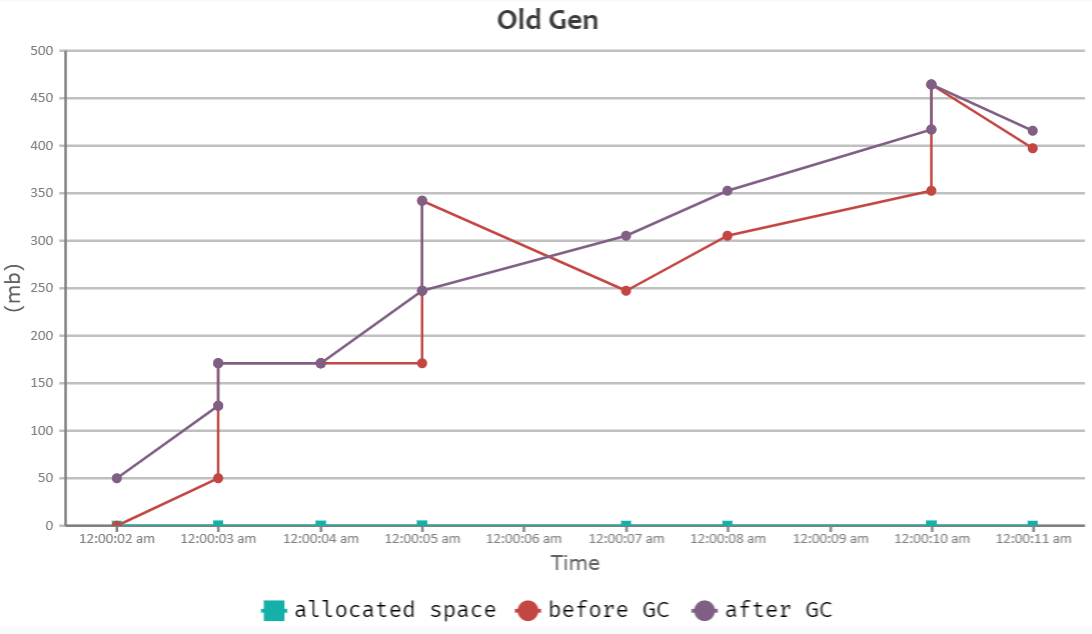
1. 两种GC单次耗费时间

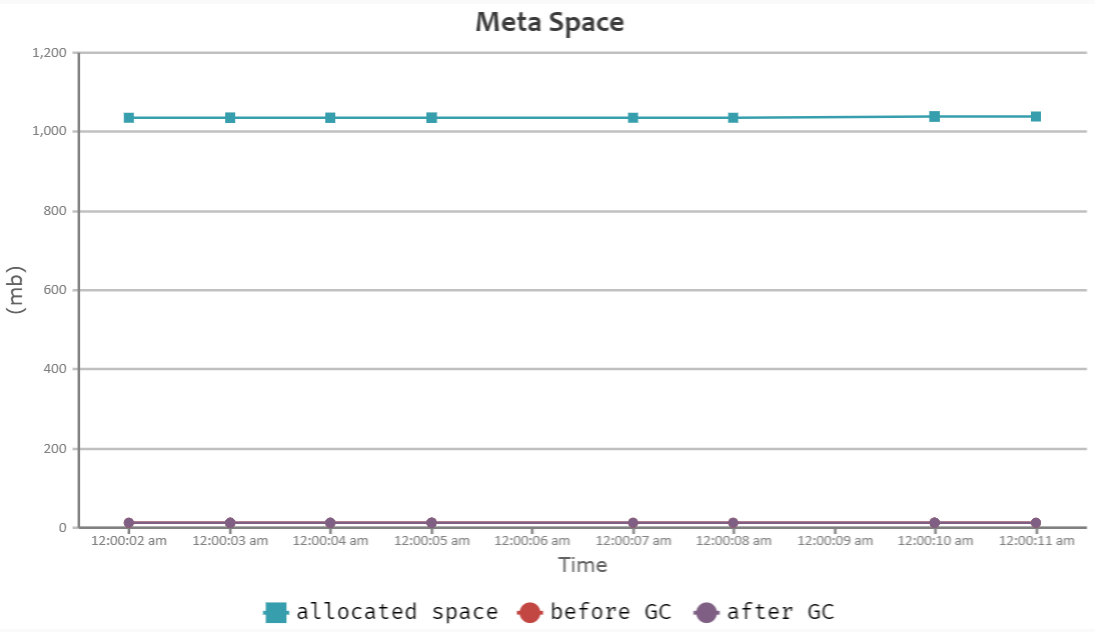
Young GC: 最少花费20ms, 最多花费876ms, 平均130ms.

Full GC: 两次, 350ms和440ms.

1. 每次GC前后heap各区域变化







分析: 可以看到, 引发GC的主要是Minor GC, 故可以增加Eden的大小来减少Minor GC.

### 用jstat命令行工具的-gc和-gcutil参数

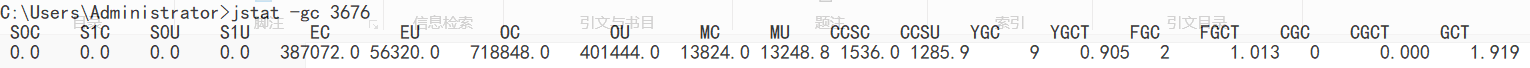


图 3 jstat -gc

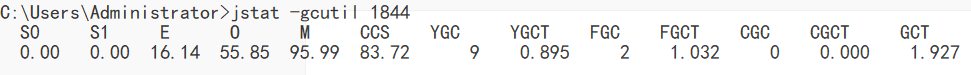


图 4 jstat -gcutil

### 使用jmap -heap命令行工具

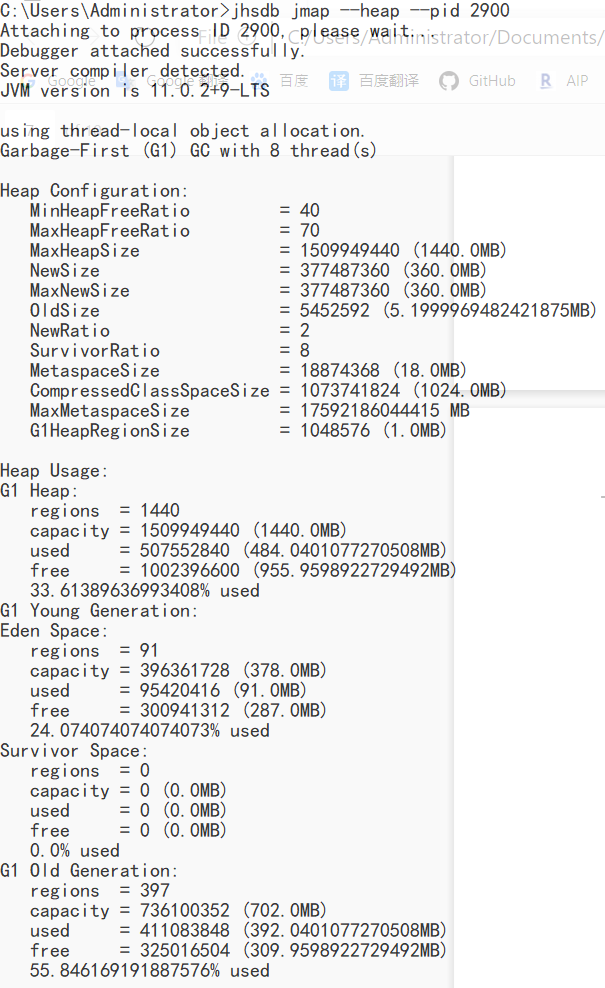


图 5 jmap -heap

### 使用jmap -clstats命令行工具

由于加载的类太多, 故不能完全列出.

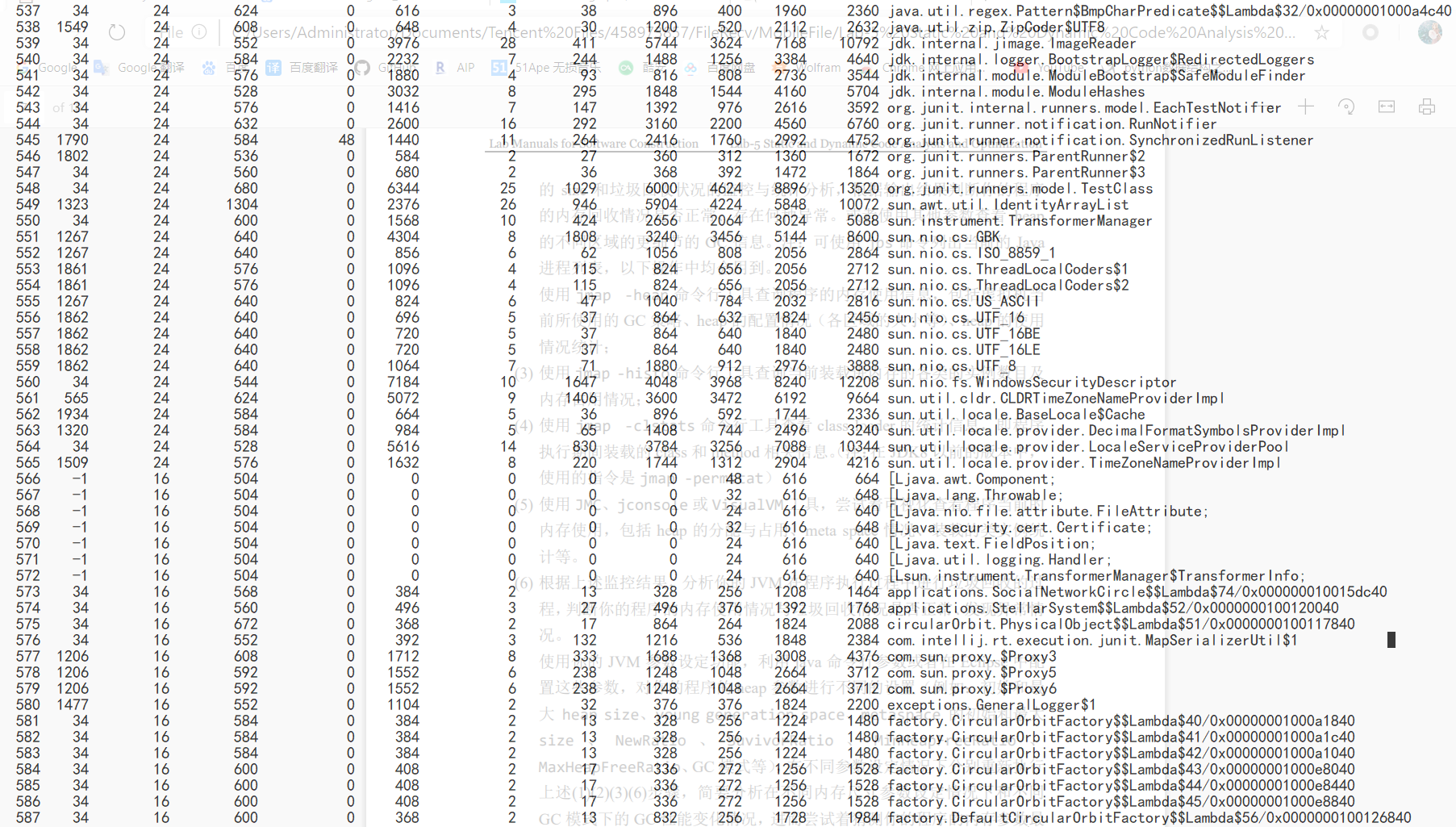


图 6 jmap -clstats

### 使用jmap -histo命令行工具

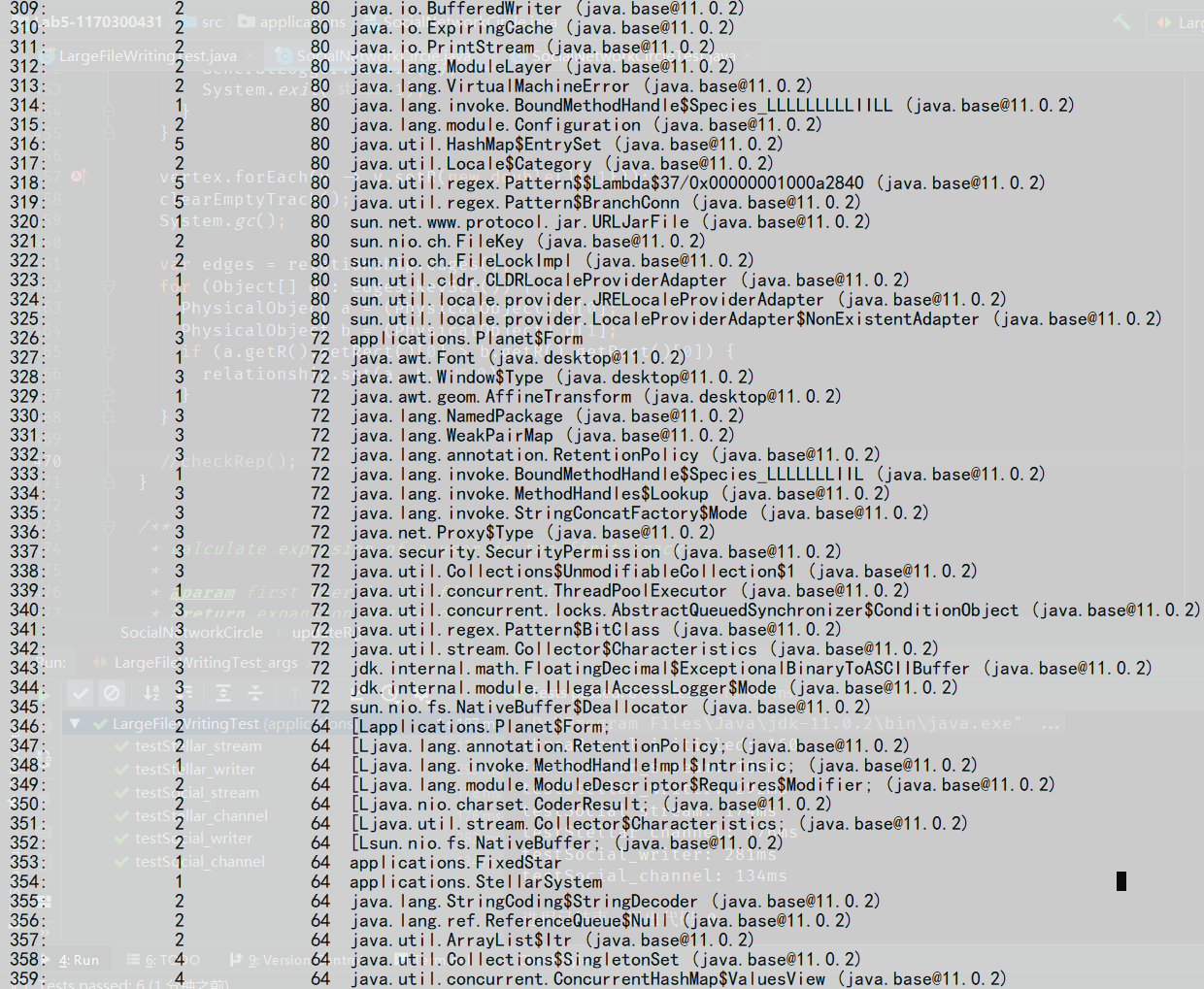


图 7 jmap -histo (不完全)

### 使用JMC/JFR、jconsole或VisualVM工具

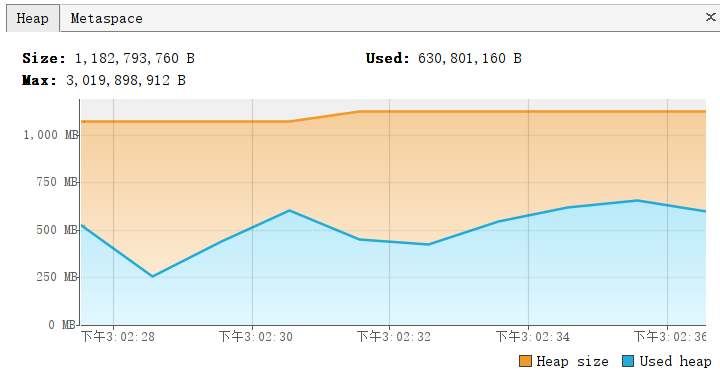


图 8 VisualVM Monitor-Heap

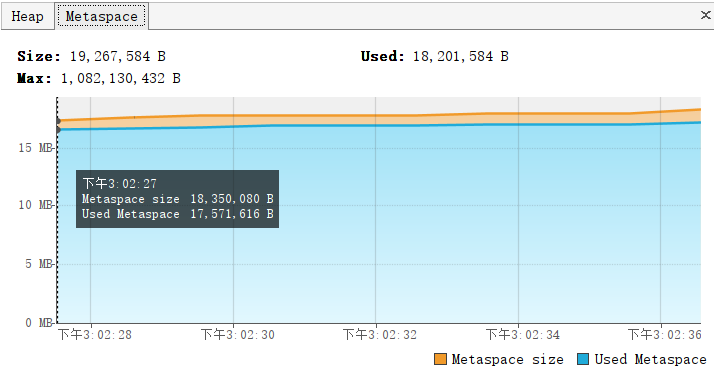


图 9 VisualVM Monitor Metaspace

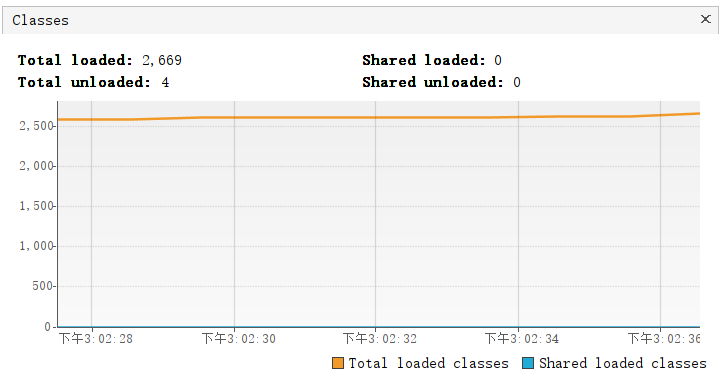


图 10 VisualVM Monitor Classes

### 分析垃圾回收过程

推断在程序启动的前段和中间部分(两次蓝色折线下降的部分)有比较明显的GC活动. 综合之前的GC日志分析, 在程序启动时由于加载类和初始化等等产生了大规模的GC活动; 在执行中段, 有稍次规模的GC(推测是手动的System.gc()和大规模的对象构建和集合操作).

### 配置JVM参数并发现优化的参数配置

有关运行参数.

尝试了三到四组heap参数, 时大致有五十次gc;

时大致八十次gc

时大致八十次gc

时十五到十六次gc (最终参数)

## Dynamic Program Profiling

### 使用JMC或VisualVM进行CPU Profiling

这里给一个已经优化过的截图...

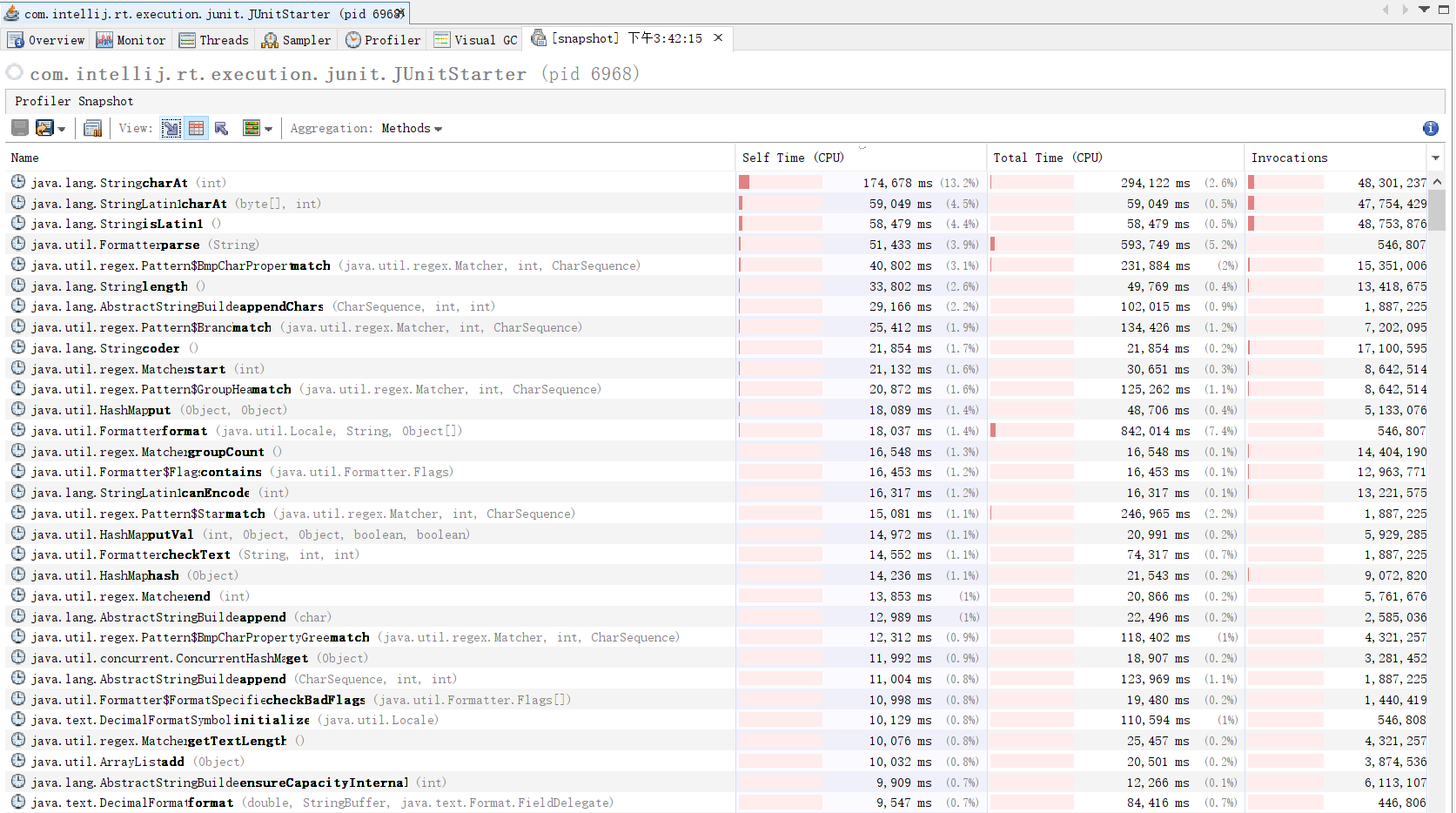


图 11 VisualVM CPU Profiling

优化过的部分包括,

* hash. 使用了大量的HashSet用于容器/去重等等. 计算重载过的hashCode就成为了一个性能瓶颈. 解决办法是在PhysicalObject内部用一个字段缓存该对象的hashCode.
* 字符串匹配. 字符串匹配是消耗比较大的操作. 尤其是在循环中调用, 比如String.split, pattern.match. 对于前者, 使用了apache开发包中的StringUtils.split函数, 在性能上相交JDK的版本有较大的提升. 对于后者, 一是尽量少用匹配, 二是将正则编译后保存起来, 避免重复编译造成的浪费.
* 算法上的优化. 显然以上的算法都是难以接受的. 在保留Lab4的某些原则的基础上, 用了比较多的空间换时间的策略来降低复杂度. 如果在算法上难以达到合适的复杂度(有些时候也难以达到满意的效率), 这里使用了多线程的方式来加快速度.

### 使用VisualVM进行Memory profiling

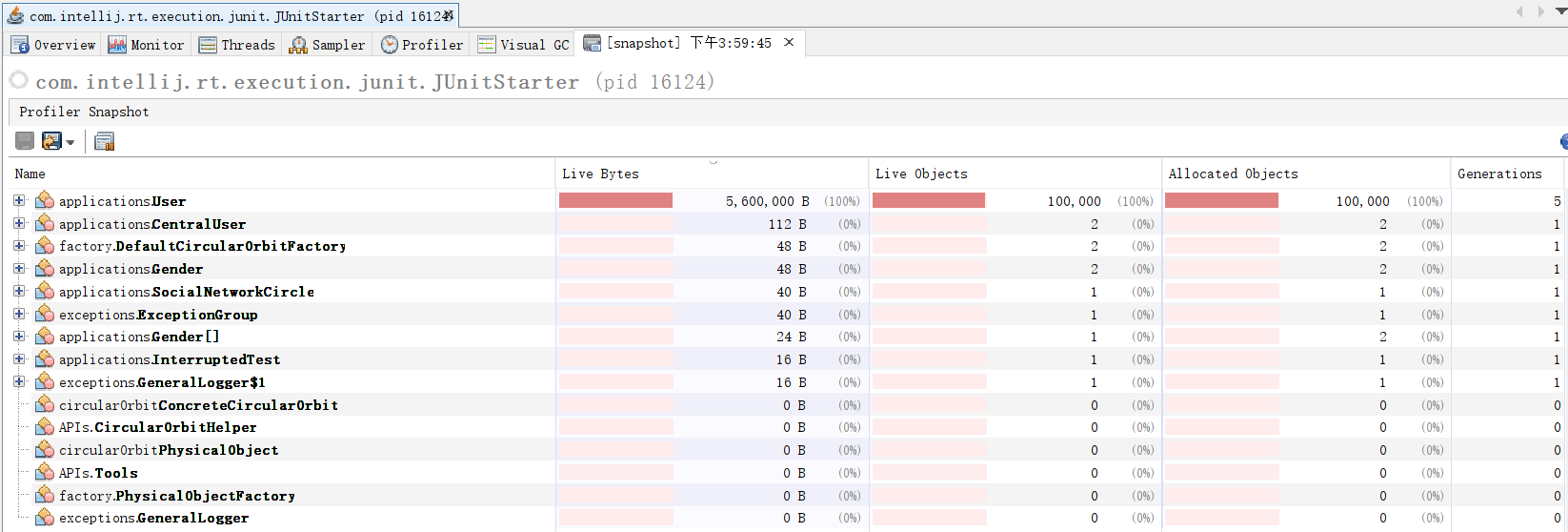


图 12 VisualVM Memory Profiling

有关内存优化, 主要关注的是减少对象构建, 即对象的复用. 参考3.5给出的设计模式, 主要用了一些对象的池化技术. 比如Track内部持有一个描述Track参数的数组. 优化前, 所有Track持有独立的double数组, 即便很多Track对象拥有相同的轨道参数. 现给Track一个double池, 初始化一些常见的轨道参数(如-1 1 2 3 4 5), 如果对象的轨道参数恰好是这些, 那么就复用池内的对象. 这样就减少了大量的double数组.

## Memory Dump Analysis and Performance Optimization

### 内存导出

在测试文件中构造CircularOrbit对象, 生成文本. 之后用while sleep的方式使程序中断(Junit运行时似乎不能从命令行输入...), 在VisualVM中heap dump, 在左侧找到dump file, 右击save as得到.hprof文件.

### 使用MAT分析内存导出文件

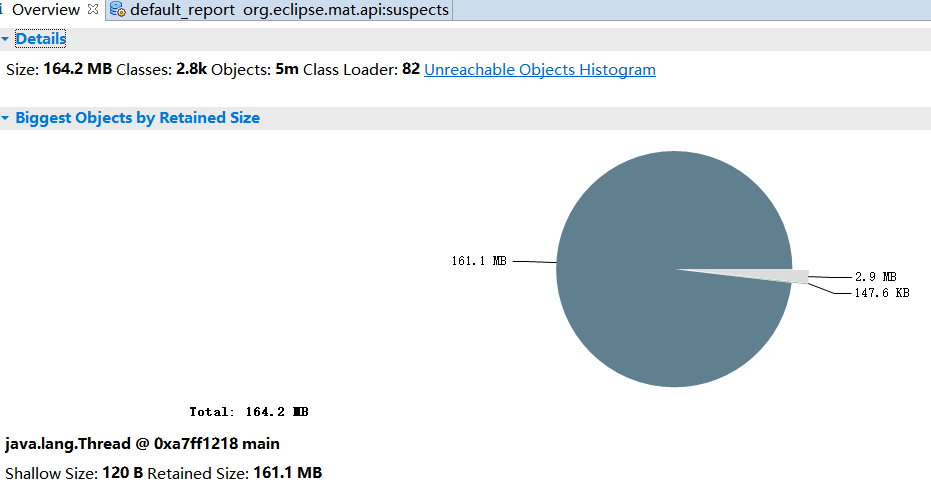


图 13 MAT Overview View

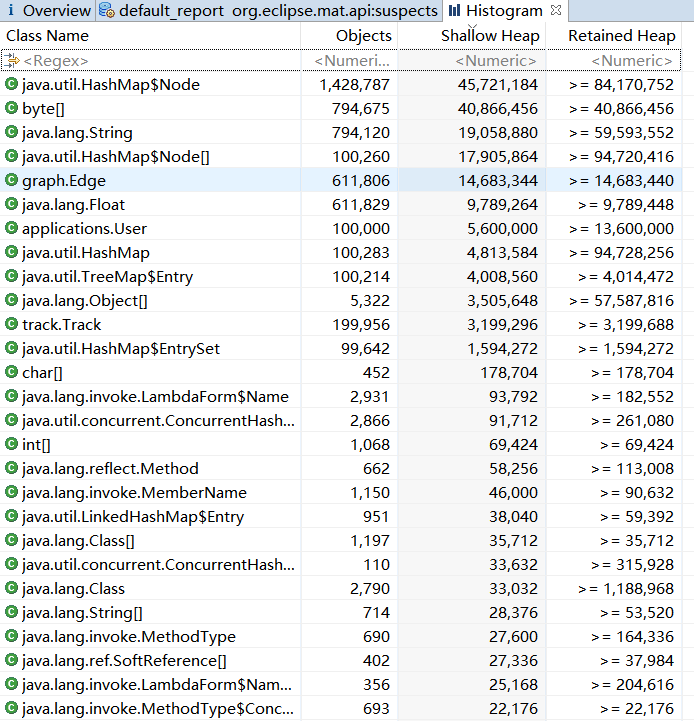


图 14 MAT Histogram View

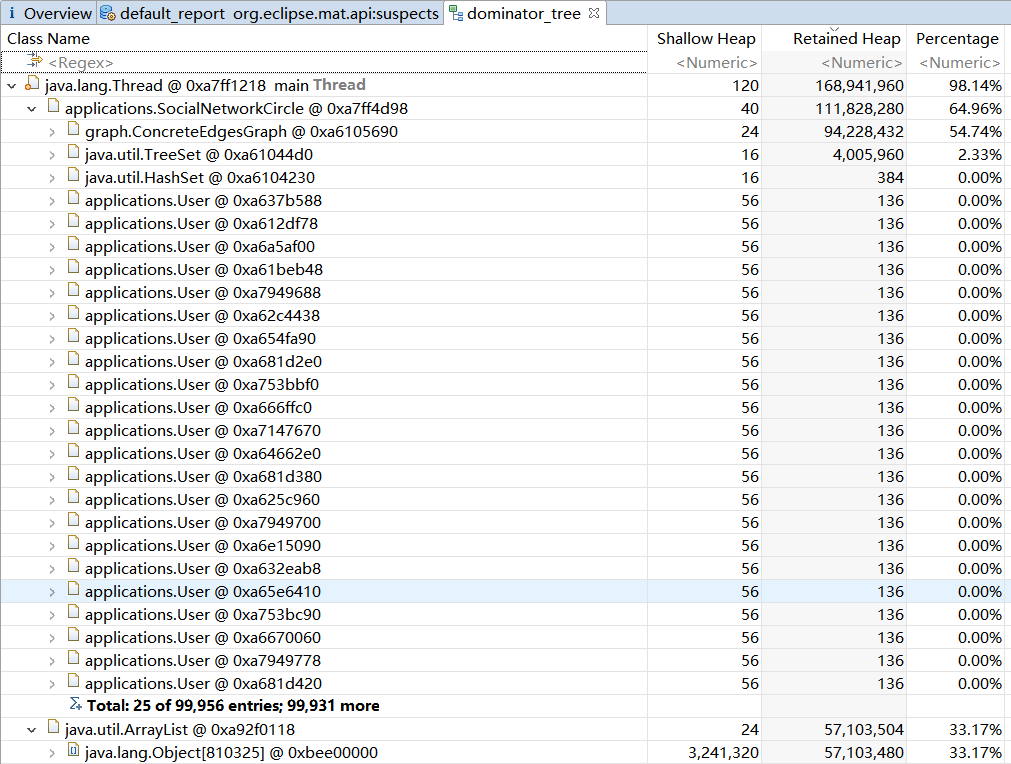


图 15 MAT Dominator Tree View

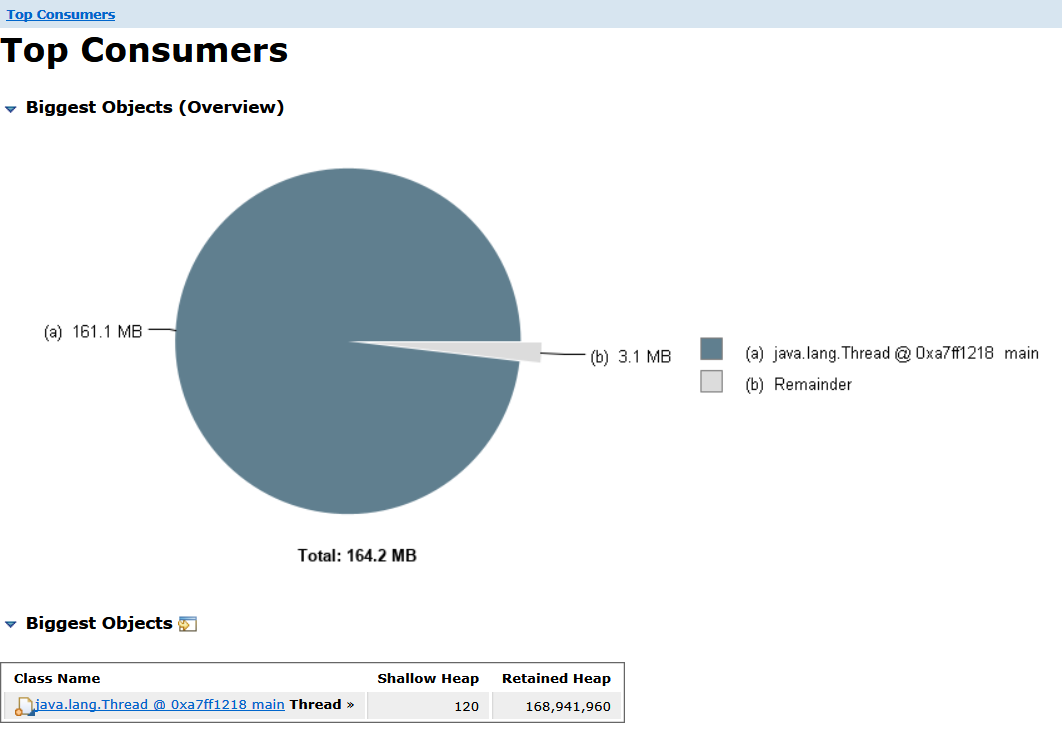


图 16 MAT Top Consumers View

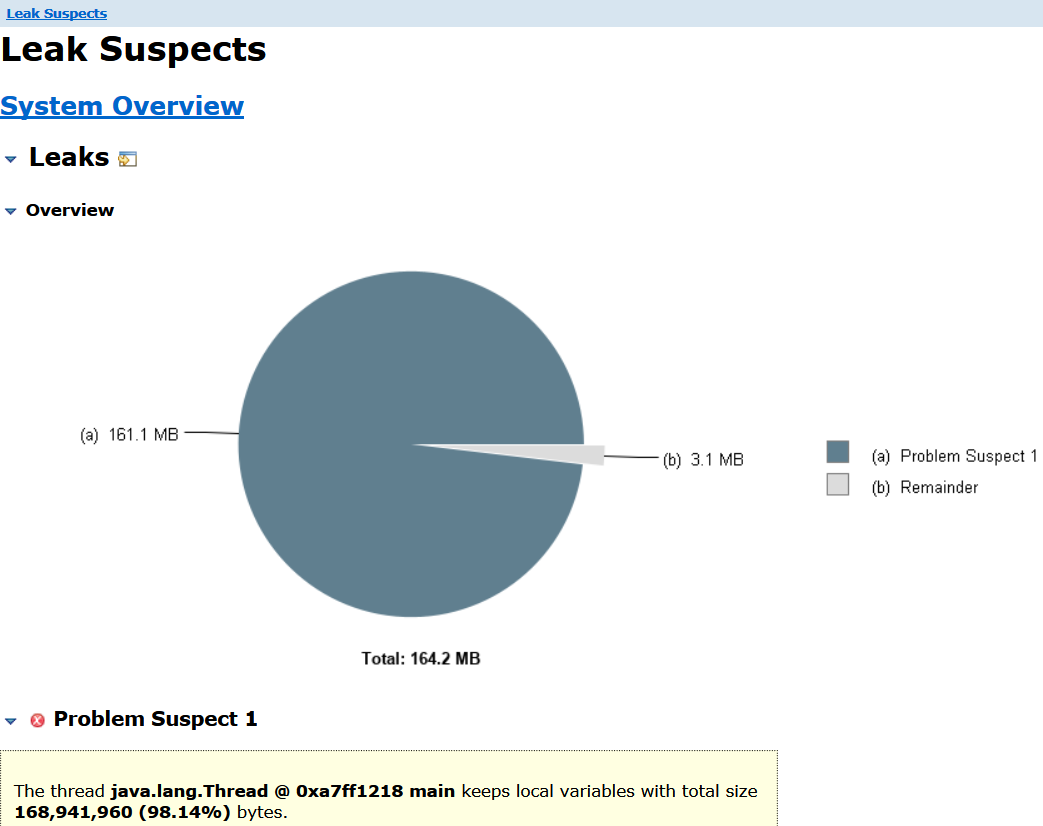


图 17 MAT Leak Suspects Report

### 发现热点/瓶颈并改进、改进前后的性能对比分析

上面给出的分析都是所有改进完成后的...

### 在MAT内使用OQL查询内存导出

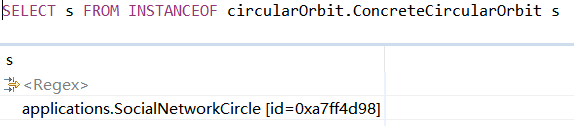


图 18 CircularOrbit的实例



图 19 长度大于256的string



图 20 size大于128的对象

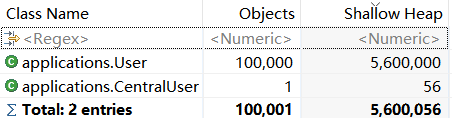


图 21 PhysicalObject的数量和总大小

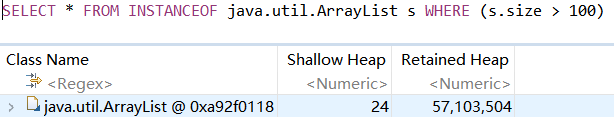


图 22 size大于100的ArrayList

### 观察jstack/jcmd导出程序运行时的调用栈



图 23 addTrack stack trace

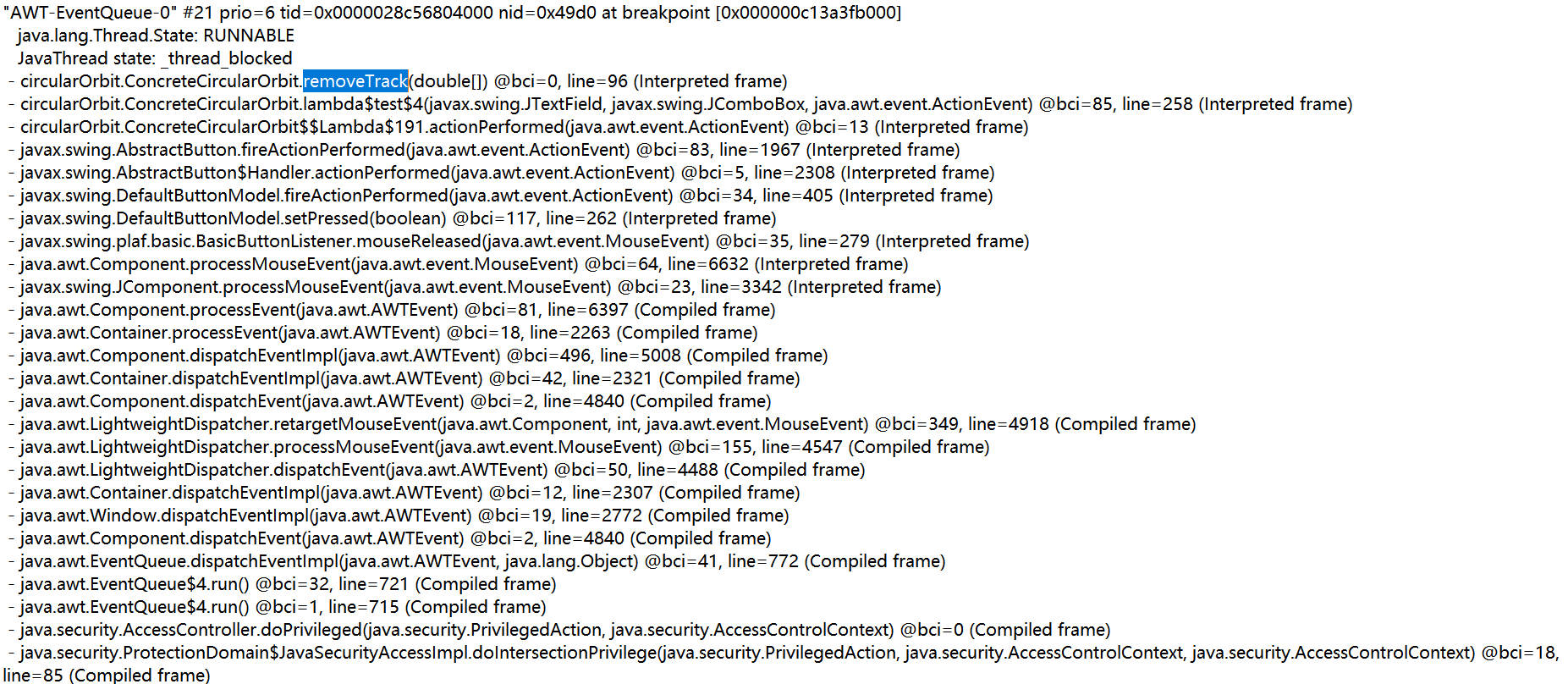


图 24 removeTracck stack trace



图 25 addObject stack trace



图 26 removeObject stack trace

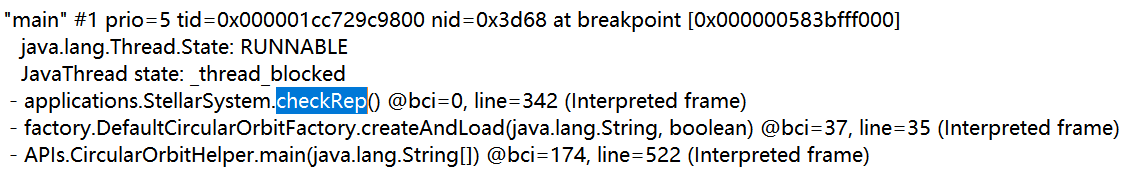


图 27 checkRep stack trace

### 使用设计模式进行代码性能优化

* 所有继承于PhysicalObject的对象均实现了clone
* CircularOrbitHelper类实现了单例模式(用于构建窗体)
* 有关池化, 使用了线程池和对象池(double[] 的对象池, 见3.4.2)

## Git仓库结构

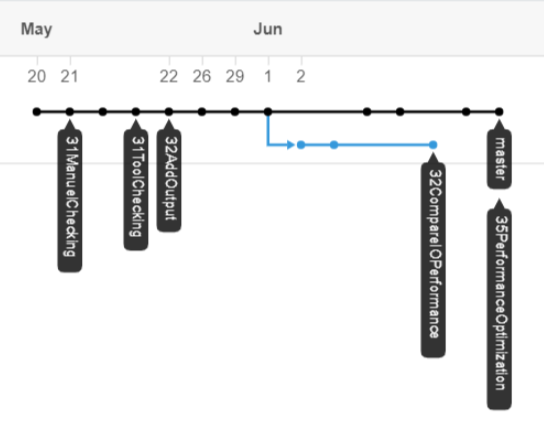


图 28 Github insights

# 实验进度记录

请使用表格方式记录你的进度情况，以超过半小时的连续编程时间为一行。

每次结束编程时，请向该表格中增加一行。不要事后胡乱填写。

不要嫌烦，该表格可帮助你汇总你在每个任务上付出的时间和精力，发现自己不擅长的任务，后续有意识的弥补。

|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 时间段 | 计划任务 | 实际完成情况 |
| 2019-5-20 |  | 环境配置 | 完成 |
| 2019-5-21 |  | 31ManuelChecking, 31ToolChecking | 完成 |
| 2019-5-22 |  | 32AddOutput | 完成 |
| 2019-5-26 |  | 多线程以及算法优化 | 测试出错 |
| 2019-5-29 |  | 多线程优化 | 完成 |
| 2019-6-1 |  | 改正bug | 完成? |

# 实验过程中遇到的困难与解决途径

|  |  |
| --- | --- |
| 遇到的难点 | 解决途径 |
| 使用的工具太老/和jdk不匹配/给的命令deperated或者用不了 | 搜索 |
| 是不是要移除某些Lab4中的规范 | 比如去重, 如果沿用Lab4中遍历查找的办法, 是根本行不通的. 但可以用一个HashSet来做到去重, 于是从T(1)->T(n), 而时间上是可接受的. 其实是希望尽可能地保留Lab4中的防御措施, 只有到不得已时再去掉. |
| 包名改不改 | 不改 |
| 线程同步和竞争问题 | 可以在单线程下取得结果来验证, 或者是多次运行多线程, 看解是否稳定. 在设计多线程任务时就要尽可能地考虑竞争问题, 而不是到调试的时候再解决. |

# 实验过程中收获的经验、教训、感想

## 实验过程中收获的经验和教训

## 针对以下方面的感受

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？

大多数时候, 可读性和效率往往有些冲突. 所以才有”不要一开始就优化代码”这种说法. 对一些性能要求高的部分, 应当尽可能地追求性能; 而其余部分其实可以保证可读性. 如果时间有限, 可读性更有利.

1. 诸如SpotBugs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？ 真的
2. 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？

不同的IO有不同的应用场景, 比如channel/buffer就比较适用于大规模读写, scanner比较方便易用之类...

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？

大家都有堆栈...区别好像在于, C对象(结构体, 或者C++类), 如果正常声明(局部且不是用new)是存在于栈中的, 而java的对象一律存在于堆中...这也涉及到指针和引用的问题,大多数时候java的这种方式更安全, 更方便.

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还有改进的空间吗？

C++在堆上资源释放是显式的(资源封装类和智能指针除外), 一方面, 这很麻烦, 另一方面是内存泄漏也很容易找到. JVM的内存回收完全不需要开发人员操心, 但隐蔽的内存泄漏(不再需要的内存引用)却很难发现.

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？ 是
2. 通过Memory Dump进行程序性能的分析，JMC/JFR、VisualVM和MAT这几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?

如果它们不是这么多这么杂这么难用的话 我还记得为C++做类似的工作是更简单更愉悦的事情(

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？ 用C++的时候是的
2. 关于本实验的工作量、难度、deadline。
3. 到目前为止，你对《软件构造》课程的意见与建议。

1. 使用参数-ea -Xlog:gc\*.\gc.log -Xms1080m -Xmx1440m -Xmn360m -XX:MetaspaceSize=18m 测得, 程序读入文件, 构建轨道系统, 生成文本并写入文件. [↑](#footnote-ref-1)