



# 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	肖行文
学号	1170300316
班号	1703003
电子邮件	xiaoxingwen@stu.hit.edu.cn
手机号码	13266573776

## 目录

1 实验目标概述 .....	1
2 实验环境配置 .....	1
3 实验过程 .....	1
3.1 Static Program Analysis .....	1
3.1.1 人工代码走查（walkthrough） .....	1
3.1.2 使用CheckStyle和SpotBugs进行静态代码分析 .....	2
3.2 Java I/O Optimization .....	2
3.2.1 多种I/O实现方式 .....	2
3.2.2 多种I/O实现方式的效率对比分析 .....	3
3.3 Java Memory Management and Garbage Collection (GC) .....	5
3.3.1 使用 <code>-verbose:gc</code> 参数 .....	5
3.3.2 用 <code>jstat</code> 命令行工具的 <code>-gc</code> 和 <code>-gcutil</code> 参数 .....	6
3.3.3 使用 <code>jmap -heap</code> 命令行工具 .....	6
3.3.4 使用 <code>jmap -clstats</code> 命令行工具 .....	8
3.3.5 使用 <code>jmap -permstat</code> 命令行工具 .....	8
3.3.6 使用JMC/JFR、 <code>jconsole</code> 或 <code>VisualVM</code> 工具 .....	8
3.3.7 分析垃圾回收过程 .....	10
3.3.8 配置JVM参数并发现优化的参数配置 .....	11
3.4 Dynamic Program Profiling .....	11
3.4.1 使用JMC或 <code>VisualVM</code> 进行CPU Profiling .....	11
3.4.2 使用 <code>VisualVM</code> 进行Memory profiling .....	12
3.5 Memory Dump Analysis and Performance Optimization .....	12
3.5.1 内存导出 .....	12
3.5.2 使用MAT分析内存导出文件 .....	12
3.5.3 发现热点/瓶颈并改进、改进前后的性能对比分析 .....	15
3.5.4 在MAT内使用OQL查询内存导出 .....	15
3.5.5 观察 <code>jstack/jcmd</code> 导出程序运行时的调用栈 .....	17

3.5.6 使用设计模式进行代码性能优化 .....	17
4 实验进度记录 .....	17
5 实验过程中遇到的困难与解决途径 .....	20
6 实验过程中收获的经验、教训、感想 .....	20
6.1 实验过程中收获的经验和教训 .....	20
6.2 针对以下方面的感受 .....	20

# 1 实验目标概述

- 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

# 2 实验环境配置

本程序用到的插件均为 Eclipse 自带的，无需额外安装。VisualVM 在官网  
上下载后直接可运行。

仓库地址：<https://github.com/ComputerScienceHIT/Lab5-1170300316>

# 3 实验过程

请仔细对照实验手册，针对每一项任务，在下面各节中记录你的实验过  
程、阐述你的设计思路和问题求解思路，可辅之以示意图或关键源代码加以说  
明（但千万不要把你的源代码全部粘贴过来！）。

## 3.1 Static Program Analysis

### 3.1.1 人工代码走查 (walkthrough)

1. Import 后类需要按照顺序字母表顺序排列，几乎都有错误。

```
import centralObject.Stellar;
import circularOrbit.CircularOrbit;
import circularOrbit.ConcreteCircularOrbit;
```

2. Import 后不能使用.\*形式引入类，而应该把每一项列出。

```
import java.nio.*;
```

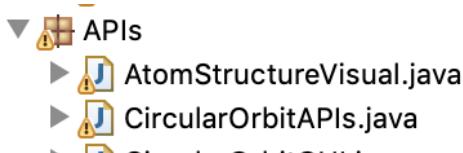
- 之前使用 Tab 键格式化代码，现改为两个空格。

```
208  
209      wh:  
210      !
```

- JavaDoc 注释第一行结尾未完结（没有输入句号）。

```
/**  
 * Read the file from the given string using the buffered reader  
 */
```

- Package 名中存在大写，应该改成小写



### 3.1.2 使用CheckStyle和SpotBugs进行静态代码分析

- 对 import 顺序进行修改，保证类名满足字典序。
- 将 import 后的.\*展开。
- 使用 Google 的 Formatter 工具进行格式化，Tab 键被替换成两个空格，参数过多时根据代码规范进行自动换行。
- 补全所有的 JavaDoc。
- 将可以改成小写的 Package 名改成小写。

SpotBugs 和 CheckStyle 都是静态代码分析工具。SpotBugs 是专门用来寻找潜在 bug 的工具，潜在的意思是错误的地方可能是 bug 也可能不是；而 CheckStyle 是用来检测代码规范的工具，代码规范不正确也不会导致程序出错。两个工具的功能不是同一个层面的，满足 CheckStyle 的要求也可能会在 SpotBugs 中找到错误。

## 3.2 Java I/O Optimization

### 3.2.1 多种I/O实现方式

程序原本使用无缓冲流的方法并于 java.io API 互操作：

```

try {
    System.out.println("Start reading from files.");
    reader = new BufferedReader(new InputStreamReader(new FileInputStream(fileName), "UTF-8"));
    String tempString = null;
}

```

后添加了使用缓冲流 I/O（理论最快读入文件方式）：

```

try {
    BufferedReader reader = Files.newBufferedReader(path, charset);
    String tempString = null;
}

```

以及 NIO 中 Files 类中的 I/O 方式：

```

java.util.List<String> stringList = new ArrayList();
try {
    stringList = Files.readAllLines(file);
}

```

设计接口 StrategySN.java 和 StrategySS.java，分别指 SocialNetwork 和 StellarSystem 中的 IO 策略。IOWithoutBuffer.java、IOByBufferedReader.java 和 IOByFiles.java 继承接口，分别代表上述三种 IO 方式，并完成实现代码。

在另一个文件 Context.java 中的以构造函数的方式引入 StrategySN 和 StrategySS，在类中用 Context(new StrategySS/SN) 方式分别对两个场景的 IO 方式进行操作。

```

public Context(StrategySS strategy) { public Context(StrategySN strategy) {
    this.strategyss = strategy;           this.strategysn = strategy;
}
}

```

### 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

用 System.currentTimeMillis() 获得开始 I/O 前的时间和结束的时间，两个时间作差获得 I/O 时间。

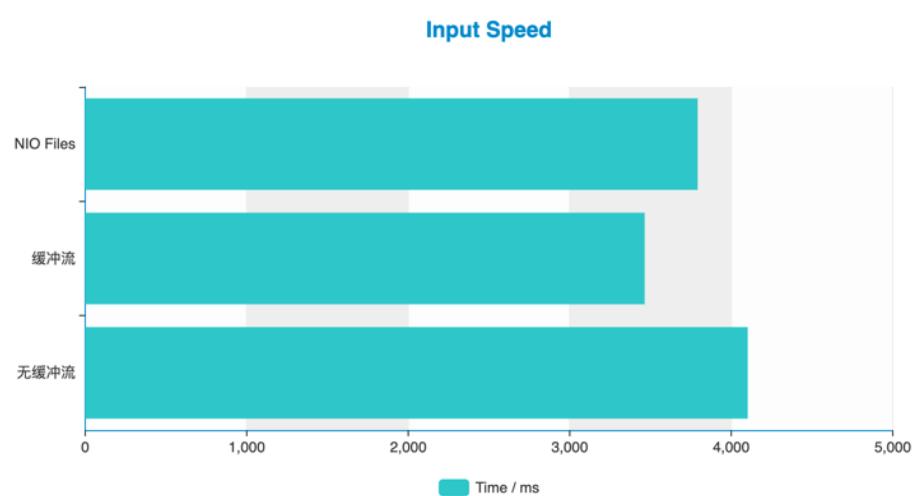
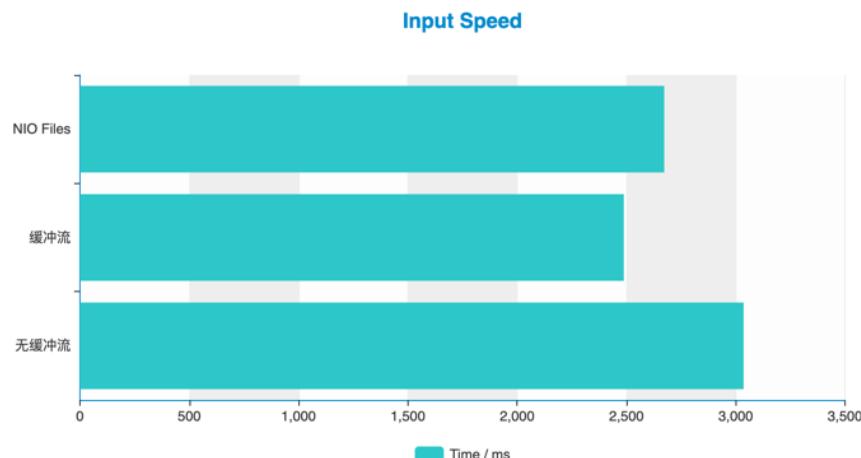
计算 Input 时间时除了读取文件，还包括新建对象的时间，不包括任何操作的时间。Output 仅输出从已知改变后的对象到文件的时间。

#### 2. 表格方式对比不同 I/O 的性能

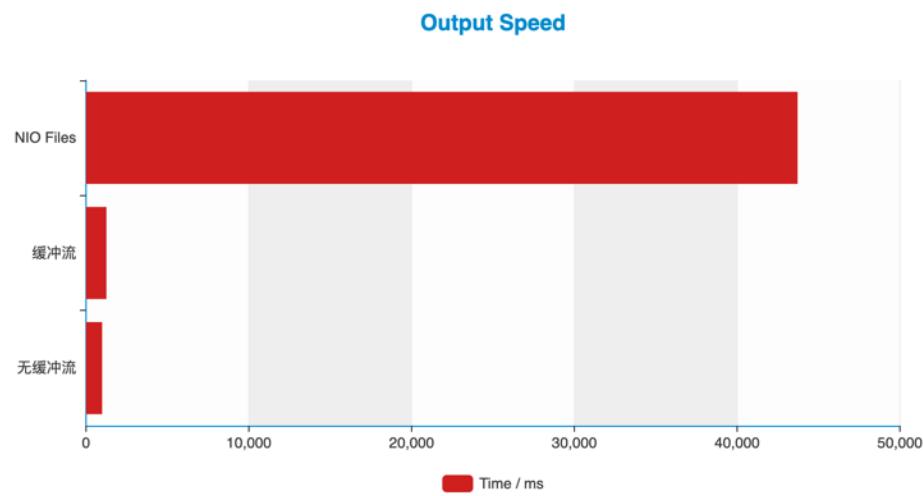
		StellarSystem.txt	SocialNetwork.txt
无缓冲流	读文件	3036ms	4102ms
	写文件	987ms	1350ms
缓冲流	读文件	2488ms	3464ms
	写文件	1249ms	1483ms
NIO Files	读文件	2673ms	3792ms
	写文件	43713ms	46659ms

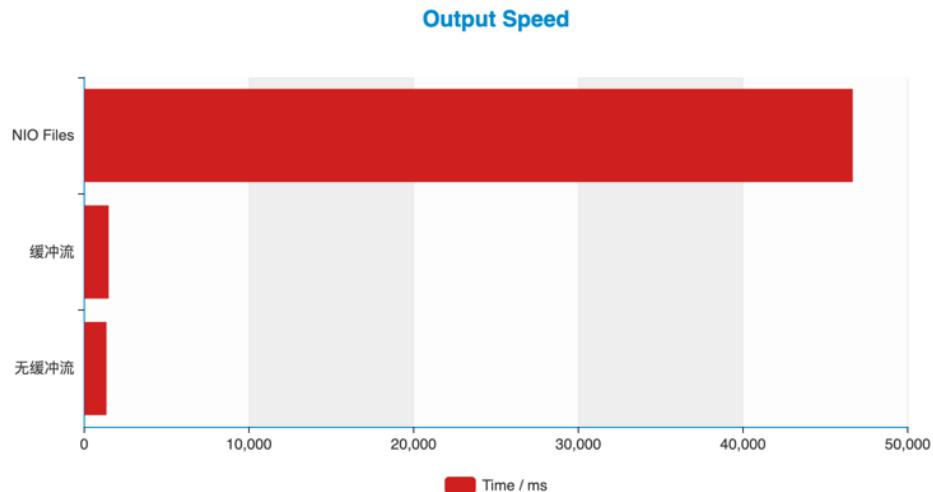
#### 3. 图形对比不同 I/O 的性能

前者是 File1 中的 Input 速度，后者是 File2 中的 Input 速度：



前者是 File1 中的 Output 速度，后者是 File2 中的 Output 速度：





### 3.3 Java Memory Management and Garbage Collection (GC)

#### 3.3.1 使用-verbose:gc参数

由于在 Java 8 之后 -XX:+PrintGCDetails 已经被停用，而改用 -Xlog:gc\* 查看 GC 详情，故本实验用 -Xlog:gc\* 查看。以下是（部分）打印内容：

```
[0.031s][info ][gc,heap] Heap region size: 1M
[0.033s][info ][gc ] Using G1
[0.033s][info ][gc,heap,coops] Heap address: 0x00000078000000, size: 2048 MB, CompressedOops mode: Zero based
[1.332s][info ][gc,start ] GC(0) Pause Young (Normal) (G1 Evacuation Pause)
[1.332s][info ][gc,task ] GC(0) Using 4 workers of 4 for evacuation
[1.338s][info ][gc,phases ] GC(0) Pre Evacuate Collection Set: 0.0ms
[1.338s][info ][gc,phases ] GC(0) Evacuate Collection Set: 5.1ms
[1.338s][info ][gc,phases ] GC(0) Post Evacuate Collection Set: 0.2ms
[1.338s][info ][gc,phases ] GC(0) Other: 0.3ms
[1.338s][info ][gc,heap ] GC(0) Eden regions: 14->0(74)
[1.338s][info ][gc,heap ] GC(0) Survivor regions: 0->2(2)
[1.338s][info ][gc,heap ] GC(0) Old regions: 0->1
[1.338s][info ][gc,heap ] GC(0) Humongous regions: 0->0
[1.338s][info ][gc,metaspace ] GC(0) Metaspace: 15145K->15145K(1062912K)
[1.338s][info ][gc ] GC(0) Pause Young (Normal) (G1 Evacuation Pause) 14M->2M(128M) 6.048ms
[1.338s][info ][gc,cpu ] GC(0) User=0.01s Sys=0.00s Real=0.01s
[5.159s][info ][gc,start ] GC(1) Pause Young (Normal) (G1 Evacuation Pause)
[5.159s][info ][gc,task ] GC(1) Using 4 workers of 4 for evacuation
[5.296s][info ][gc,phases ] GC(1) Pre Evacuate Collection Set: 0.0ms
[5.296s][info ][gc,phases ] GC(1) Evacuate Collection Set: 135.7ms
[5.296s][info ][gc,phases ] GC(1) Post Evacuate Collection Set: 0.6ms
[5.296s][info ][gc,phases ] GC(1) Other: 0.1ms
[5.296s][info ][gc,heap ] GC(1) Eden regions: 74->0(1)
[5.296s][info ][gc,heap ] GC(1) Survivor regions: 2->10(10)
[5.296s][info ][gc,heap ] GC(1) Old regions: 1->59
[5.296s][info ][gc,heap ] GC(1) Humongous regions: 18->18
[5.296s][info ][gc,metaspace ] GC(1) Metaspace: 19312K->19312K(1067008K)
[5.296s][info ][gc ] GC(1) Pause Young (Normal) (G1 Evacuation Pause) 94M->86M(163M) 136.515ms
[5.296s][info ][gc,cpu ] GC(1) User=0.25s Sys=0.04s Real=0.13s
[5.298s][info ][gc,start ] GC(2) Pause Young (Concurrent Start) (G1 Evacuation Pause)
[5.298s][info ][gc,task ] GC(2) Using 4 workers of 4 for evacuation
[5.313s][info ][gc,phases ] GC(2) Pre Evacuate Collection Set: 0.0ms
[5.313s][info ][gc,phases ] GC(2) Evacuate Collection Set: 14.7ms
[5.313s][info ][gc,phases ] GC(2) Post Evacuate Collection Set: 0.3ms
[5.313s][info ][gc,phases ] GC(2) Other: 0.3ms
[5.313s][info ][gc,heap ] GC(2) Eden regions: 1->0(7)
[5.313s][info ][gc,heap ] GC(2) Survivor regions: 10->1(2)
[5.313s][info ][gc,heap ] GC(2) Old regions: 59->70
[5.313s][info ][gc,heap ] GC(2) Humongous regions: 18->18
[5.313s][info ][gc,metaspace ] GC(2) Metaspace: 19312K->19312K(1067008K)
```

GC 是 G1 机制，Java 8 中是 CMS 机制，现就 G1 机制进行分析。在程序运行早期，对 Eden Region 的操作多，对 Old Region 的操作少，随着程序的运

行，有其是读入文件完成，Old Region 数量很大，Eden Region 数量相比之下很少，但对 Eden Region 的操作仍然非常频繁。

```
[12.881s][info][gc,heap] GC(36) Eden regions: 163->0(68)
[12.881s][info][gc,heap] GC(36) Survivor regions: 21->23(23)
[12.881s][info][gc,heap] GC(36) Old regions: 361->413
[12.881s][info][gc,heap] GC(36) Humongous regions: 175->175
[12.881s][info][gc,metaspace] GC(36) Metaspace: 20695K->20695K(1069056K)
[12.881s][info][gc] GC(36) Pause Young (Prepare Mixed) (G1 Evacuation Pause) 720M->611M(1822M) 153.589ms
[12.881s][info][gc,cpu] GC(36) User=0.30s Sys=0.02s Real=0.15s
[12.914s][info][gc,marking] GC(35) Concurrent Cleanup for Next Mark 207.438ms
[12.914s][info][gc] GC(35) Concurrent Cycle 1511.707ms
[13.124s][info][gc,start] GC(37) Pause Young (Mixed) (G1 Evacuation Pause)
[13.124s][info][gc,task] GC(37) Using 4 workers of 4 for evacuation
[13.205s][info][gc,phases] GC(37) Pre Evacuate Collection Set: 0.1ms
[13.205s][info][gc,phases] GC(37) Evacuate Collection Set: 80.2ms
[13.205s][info][gc,phases] GC(37) Post Evacuate Collection Set: 0.6ms
[13.205s][info][gc,phases] GC(37) Other: 0.4ms
[13.205s][info][gc,heap] GC(37) Eden regions: 68->0(123)
[13.205s][info][gc,heap] GC(37) Survivor regions: 23->12(12)
[13.205s][info][gc,heap] GC(37) Old regions: 413->412
[13.205s][info][gc,heap] GC(37) Humongous regions: 212->212
[13.205s][info][gc,metaspace] GC(37) Metaspace: 20734K->20734K(1069056K)
[13.205s][info][gc] GC(37) Pause Young (Mixed) (G1 Evacuation Pause) 716M->635M(1822M) 81.367ms
[13.205s][info][gc,cpu] GC(37) User=0.18s Sys=0.01s Real=0.08s
LENGTH320000
[14.038s][info][gc,heap,exit] Heap
[14.038s][info][gc,heap,exit] garbage-first heap total 1865728K, used 758272K [0x0000000780000000, 0x0000000800000000)
[14.038s][info][gc,heap,exit] region size 1024K, 111 young (113664K), 12 survivors (12288K)
[14.038s][info][gc,heap,exit] Metaspace used 20797K, capacity 21235K, committed 21504K, reserved 1069056K
[14.038s][info][gc,heap,exit] class space used 2232K, capacity 2449K, committed 2560K, reserved 1048576K
```

由此可以看出，对年轻代（Eden Region 和 Survivor Region）的操作快于老年代。老年代增多时，堆空间会明显变大，年轻代总是在不断地被回收，不会明显地影响堆空间。

### 3.3.2 用jstat命令行工具的-gc和-gcutil参数

使用 jstat 的 -gc 命令的结果：

```
TonyShawdeMacBook-Pro:Lab5-1170300316 tony$ jstat -gc 92743
S0C      S1C      S0U      S1U      EC      EU      OC      OU
0.0     34816.0    0.0     34816.0   274432.0  207872.0  1667072.0  759296.0
```

MC	MU	CCSC	CCSU	YGC	YGCT	FGC	FGCT	CGC	CGCT	GCT	
19456.0	18790.5	2304.0	2018.0		53	3.216	0	0.000	14	0.028	3.244

使用 jstat 的 -gcutil 参数：

YGCT	FGC	FGCT	CGC	CGCT	GCT
2.432	0	0.000	8	0.019	2.451

S0	S1	E	0	M	CCS	YGC
0.00	100.00	34.97	41.80	95.61	87.21	45

### 3.3.3 使用jmap -heap命令行工具

使用 jmap -heap 命令结果如下：

```
Debugger attached successfully.
Server compiler detected.
JVM version is 11.0.2+9-LTS

using thread-local object allocation.
Garbage-First (G1) GC with 4 thread(s)

Heap Configuration:
  MinHeapFreeRatio      = 40
  MaxHeapFreeRatio      = 70
  MaxHeapSize           = 2147483648 (2048.0MB)
  NewSize               = 1363144 (1.2999954223632812MB)
  MaxNewSize             = 1287651328 (1228.0MB)
  OldSize               = 5452592 (5.1999969482421875MB)
  NewRatio              = 2
  SurvivorRatio          = 8
  MetaspaceSize          = 21807104 (20.796875MB)
  CompressedClassSpaceSize = 1073741824 (1024.0MB)
  MaxMetaspaceSize       = 17592186044415 MB
  G1HeapRegionSize        = 1048576 (1.0MB)

Heap Usage:
G1 Heap:
  regions   = 2048
  capacity   = 2147483648 (2048.0MB)
  used       = 760741888 (725.5MB)
  free        = 1386741760 (1322.5MB)
  35.4248046875% used

G1 Young Generation:
Eden Space:
  regions   = 50
  capacity   = 84934656 (81.0MB)
  used       = 52428800 (50.0MB)
  free        = 32505856 (31.0MB)
  61.72839506172839% used

Survivor Space:
  regions   = 13
  capacity   = 13631488 (13.0MB)
  used       = 13631488 (13.0MB)
  free        = 0 (0.0MB)
  100.0% used

G1 Old Generation:
  regions   = 663
  capacity   = 1675624448 (1598.0MB)
  used       = 693633024 (661.5MB)
  free        = 981991424 (936.5MB)
  41.39549436795995% used
```

### 3.3.4 使用jmap -clstats命令行工具

```

Attaching to process ID 92990, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 11.0.2+9-LTS
finding class loader instances ..done.
computing per loader stat ..done.
please wait.. computing liveness.....done.
class_loader      classes bytes   parent_loader    alive? type
<bootstrap>      2339    7021067    null      live     <internal>
0x00000000787f94fa8      0      0      0x00000000787f94810      live     jdk/internal/loader/ClassLoaders$PlatformClassLoader@0x00000000800010e30
0x00000000787f98b58      4    14289    0x00000000787f94fa8      live     jdk/internal/loader/ClassLoaders$AppClassLoader@0x000000008000109b8
0x00000000787f94810      0      0      null      live     jdk/internal/loader/ClassLoaders$BootClassLoader@0x000000008000353b0

total = 4      2343    7035356      N/A      alive=4, dead=0      N/A

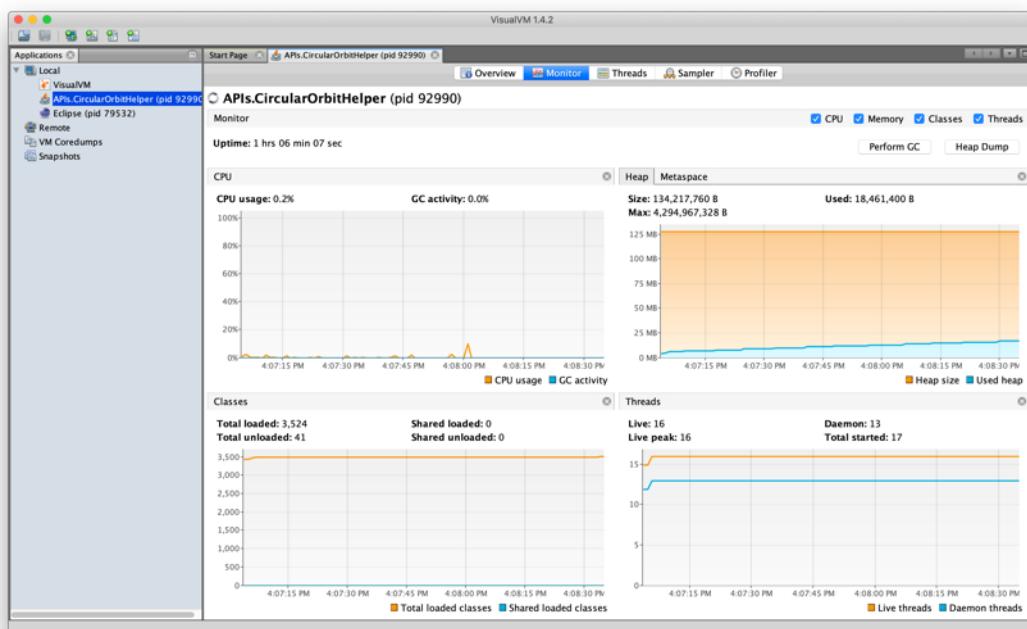
```

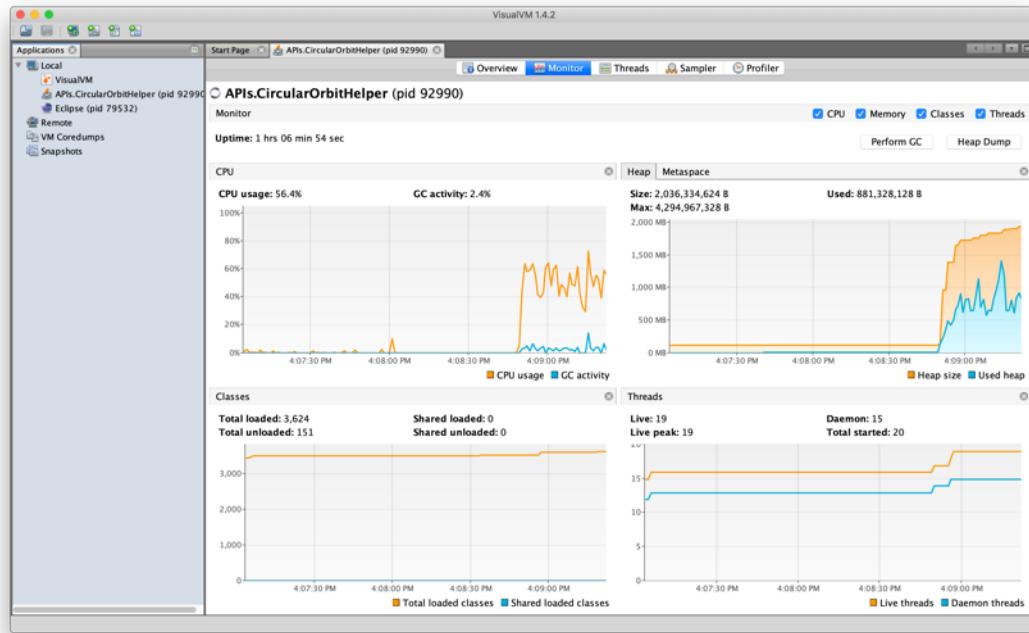
### 3.3.5 使用jmap -permstat命令行工具

使用 JDK11，已用 jmap -clstats 代替。

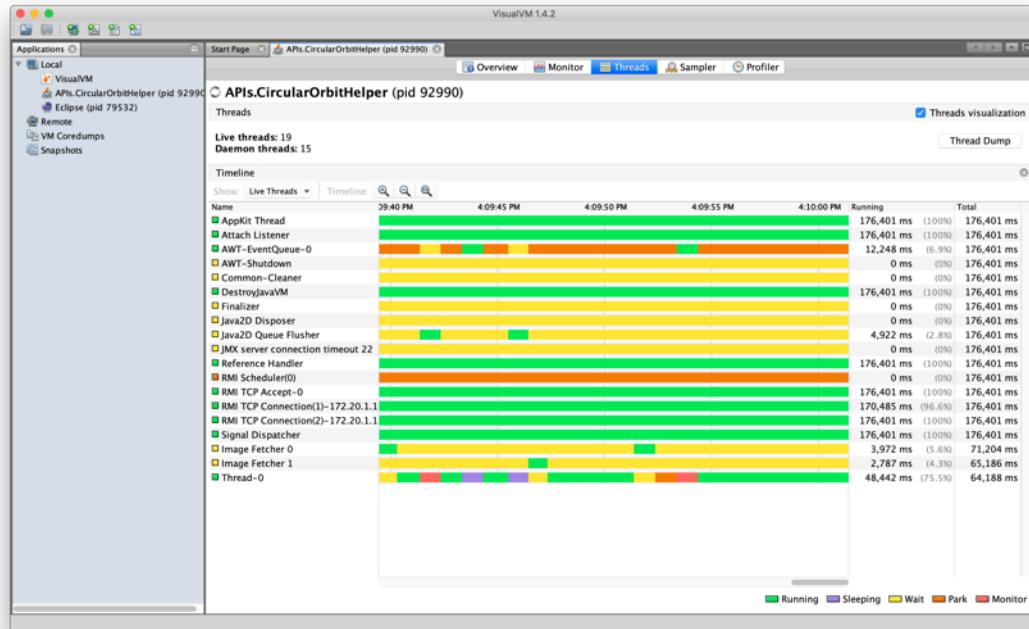
### 3.3.6 使用JMC/JFR、jconsole或VisualVM工具

Monitor An Application:

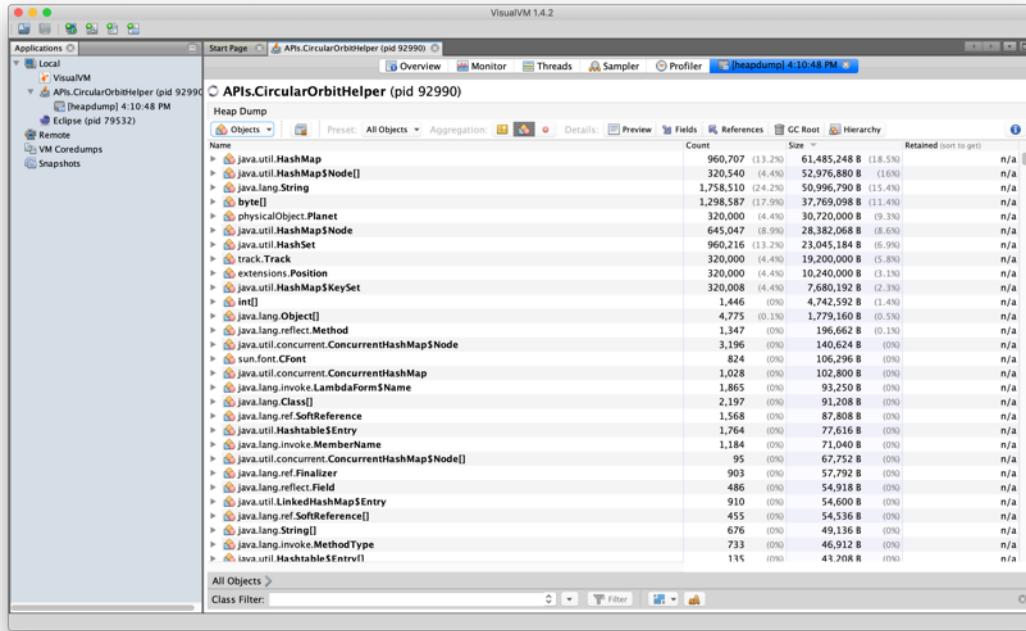




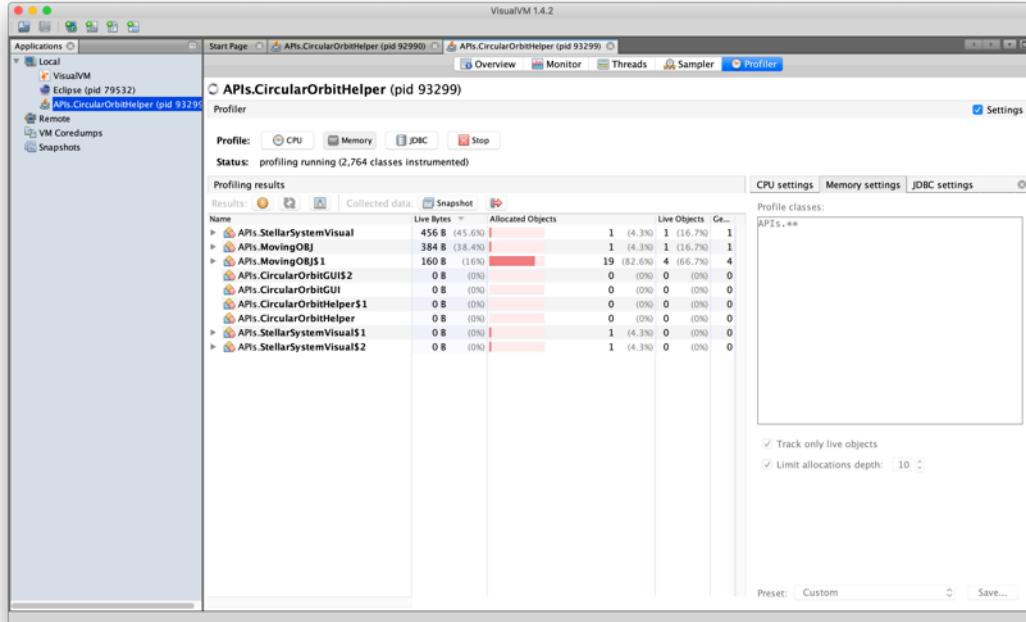
查看线程活动：



查看 heap 的分配和使用：

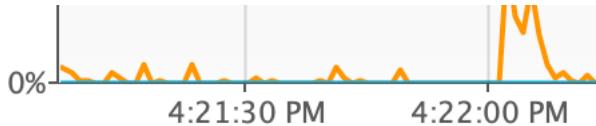


查看内存使用状况：

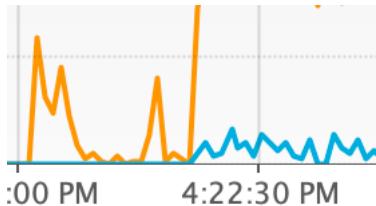


### 3.3.7 分析垃圾回收过程

程序进入场景选择窗口时，堆空间远大于使用的堆空间，垃圾回收量很小。图中蓝线是垃圾回收量，可以看出一直贴着 0%。



读入文件时，使用的堆空间增大，CPU 使用量上升，同时垃圾回收频率也上升。



文件读入后，由于动画线程要不断刷新，垃圾回收频率也呈周期性变化。

关于内存占用，主要内存用在动画线程刷新以及读入文件的过程中，占比一半以上，当读入文件完成后，动画线程刷新成为内存的主要消耗。未出现异常的内存使用。

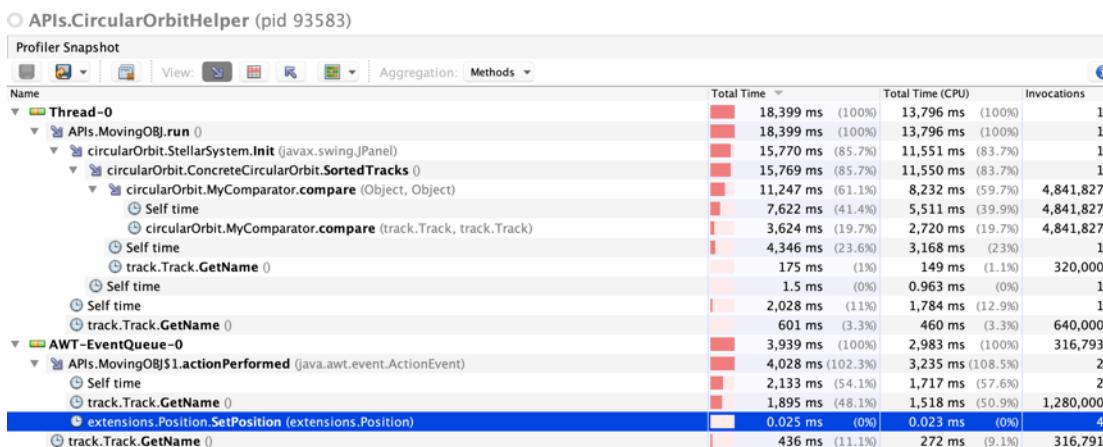
### 3.3.8 配置JVM参数并发现优化的参数配置

年轻代尽可能设置到最大，这样年轻代手机发生的频率是最小的，同时可以减少老年代的对象。

将老年代大小分成原有老年代大小的 60%，80%，100%，120%进行尝试，发现在原老年代大小的 80% 时 GC 最快。老年代过大时会早场高频率回收，过大时堆过大，CMS 对过大的堆回收十分不擅长。

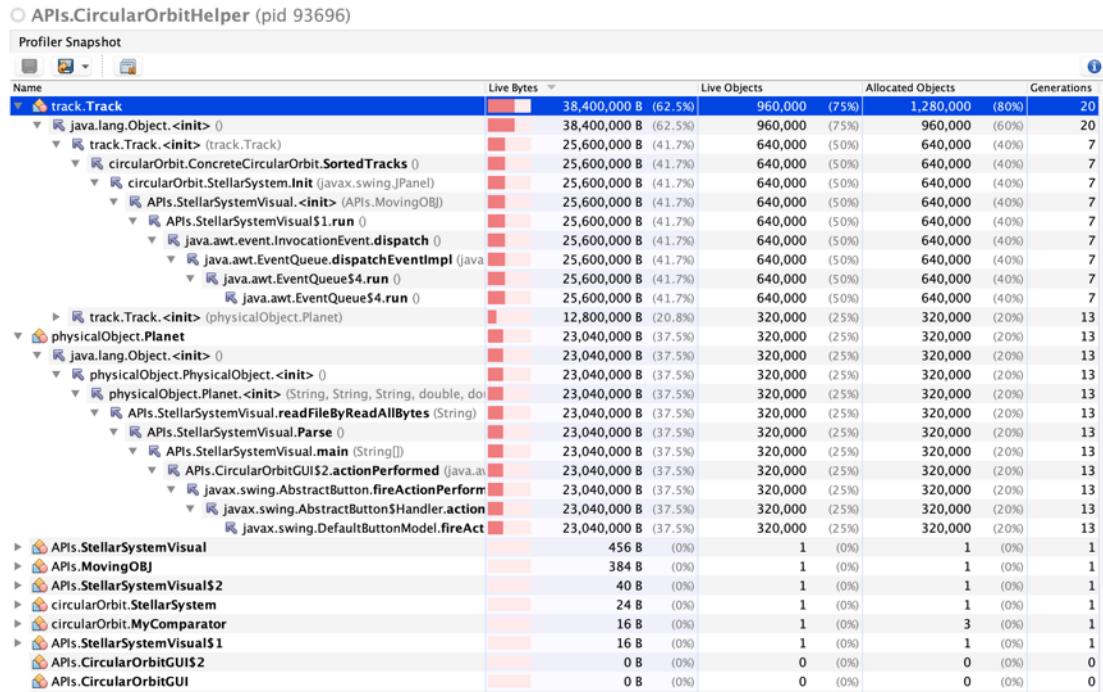
## 3.4 Dynamic Program Profiling

### 3.4.1 使用JMC或VisualVM进行CPU Profiling



绝大部分的 CPU 资源被轨道排序占用。因为要比较的轨道数量太多，所以资源消耗占比 85.7%。其它剩余的资源很多是调用存储的轨道/Physical Object 属性。CPU 资源消耗合理。

### 3.4.2 使用VisualVM进行Memory profiling



绝大部分的内存资源被轨道操作占用，其次是 PhysicalObject 的操作。考虑到存取 PhysicalObject 和轨道的操作数量多，内存资源消耗合理。

## 3.5 Memory Dump Analysis and Performance Optimization

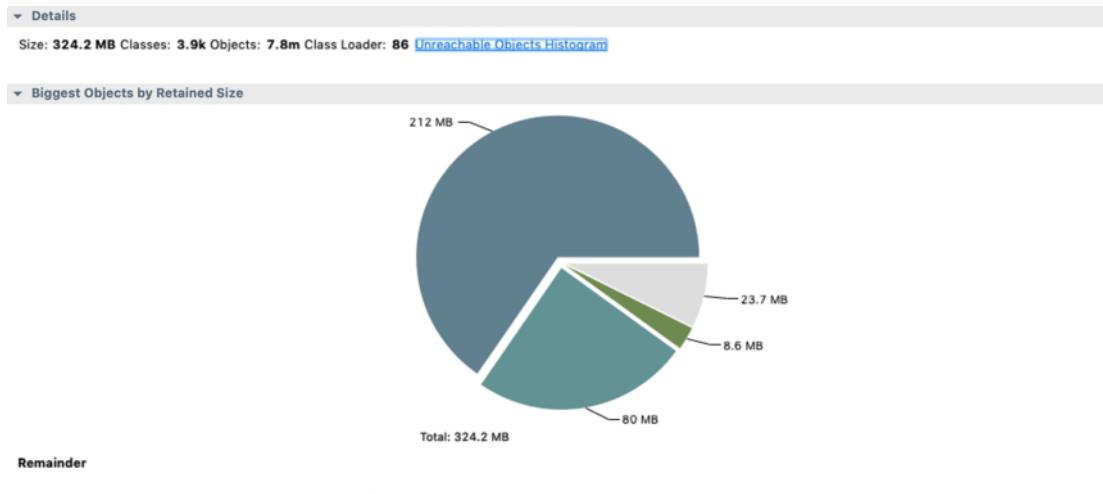
### 3.5.1 内存导出

用 VisualVM 监测程序，点击 Monitor 中的 Heap Dump，左侧会出现 Heap Dump 栏，右击可保存.hprof 文件。

存储的.hprof 放置在 src/hprof 中。

### 3.5.2 使用MAT分析内存导出文件

Overview:

**Histogram:**

Class Name	Objects	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
C int[]	1,647	88,349,968	
C java.lang.String	2,256,396	54,153,504	
C java.util.HashMap	960,738	46,115,424	
C byte[]	1,297,031	31,679,896	
C java.util.HashMap\$Node[]	320,585	27,775,992	
C physicalObject.Planet	320,000	23,040,000	
C java.util.HashMap\$Node	645,318	20,650,176	
C java.util.HashSet	960,218	15,363,488	
C track.Track	320,000	12,800,000	
C extensions.Position	320,002	10,240,064	
C java.util.HashMap\$KeySet	320,010	5,120,160	
C java.lang.Object[]	6,190	2,928,968	
C java.util.concurrent.ConcurrentHashMap\$Node	3,233	103,456	
C java.lang.invoke.LambdaForm\$Name	3,158	101,056	
C sun.font.CFont	824	72,512	
C java.util.concurrent.ConcurrentHashMap	1,038	66,432	
C java.util.Hashtable\$Entry	1,877	60,064	
C java.lang.reflect.Method	646	56,848	
C java.lang.invoke.MemberName	1,295	51,800	
C java.lang.Class	3,914	47,888	
C char[]	51	46,024	
C java.lang.ref.SoftReference	1,102	44,080	
C java.lang.Class[]	1,243	37,608	
C java.util.LinkedHashMap\$Entry	924	36,960	
C java.lang.ref.Finalizer	901	36,040	
<b>Σ Total: 25 of 3,905 entries; 3,880 more</b>	<b>7,774,188</b>	<b>339,964,792</b>	

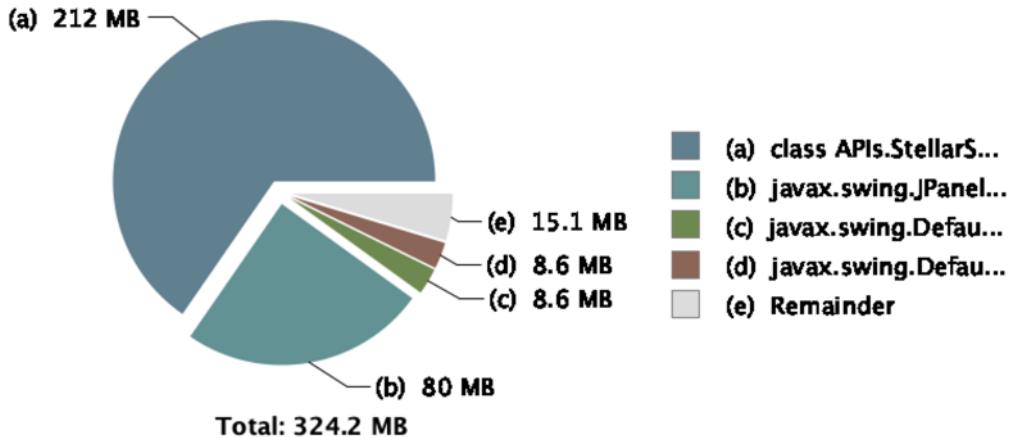
**Dominator Tree:**

Class Name	Shallow Heap <Numeric>	Retained Heap <Numeric>	Percentage <Numeric>
> <Regex>			
C class APIs.StellarSystemVisual @ 0x7804949e0	8	222,258,312	65.38%
D javax.swing.JPanel @ 0x78e938e00	344	83,848,800	24.66%
D javax.swing.DefaultComboBoxModel @ 0x78e97f808	24	8,990,824	2.64%
D javax.swing.DefaultComboBoxModel @ 0x7906188a0	24	8,990,824	2.64%
D sun.awt.image.ToolkitImage @ 0x78f331510	48	2,459,592	0.72%
D sun.awt.AppContext @ 0x78037c3b0	56	1,059,728	0.31%
C class sun.awt.SunToolkit @ 0x7801c2eb8 System Class	88	1,003,256	0.30%
C class sun.util.calendar.ZoneInfoFile @ 0x780099188 System Class	120	151,176	0.04%
C class java.io.ObjectStreamClass\$Caches @ 0x7800db370 System Class	16	65,176	0.02%
D java.util.HashSet @ 0x7800582d0	16	63,152	0.02%
D sun.font.CFontManager @ 0x7801733c0	160	57,984	0.02%
C class java.lang.invoke.MethodType @ 0x78025bd38 System Class	48	56,584	0.02%
D java.util.zip.ZipFile\$Source @ 0x780350428	64	51,160	0.02%
C class sun.util.cldr.CLDRBaseLocaleDataMetaInfo\$TZCanonicalIDMapHolder @ 0x78048:	8	51,040	0.02%
D com.sun.management.internal.DiagnosticCommandImpl @ 0x7802ceb48	40	48,744	0.01%
C class sun.java2d.loops.GraphicsPrimitiveMgr @ 0x7802032d8 System Class, Native S	24	41,720	0.01%
D sun.awt.image.ToolkitImage @ 0x7804c8db8	48	41,448	0.01%
D class jdk.internal.loader.BuiltinClassLoader @ 0x7801e97c0 System Class	16	38,128	0.01%
C class jdk.internal.math.FDBigInteger @ 0x78c518910 System Class	40	37,472	0.01%
C class sun.util.resources.Bundles @ 0x78046c7c0 System Class	24	37,312	0.01%
C class java.lang.System @ 0x780202f80 System Class, Native Stack	40	36,480	0.01%
C class java.lang.ref.Finalizer @ 0x7800c38e0 System Class	16	36,072	0.01%
D sun.security.provider.Sun @ 0x7801056f0	104	34,032	0.01%
D java.util.zip.ZipFile\$Source @ 0x780366c10	64	33,896	0.01%
C class sun.font.FontFamily @ 0x7800a6900 System Class	16	29,120	0.01%
Total: 25 of 336,914 entries; 336,889 more			

Top Consumers:

## Top Consumers

### ▼ Biggest Objects (Overview)



Leak Suspect:

### ▼ ❌ Problem Suspect 1

The class "**APIs.StellarSystemVisual**", loaded by "**jdk.internal.loader.ClassLoaders\$AppClassLoader @ 0x78021d5c0**", occupies **222,258,312 (65.38%) bytes**. The memory is accumulated in one instance of "**java.util.HashMap\$Node[]**" loaded by "**<system class loader>**".

#### Keywords

**APIs.StellarSystemVisual**  
**jdk.internal.loader.ClassLoaders\$AppClassLoader @ 0x78021d5c0**  
**java.util.HashMap\$Node[]**

[Details »](#)

### ▼ ❌ Problem Suspect 2

One instance of "**javax.swing.JPanel**" loaded by "**<system class loader>**" occupies **83,848,800 (24.66%) bytes**. The instance is referenced by **APIs.StellarSystemVisual @ 0x78e938f58**, loaded by "**jdk.internal.loader.ClassLoaders\$AppClassLoader @ 0x78021d5c0**". The memory is accumulated in one instance of "**java.lang.Object[]**" loaded by "**<system class loader>**".

#### Keywords

**java.lang.Object[]**  
**jdk.internal.loader.ClassLoaders\$AppClassLoader @ 0x78021d5c0**  
**javax.swing.JPanel**

[Details »](#)

### 3.5.3 发现热点/瓶颈并改进、改进前后的性能对比分析

### 3.5.4 在MAT内使用OQL查询内存导出

- CircularOrbit 的所有对象实例;



```
Heap Dump
OQL Console OQL Query: Run Results:
circularOrbit.StellarSystem
1 select classof(cl).name from instanceof circularOrbit.StellarSystem cl
2
```

- 大于特定长度 n 的 String 对象;

这里查询长度大于 100 的 String:

○ APIs.StellarSystemVisual (pid 95312)

Heap Dump

Name	Size	(%)
java.lang.String#1705362	29 B	(0%)
java.lang.String#1717986	29 B	(0%)
java.lang.String#3862	29 B	(0%)
java.lang.String#1717185	29 B	(0%)
java.lang.String#1717254	29 B	(0%)
java.lang.String#3781	29 B	(0%)
java.lang.String#1717306	29 B	(0%)
java.lang.String#5041	29 B	(0%)
java.lang.String#3789	29 B	(0%)
java.lang.String#1785476	29 B	(0%)
java.lang.String#1716844	29 B	(0%)
java.lang.String#1785497	29 B	(0%)
java.lang.String#4465	29 B	(0%)
java.lang.String#3242	29 B	(0%)
java.lang.String#6026	29 B	(0%)
java.lang.String#3872	29 B	(0%)
java.lang.String#3771	29 B	(0%)
java.lang.String#4956	29 B	(0%)
java.lang.String#1717443	29 B	(0%)

Results >

```

1 select s
2 from java.lang.String s
3 where s.value.length >= 100
4
5
6
7

```

- 大于特定大小的任意类型对象实例；

此处查询长度大于等于 256 的 int 数组：

Heap Dump

int[]#78	256 items
int[]#201	256 items
int[]#308	969 items
int[]#329	378 items
int[]#335	1,086 items
int[]#341	339 items
int[]#344	390 items
int[]#362	10,000 items
int[]#408	12,528 items
int[]#417	13,440 items
int[]#981	3,600 items
int[]#1193	256 items
int[]#1299	599 items
int[]#1401	18,432 items

Results >

```

1 select s
2 from int[] s
3 where s.length >= 256

```

### 3.5.5 观察jstack/jcmd导出程序运行时的调用栈

### 3.5.6 使用设计模式进行代码性能优化

使用 Flyweight 模式重新设计电子：

先新建对于电子的接口 Flyweight. java，内包含原本电子的方法，后用 ElectronFW. java 继承该接口。用 ElectronFactory. java 中的方法新建电子，并用层数作为享元 Map 的 key 值。

```
public static Flyweight getElectron(int level) {  
    ElectronFW electronFW = (ElectronFW) electronMap.get(level);  
  
    if (electronFW == null) {  
        electronFW = new ElectronFW(level);  
        electronMap.put(level, electronFW);  
    }  
    return electronFW;  
}
```

### 3.6 Git 仓库结构

## Git Log:

```
commit 7ef71494cb0963738872baafc7e0d92f0b7c13a
Author: 1178300316 <xiaoxingwen@stu.hit.edu.cn>
Date: Thu May 30 20:56:01 2019 +0800

...skipping...
commit 917e0d64eac815bac4819542fb1cd0780bafc687 (HEAD -> 35PerformanceOptimization, origin/35PerformanceOptimization, origin/32CompareIOPerformance, master, 3
`CompareIOPerformance)
Author: 1178300316 <xiaoxingwen@stu.hit.edu.cn>
Date: Sun Jun 2 21:51:01 2019 +0800

    Upgrade

commit 40593dbfc0e8c975372a13c088014d8491608449c
Author: 1178300316 <xiaoxingwen@stu.hit.edu.cn>
Date: Sat Jun 1 11:12:58 2019 +0800

    Update

commit 2630d9c80383f7a91f13782c4bb3e69c9208978f
Author: 1178300316 <xiaoxingwen@stu.hit.edu.cn>
Date: Fri May 31 12:26:55 2019 +0800

    Update

commit 2630d9c80383f7a91f13782c4bb3e69c9208978f
Author: 1178300316 <xiaoxingwen@stu.hit.edu.cn>
Date: Fri May 31 12:26:55 2019 +0800

    ...skipping...
commit 917e0d64eac815bac4819542fb1cd0780bafc687 (HEAD -> 35PerformanceOptimization, origin/35PerformanceOptimization, origin/32CompareIOPerformance, master, 3
`CompareIOPerformance)
Author: 1178300316 <xiaoxingwen@stu.hit.edu.cn>
Date: Sun Jun 2 21:51:01 2019 +0800

    Upgrade

commit 40593dbfc0e8c975372a13c088014d8491608449c
Author: 1178300316 <xiaoxingwen@stu.hit.edu.cn>
Date: Sat Jun 1 11:12:58 2019 +0800

    Update

commit 2630d9c80383f7a91f13782c4bb3e69c9208978f
Author: 1178300316 <xiaoxingwen@stu.hit.edu.cn>
Date: Fri May 31 12:26:55 2019 +0800

    Update
```

```
Lab5-1170300316 — less - git log — 158x47
commit 7ef71494cb0963738872baafca7e0d92f8b7c13a
Author: 1170300316 <xiaoxingwen@stu.hit.edu.cn>
Date: Thu May 30 20:56:01 2019 +0800

    Common Upgrade
...skipping...
commit 917ed6c6eac15bac4019542fb1cd0780ba7c687 (HEAD -> 35PerformanceOptimization, origin/35PerformanceOptimization, origin/32CompareIOPerformance, master, 3
2CompareIOPerformance)
Author: 1170300316 <xiaoxingwen@stu.hit.edu.cn>
Date: Sun Jun 2 21:51:01 2019 +0800

        Update

commit 2638d9c80383f7a91f13782c4bb3e69c9200978f
Author: 1170300316 <xiaoxingwen@stu.hit.edu.cn>
Date: Sat Jun 1 11:12:58 2019 +0800

        Update

commit 7ef71494cb0963738872baafca7e0d92f8b7c13a
Author: 1170300316 <xiaoxingwen@stu.hit.edu.cn>
Date: Thu May 30 20:56:01 2019 +0800

    Common Upgrade
commit 4264e8db571de914649acce3b5ce44f0014d8491698449c
Author: 1170300316 <xiaoxingwen@stu.hit.edu.cn>
...skipping...
commit 917ed6c6eac15bac4019542fb1cd0780ba7c687 (HEAD -> 35PerformanceOptimization, origin/35PerformanceOptimization, origin/32CompareIOPerformance, master, 3
2CompareIOPerformance)
Author: 1170300316 <xiaoxingwen@stu.hit.edu.cn>
Date: Sun Jun 2 21:51:01 2019 +0800

        Update

commit 40593dbfc0e8c975372a13c00014d8491698449c
Author: 1170300316 <xiaoxingwen@stu.hit.edu.cn>
Date: Sat Jun 1 11:12:58 2019 +0800

        Update
```

```
Lab5-1170300316 — less - git log — 158x47
commit a0ec28b11127474a4079bf701cf0iae9e9314bb9
Author: 1170300316 <xiaoxingwen@stu.hit.edu.cn>
Date: Sun May 26 11:34:10 2019 +0800

...skipping...
commit 917ed6c6eac15bac4019542fb1cd0780ba7c687 (HEAD -> 35PerformanceOptimization, origin/35PerformanceOptimization, origin/32CompareIOPerformance, master, 3
2CompareIOPerformance)
Author: 1170300316 <xiaoxingwen@stu.hit.edu.cn>
Date: Sun Jun 2 21:51:01 2019 +0800

        Update

commit 40593dbfc0e8c975372a13c00014d8491698449c
Author: 1170300316 <xiaoxingwen@stu.hit.edu.cn>
Date: Sat Jun 1 11:12:58 2019 +0800

        Update

commit 2638d9c80383f7a91f13782c4bb3e69c9200978f
Author: 1170300316 <xiaoxingwen@stu.hit.edu.cn>
Date: Fri May 31 12:26:55 2019 +0800

        Update

commit 7ef71494cb0963738872baafca7e0d92f8b7c13a
Author: 1170300316 <xiaoxingwen@stu.hit.edu.cn>
Date: Thu May 30 20:56:01 2019 +0800

    Common Upgrade
commit 4254e8db571de914649acce3b5ce44f0014d8491698449c
Author: 1170300316 <xiaoxingwen@stu.hit.edu.cn>
Date: Tue May 28 21:17:34 2019 +0800

    32CompareIOPerformance

commit a0ec28b11127474a4079bf701cf0iae9e9314bb9
Author: 1170300316 <xiaoxingwen@stu.hit.edu.cn>
Date: Sun May 26 11:34:10 2019 +0800

    BufferedReader

commit 7b6bbbc02afe696d673141e5310fbbcf5f02ffa (origin/32AddOutput, 32AddOutput)
Author: 1170300316 <xiaoxingwen@stu.hit.edu.cn>
...skipping...
commit 917ed6c6eac15bac4019542fb1cd0780ba7c687 (HEAD -> 35PerformanceOptimization, origin/35PerformanceOptimization, origin/32CompareIOPerformance, master, 3
2CompareIOPerformance)
```

```

Date: Fri May 31 12:26:55 2019 +0800
Update
commit 7ef71494cb0963738872baafca7e0d92f8b7c13a
Author: 1179300316 <xiaoxingwen@stu.hit.edu.cn>
Date: Thu May 30 26:56:01 2019 +0800
Common Upgrade
commit 4254e8db571de914649cce3b5ce44f001489922
Author: 1179300316 <xiaoxingwen@stu.hit.edu.cn>
Date: Tue May 28 21:17:34 2019 +0800
32CompareIOPerformance
commit a0ec28b11127474a4079bf701cf01ae9e9314bb9
Author: 1179300316 <xiaoxingwen@stu.hit.edu.cn>
Date: Sun May 26 11:34:10 2019 +0800
BufferedReader
commit 7b4bdc0c72af5e69dd73141e5310bbcfc5f92ff9 (origin/32AddOutput, 32AddOutput)
Author: 1179300316 <xiaoxingwen@stu.hit.edu.cn>
Date: Sun May 26 11:09:00 2019 +0800
32AddOutput
...skipping...
commit 917e6c60eac815bac4819542fb1cd0780bafc687 (HEAD -> 35PerformanceOptimization, origin/35PerformanceOptimization, origin/32CompareIOPerformance, master, 32CompareIOPerformance)
Author: 1179300316 <xiaoxingwen@stu.hit.edu.cn>
Date: Sun Jun 2 21:51:01 2019 +0800
Upgrade
commit 40593dbfc0e8c975372a13c08014d0491698449c
Author: 1179300316 <xiaoxingwen@stu.hit.edu.cn>
Date: Sat Jun 1 11:12:58 2019 +0800
Update
commit 2638d9c00383f7a91f13782cabb3e69c9200978f
Author: 1179300316 <xiaoxingwen@stu.hit.edu.cn>
Date: Fri May 31 12:26:55 2019 +0800
Update

```

## 4 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
5.26	19:00-23:30	完成 3.1 的人工检查	完成
5.27	18:00-22:00	完成 3.1	完成
5.28	19:00-24:00	完成 3.2 的两种读入方式	完成
5.29	20:00-22:00	完成 3.2	完成
5.30	19:00-23:00	完成 3.3	完成
5.31	21:00-23:00	完成 3.4	完成
6.1	21:00-23:30	完成 3.5	完成

## 5 实验过程中遇到的困难与解决途径

遇到的难点	解决途径
代码格式化工作冗余，很多重复性操作。	使用 Google 官方的 Java Google Formatter，完成大量的重复性工作。
读入大量文件后，提示 heap 溢出。	思路主要有两种，其一是提前开辟更大的堆。但发现是 GUI 过多的图片（因为每个 planet 都有个 planet 图片代表）引起的后，将 GUI 适度简化。

## 6 实验过程中收获的经验、教训、感想

### 6.1 实验过程中收获的经验和教训

1. 写一个程序和项目应该从最开始就保证程序的规范，不应该到最后进行修改。这样会显得十分混乱。
2. 用工具对程序性能进行检查是必须的，这样可以节省大量的内存和 CPU 资源。在本实验中的直观体现是 IO 过程大大提速。

### 6.2 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？

看起来很美的代码不一定运行起来美，运行起来美的代码不一定看起来美。看来的美的代码方便开发者，运行起来美的代码方便使用者，我认为运行起来美更重要，在有限的时间，我也希望能最大程度提升运行起来的美感。

(2) 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？

可以。SpotBugs 可以找出 bug，也能保障以后再添加功能的时候程序的安全性。CheckStyle 统一了程序的风格，使得阅读和开发效率提高。

(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？

一开始计算机 IO 数据量小，故用字节流类 IO，如 FileInputStream，后来字节流类操作太底层，就增加了字符流类，如 FileReader，BufferedReader。为了提供快速异步 IO，后来添加了 NIO 的 Buffer 类和 Channel 类。

(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？

计算机系统中的内存管理机制将内存分为数据区、字段区、堆、共享库和栈，JVM 将内存分为方法区、堆区、本地方法栈、虚拟机栈、程序计数器，可以看出 JVM 的内存管理机制是基于普通的内存管理机制，但更加细致。

差异在于 JVM 中

(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还有改进的空间吗？

Java 开发者不需要专门编写内存回收和垃圾清理代码，内存泄露和溢出发生的可能性较低，简化了程序员的工作，在面向工程编写、面向对象程序编写过程中是需要自动 GC 的。

课程介绍的 JVM 的 GC 机制 CMS，面对超大堆时会力不从心，一次 full GC 等待时间过长，这不利于编写有大量图形程序或者处理其它有大量数据的程序，这是有很大提升空间的。

(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？

有。根据 VisualVM 的分析结果，GC 待提升之处很多，这次实验中能降低 30% 的内存占用。

(7) 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？

是。比如通过 MAT 的 Memory Leak 功能，立刻能找到消耗资源的类和对象，能对提升程序性能对症下药。当然这是建立在程序模块化的基础上的，如果程序编写功能过于集中，优化还是很困难。

(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？

有。整体代码质量的提升应该是从一个程序、项目一开始做起的，不应该到最后对代码进行修补。

(9) 关于本实验的工作量、难度、deadline。

工作量、deadline 都合适，但是难度大，涉及很多工具的使用。

(10) 到目前为止，你对《软件构造》课程的意见与建议。

没有更多建议了。