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	强文杰
学号	1171000410
班号	1703005
电子邮件	672334335@qq.com
手机号码	18800421389

目录

1 实验目标概述 1
2 实验环境配置 1
3 实验过程 3
3.1 Static Program Analysis 4
3.1.1 人工代码走查（walkthrough） 4
3.1.2 使用 CheckStyle 和 SpotBugs 进行静态代码分析 6
3.2 Java I/O Optimization 11
3.2.1 多种 I/O 实现方式 11
3.2.2 多种 I/O 实现方式的效率对比分析 16
3.3 Java Memory Management and Garbage Collection (GC) 19
3.3.1 使用-verbose:gc 参数 19
3.3.2 用 jstat 命令行工具的-gc 和-gcutil 参数 21
3.3.3 使用 jmap -heap 命令行工具 23
3.3.4 使用 jmap -clstats 命令行工具 25
3.3.5 使用 jmap -permstat 命令行工具 26
3.3.6 使用 JMC/JFR、jconsole 或 VisualVM 工具 26
3.3.7 分析垃圾回收过程 27
3.3.8 配置 JVM 参数并发现优化的参数配置 28
3.4 Dynamic Program Profiling 29
3.4.1 使用 JMC 或 VisualVM 进行 CPU Profiling 31
3.4.2 使用 VisualVM 进行 Memory profiling 32
3.5 Memory Dump Analysis and Performance Optimization 34
3.5.1 内存导出 34
3.5.2 使用 MAT 分析内存导出文件 34
3.5.3 发现热点/瓶颈并改进、改进前后的性能对比分析 39
3.5.4 在 MAT 内使用 OQL 查询内存导出 40
3.5.5 观察 jstack/jcmd 导出程序运行时的调用栈 42
3.5.6 使用设计模式进行代码性能优化 44
4 实验进度记录 46

5 实验过程中遇到的困难与解决途径 47
6 实验过程中收获的经验、教训、感想 48
6.1 实验过程中收获的经验和教训 48
6.2 针对以下方面的感受 48

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），对其进行持续的改进和优化。

需要使用的工具和方法主要有以下几种：

1. 静态代码分析（CheckStyle 和 SpotBugs）
2. 动态代码分析（Java 命令行工具 jstat、jmap、jcmd、VisualVM、JMC、JConsole 等）
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

简要陈述你配置本次实验所需环境的过程，必要时可以给出屏幕截图。

特别是要记录配置过程中遇到的问题和困难，以及如何解决的。

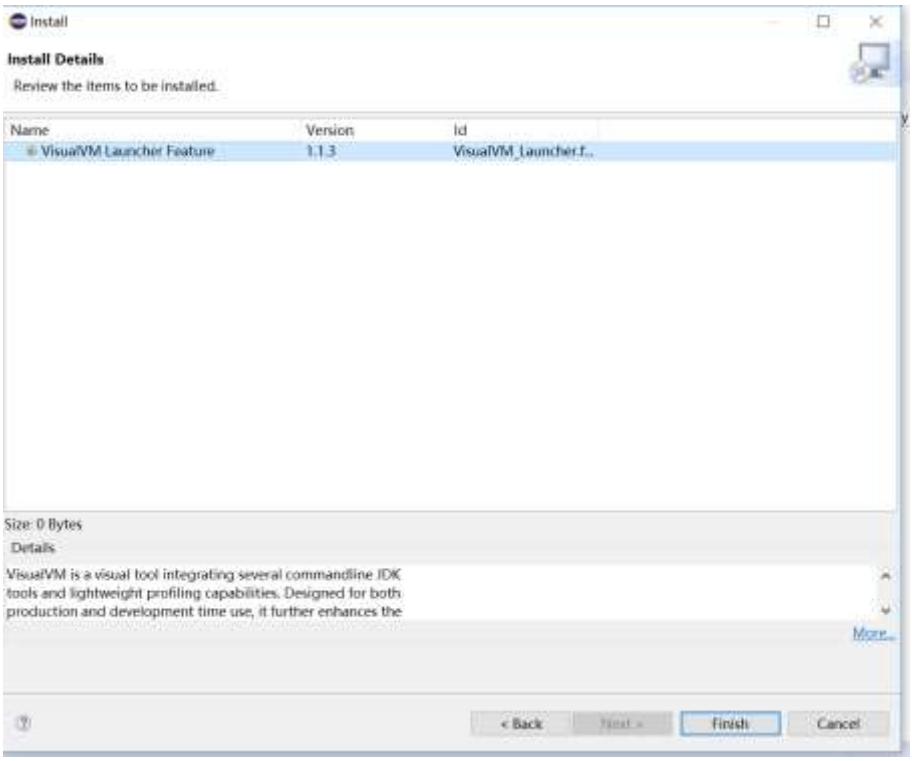
一. 配置 Visual VM

首先在 https://visualvm.github.io/download.html 下载 VisualVM 启动器和在 https://visualvm.github.io/idesupport.html 下载 visualVM 到 Eclipse 的插件。

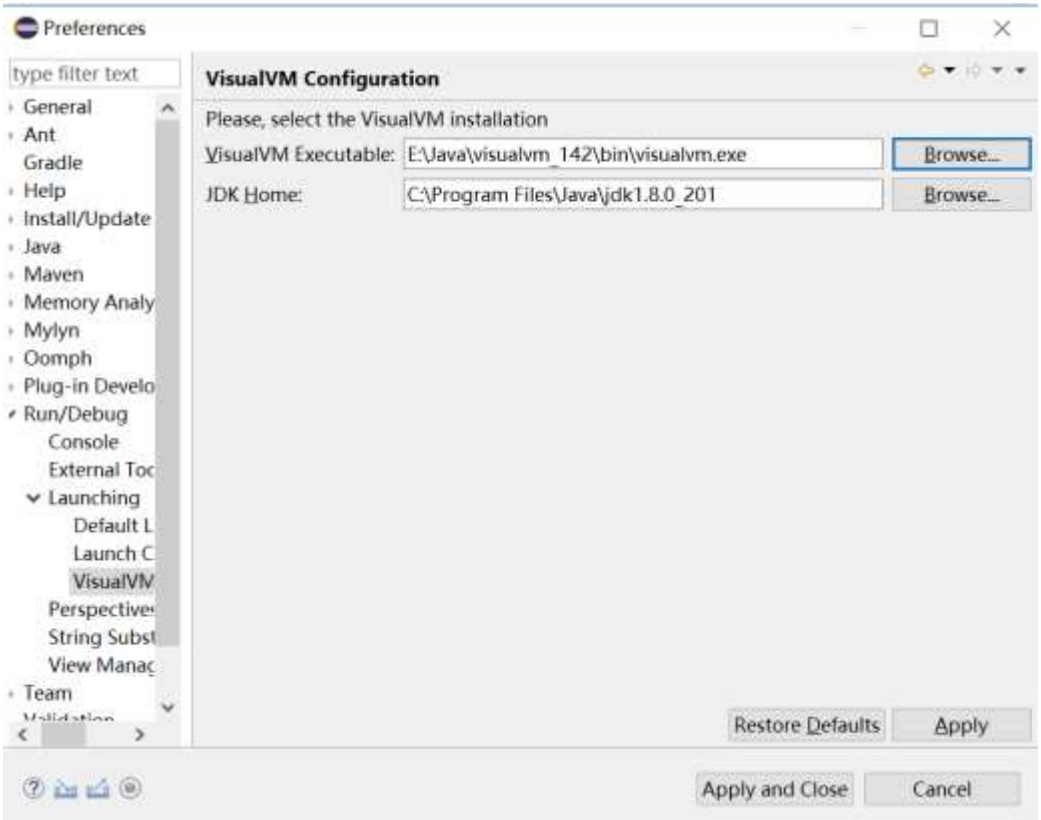
下载完成后将 visualvm_141.zip 解压到当前目录下。

将 visualvm_launcher_u2_eclipse.zip 解压到当前目录下。

在 Eclipse 中选择 helper 下的 install new software，在目录选择解压后的 visualvm_launcher_u2_eclipse，然后进行安装。

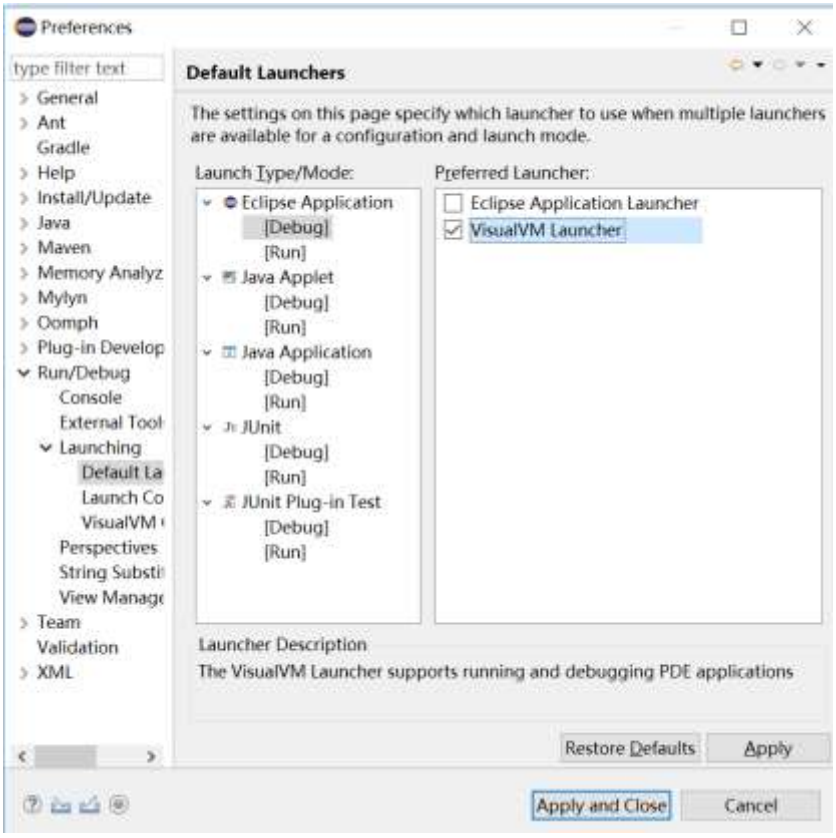


在 Eclipse 的 Window 下的 Preferences 中按如图配置 VisualVM 的安装目录以及 jdk 的目录。



配置完成 visualVM 之后，再进行启动注册的配置，在 Default Launchers 中

选择启动器。



二. 配置 MAT

在 Eclipse 中选择 helper 下的 install new software，在网址中输入 http://download.eclipse.org/mat/1.8.1/update-site/ 进行安装。

在这里给出你的 GitHub Lab5 仓库的 URL 地址（Lab5-学号）。

https://github.com/ComputerScienceHIT/Lab5-1171000410.git

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）

列出你所发现的问题和所做的修改。每种类型的问题只需列出一个示例即可。

以下选定 Google 编程规范。

#源文件基础

没有需要修改的地方

#源文件结构

1.import 语句：删去了 import 语句中的通配符

2.对类的成员顺序重新排序，修改了 ConcreteCircularOrbit 中一些混乱的成员顺序。

#格式

1.使用大括号(即使是可选的)：将很多 if 后面省去的大括号重新添加上。

2.对于非空块和块状结构，大括号遵循 Kernighan 和 Ritchie 风格。重新调整了大括号的使用情况。

3.块缩进：2 个空格。把许多块缩进使用的 tab 符改成了两个空格。

4.列限制：80 或 100

对于 AtomStructure 等继承和实现关系比较复杂的类，单行字符超出限制，换行处理。

5. 把方法体中的组合声明修改，比如 int a, b;

6. 修改了 C 风格的数组声明

在 read File 方法中，定义了字符串数组，中括号是类型的一部分：String[]

args，而非 String args[]

7.在 MyOrbitScences 类中，存在大量的 switch 语句，缩进与其它块状结构一致，switch 块中的内容缩进为 2 个空格。

8.把类中原来自动换行时缩进的 tab 符改为至少+4 个空格

9.对于空白。在函数体内，语句的逻辑分组间使用了空行。

#命名约定

1.根据对所有标识符都通用的规则，把方法中 friend_set 这种类型的命名修改成 friendSet

2.包名全部小写，连续的单词只是简单地连接起来，不使用下划线。
把 APIS 包 rename 为 apis，把 centralObject 包 rename 为 centralobject

3. 类名以 UpperCamelCase 风格编写；方法名都以 lowerCamelCase 风格编写。这类型的问题没有发现。

4.对方法中出现的常量啊 a，修改其命名为：全部字母大写，用下划线分隔单词。

5. 参数名
把单个字符命名的参数名全部修改为一个单词。

#编程实践

1.把 MyException 类、MyFormatter 和 ConcreteCircularOrbit 类中的合法重写全部加上了@Override

2.对捕获的异常全部做出相应

#Javadoc

1.添加<p>。添加的情况：空行(即，只包含最左侧星号的行)会出现在段落之间和 Javadoc 标记(@XXX)之前(如果有的话)。 除了第一个段落，每个段落第一

个单词前都有标签<p>，并且它和第一个单词间没有空格。

2. 标准的 Javadoc 标记按以下顺序出现：@param, @return, @throws, @deprecated, 前面这 4 种标记如果出现，描述都不能为空。

如图所示：

```
/**
 * A track system, the objects on the track are moving
 *
 * <p>PS2 instructions: this is a required ADT interface.
 *
 * @param <L> type of central object labels in this graph, must be immutable
 * @param <E> type of orbital object labels in this graph, must be immutable
 */
public interface CircularOrbit <L,E>{
```

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

列出你所发现的问题和所做的修改。每种类型的问题只需列出一个示例即可。在修改之前很恐怖：

Checkstyle violations

Overview of Checkstyle violations - 6240 markers in 29 categories (Filter matched 6240 of 624

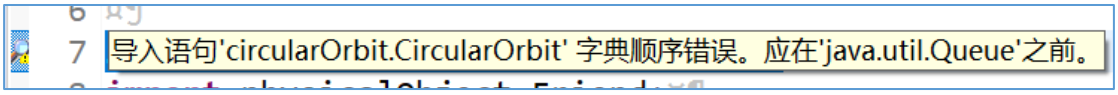
Checkstyle violation type	Marker count
第 X 个字符 'X'应该与当前多代码块的下一部分...	5
'X' 缩进了X个。	990
Javadoc的第一句缺少一个结束时期。	112
'X' 子元素缩进了X个。	1377
每一个变量的定义必须在它的声明处，且在同...	13
'X' 缩进了X。	6
'X' 子元素缩进了X。	8
变量'X 行）。若需要存储该变量的值，请将其...	9
数组大括号位置错误。	1
Javadoc 缩进级别错误，应为 X 个缩进符。	1
导入语句'X' 字典顺序错误。应在'X'之前。	77
'X' 前应有空行。	6
'X' 的无用标签 X。	2
缺少 Javadoc。	7
'X' 前应有空格。	65
空行后应有 <p> 标签。	1
注释应与第X个。	26
'X' 前不应有空格。	59
不应使用 '.*' 形式的导入 - X。	18
顶级类 X 应位于它自己的源文件中。	3
本行字符数 X个。	142
'X' 应另起一行。	3
'X' 后不应有空格。	1
行内含有制表符 tab。	3154
名称 'X' 中不能出现超过 'X' 个连续大写字母。	1
'X' 后字符不合法。	3
'X' 结构必须使用大括号 '{}'。	8
名称 'X' 必须匹配表达式：'X'。	59

#源文件基础

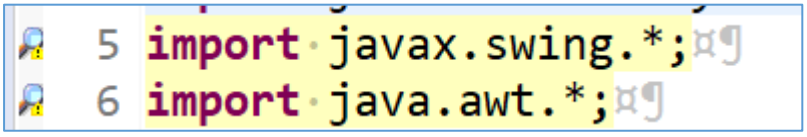
没有需要修改的地方

#源文件结构

1.修改 import 的顺序



2.import 中避免使用通配符



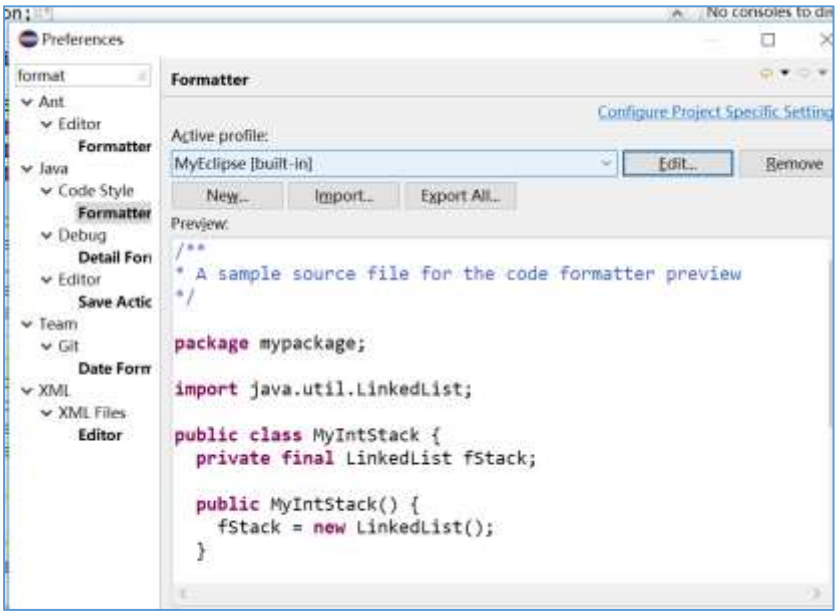
#格式

1.每一个变量的定义必须在它的声明处，且在同一行

要求同一行只声明一个变量。

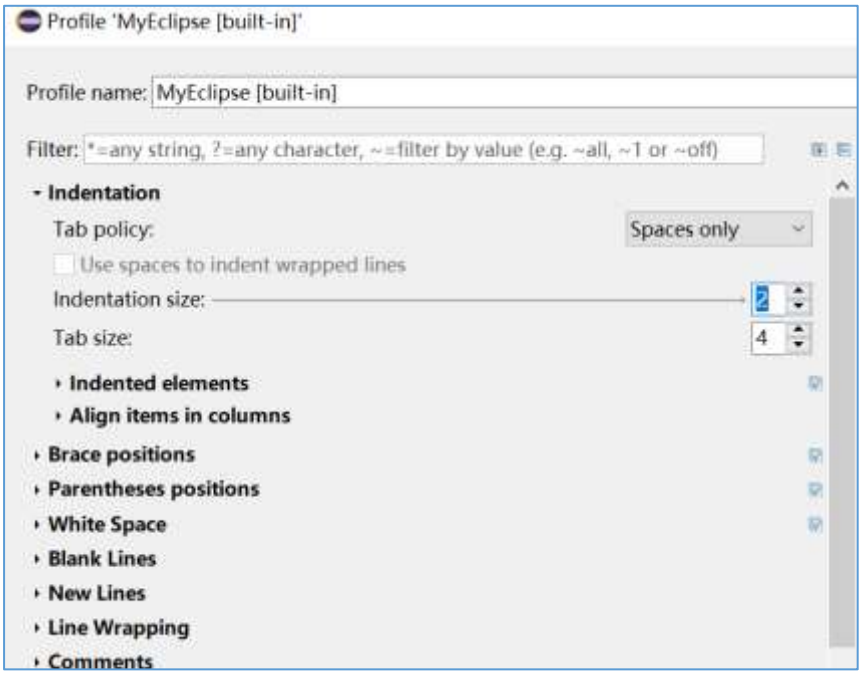
2.行内含有制表符 tab，缩进符数量不正确。

这个问题在所有的代码行中都出现了，当然对于这么大量的代码，我们不可能一行行修改，修改方式如下：

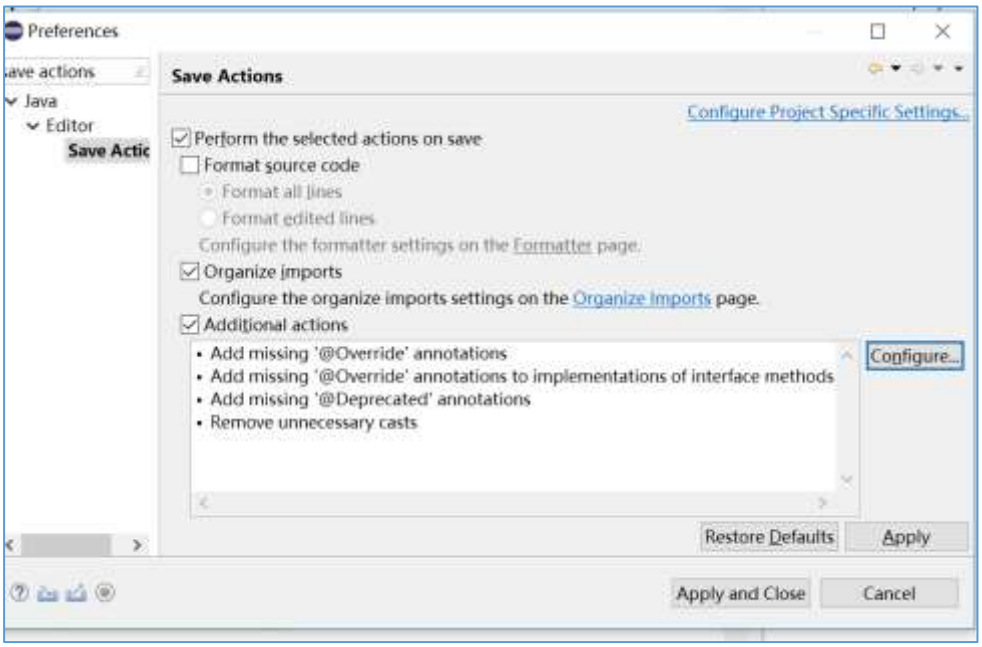


采用的方法为：点击“Java”->“Code Style”-> “Formatter”菜单，在右侧选中模板后点击“Edit”按钮

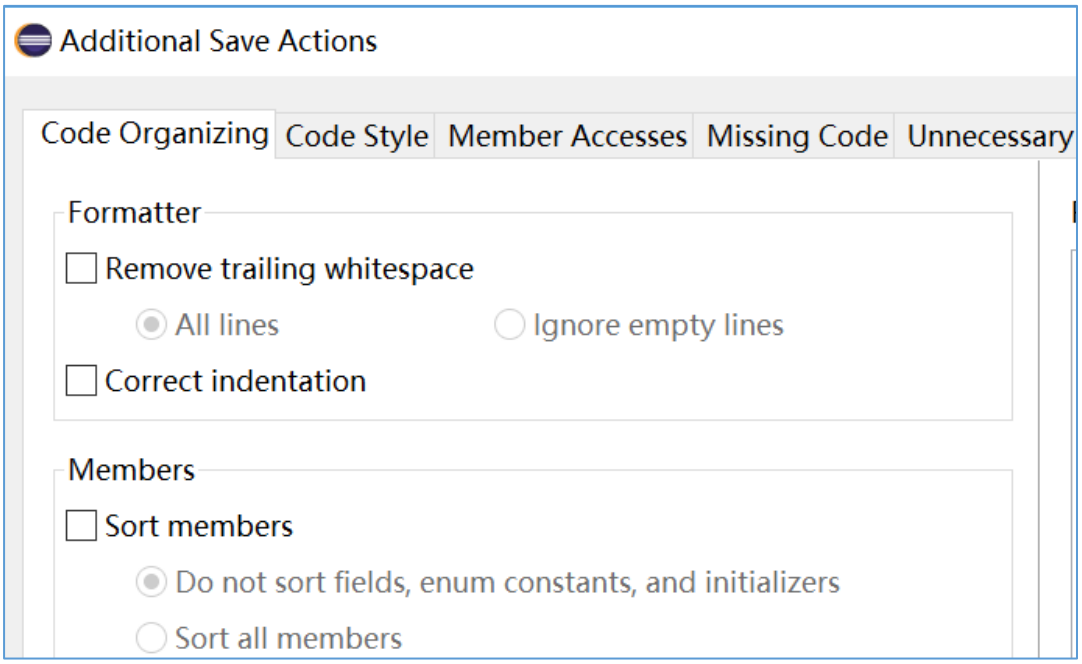
在 Tab policy 处选择 Space only ，并且 Indentation Size 设置为 2，这是为了使 4 个缩进符变为 2 个缩进符。



最后勾选上”Perform the selected actions on save”,”Additional actions”,并点击”Configure”进行配置.



使 Correcct indentation 为未勾选的状态即可。



最后“CTRL＋SHIFT＋F”快捷键，此问题解决。（怎么感觉在写教程 2333）

3. 变量'y'声明及第一次使用距离 5 行（最多：3 行）。若需要存储该变量的值，请将其声明为 final 的（方法调用前声明以避免副作用影响原值）。

解决方法：变量声明前增加 final

4.缩进符数目不正确

有些无法批量修改的只能一点点改了。

5. WhitespaceAround: '+' is not preceded with whitespace.

“+”左右要留空白

#命名约定

1. Local variable name 'E2E' must match pattern '^[a-z]([a-z0-9][a-zA-Z0-9]*)?$'.

把原本使用的 T2E 这种局部变量名称修改为 t2e

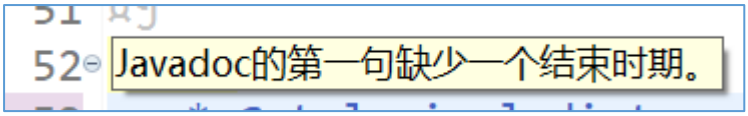
#编程实践

1.把 MyException 类、MyFormatter 和 ConcreteCircularOrbit 类中的合法重写全部加上了@Override

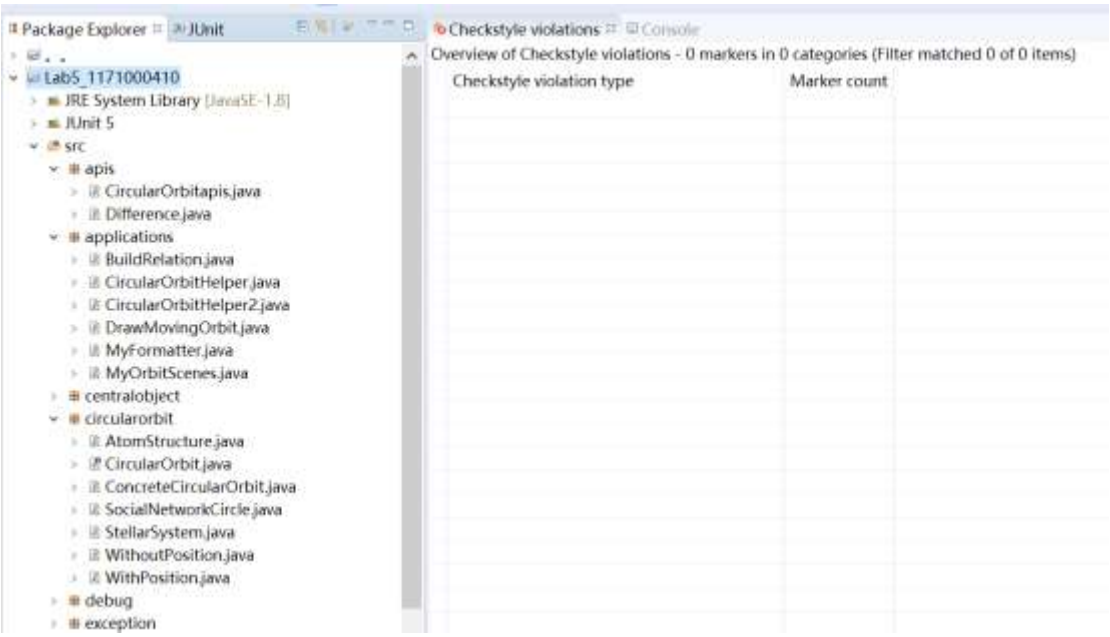
2.对捕获的异常全部做出相应

#Javadoc

1.给 Javadoc 的语句添加结束时期



结果：手动改完所有的问题后：（警告终于全部消除了）



对比分析两种工具发现问题的能力和发现问题的类型上有何差异。

发现问题的能力：两种工具发现问题的能力来说，Checkstyle 运行起来更快，spotbugs 运行稍慢一些，二者都能很细致的发现问题并且提示。个人认为 spotbugs 发现的问题对程序的影响更大，Checkstyle 发现的主要是程序员写代码格式等不规范问题，其作用相对小一些。

发现问题的类型：Checkstyle 发现的大多数是 java 代码中的格式错误，它能够自动化代码规范检查过程；而 spotbugs 发现的大多数是代码中可能存在的安全问题，例如对 null 的防范，或是考虑代码的正确性。

3.2 Java I/O Optimization

3.2.1 多种 I/O 实现方式

实现了哪些 I/O 方式来读写文件，具体如何实现的。

Input：

首先为了 strategy 设计模式的方便实现和对 input 时间的准确收集，我选择了一次性读完整个文件，然后将文件存在一个 string 中（list 中同理），然后通过分割字符串实现类似按行读文件的操作。

其中将文件存到字符串的操作如下：

```
StringBuffer buffer = new StringBuffer();
buffer.append(s.trim()+"\n");
String fileStr = buffer.toString();
```

1.BufferedReader

字符串缓冲读取类。java 的 IO 包使用包装器模式设计的，也就是说用 FileReader 包装了 File，又用 BufferedReader 包装了 FileReader，才可以用 BufferedReader。

```
BufferedReader bf = new BufferedReader(new FileReader(file));
String s = null;
while ((s = bf.readLine()) != null) { // 使用readLine方法，一次读一行
    buffer.append(s.trim()+"\n");
}
bf.close(); //结束文件读取
```

2.InputStream

字节输入流。

具体步骤如下：

使用 File 类找到一个文件；通过子类实例化父类对象；进行读操作，所有的内容都读到字符数组之中；读取内容；关闭输出流。

```
            in = new FileInputStream(file);
            fileContent = new byte[in.available()];

            in.read(fileContent);
            in.close();
```

3.Scanner

Scanner 经常被我们用于读取控制台的输入，但是我们只需要将他构造函数中的 System.in 替换成一个 File 文件，他就可以变成一个读取文件的 Scanner 了，Scanner 的按行读取方法是 nextLine()我们在使用它的时候可以在前面判断它时候 hasNextLine()。

```
    in = new Scanner(file);
    while(in.hasNextLine()) {
        a.append(in.nextLine()+"\n");
    }
```

Output:

1.BufferedWriter

首先创建读取字符数据流对象关联所要复制的文件，创建缓冲区对象关联流对象，最后从缓冲区中将字符创建并写入到要目的文件中。

简单说 BufferedWriter 为我们提供了写字符串的方法，直接 write()里面放个字符串就可以了。

最后需要 writer.flush();
writer.close();

2.OutputStream

创建字节输出流对象了做了几件事情：

调用系统功能去创建文件；创建 outputStream 对象；把 outputStream 对象指向这个文件

3.FileChannel

开启 FileChannel：无法直接打开 FileChannel 通过 OutputStream 获取 FileChannel。

使用 FileChannel.write()方法向 FileChannel 写数据，该方法的参数是一个

Buffer。按字节写入。用完 FileChannel 后需将其关闭。

以下是 AtomStructureFileChannel 示例：

```
// 创建缓冲区
ByteBuffer buf = ByteBuffer.allocate(1024);

Nucleus n = (Nucleus) c.getCentralObject();
int trackNum = c.getTracks().size();
String a = "ElementName ::= " + n.getName() + "\n";
String b = "NumberOfTracks ::= " + trackNum + "\n";
String d = "NumberOfElectron ::= ";

// 将数据装入缓冲区
buf.put(a.getBytes());
buf.put(b.getBytes());
buf.put(d.getBytes());

ArrayList<ArrayList<Electron>> t2e = c.getT2E();
for (int i = 0; i < trackNum; i++) {
    a = (i + 1) + "/" + t2e.get(i).size() + ";";
    buf.put(a.getBytes());
}

// 反转缓冲区(limit设置为position,position设置为0,mark设置为-1)
buf.flip();
// 将缓冲区中的数据写入到通道
fc.write(buf);
// 写完将缓冲区还原(position设置为0,limit设置为capacity,mark设置为-1)
buf.clear();
```

另外，在写实验中提供的大文件时，经常会出现缓冲区溢出的情况，因此我们需要对缓冲区剩余容量进行估计。写文件之前我给设置 ByteBuffer 缓冲区大小为 0x300000（也就是 3M），然后在后面向文件中写的时候。增加如下判断：

```
if(buf.remaining() < 1024) {
        buf.flip();
        fc.write(buf);
        buf.clear();
    }
```

如何用 strategy 设计模式实现在多种 I/O 策略之间的切换。

因为我程序中 Input 和 Output 策略是分开使用的，故这部分也区分 Input 和 Output 来说明。

Input:

因为对于输入，我的每个策略都返回文件的字符串即可，因此可以借此来提

高程序的可复用性，具体的策略只需要 3 个类，分别代表了三种读文件的方式。

抽象策略对象(Strategy)：它可由接口或抽象类来实现。

```
public interface ReadStrategy {

    public String readFile(File file);

}
```

具体策略对象(ConcreteStrategy)：它封装了读文件的不同实现策略，所有策略都需要 implements ReadStrategy 接口，每一个都根据自己这个方法的特点对方法进行了重写，其中具体的读文件方式已说明。

环境对象(Context)：分别在三个应用场景中实现了对抽象策略中定义的接口的引用。我们在使用它的时候，直接声明一个应用场景类，通过用户的不同输入来给 Context 传入不同的读策略即可，这样我们就能够只使用一个变量而获得多种行为模式了。

以下为三种读取策略的切换场景。

```
        printReadMenu();
        String read = reader.readLine();
        File file = new File(fileName);

        System.out.println("AtomicStructure读取文件时间：");
        switch (read) {
        case "1":
          ReadStrategy r1 = new AllBufferedReader();
          ast.readFile(r1.readFile(file));
          break;

        case "2":
          ReadStrategy r2 = new AllInputStream();
          ast.readFile(r2.readFile(file));
          break;

        case "3":
          ReadStrategy r3 = new AllScanner();
          ast.readFile(r3.readFile(file));
          break;

        default:
          System.out.println("输入选项错误，请重新输入！");
          break;
        }
```

Output：

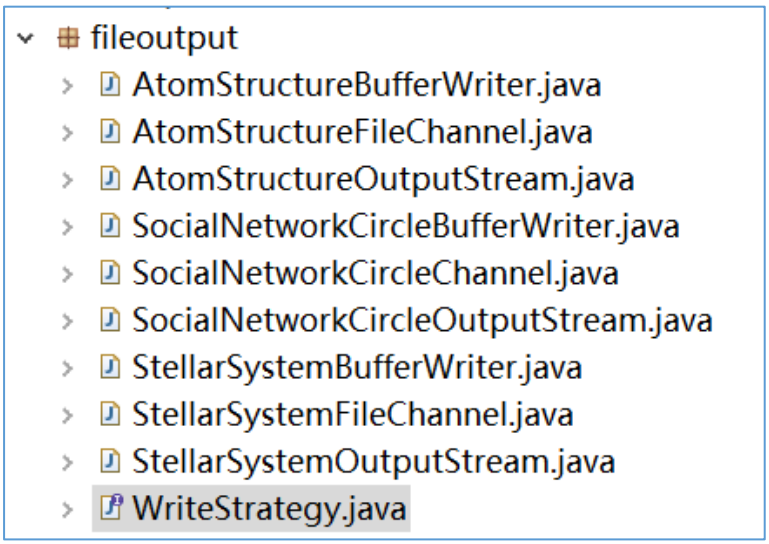
因为三个不同应用场景要实现三种不同的写文件的方式，并且此时没有很好提高复用性的方法，因此具体策略我用 9 个类来实现。

抽象策略对象(Strategy)：它可由接口或抽象类来实现。

```
public interface WriteStrategy {

    public void writeFile(String fileName,CircularOrbit c);

}
```

具体策略对象(ConcreteStrategy)：它封装了写文件的不同实现策略，所有策略都需要 implements WriteStrategy 接口，每一个都根据自己这个方法的特点对方法进行了重写，其中具体的读文件方式已说明。

写文件 package 的结构如下：



环境对象(Context)：分别在三个应用场景中实现了对抽象策略中定义的接口的引用。同理，我们在使用它的时候，直接声明一个应用场景类，通过用户的不同输入来给 Context 传入不同的写策略即可，这样我们就能够只使用一个变量而获得多种行为模式了。

方法：public void write(WriteStrategy writeStrategy)

```
switch (line) {
case "1":
  WriteStrategy w = new AtomStructureBufferWriter();
  w.writeFile("write_txt/AtomStructureWrite.txt", ast);
  ast.write(w);
  System.out.println("成功写入文件. AtomStructureBufferWriter时间为: ");
  break;

case "2":
  WriteStrategy w2 = new AtomStructureOutputStream();
  w2.writeFile("write_txt/AtomStructureWrite.txt", ast);
  ast.write(w2);
  System.out.println("成功写入文件. AtomStructureOutputStream时间为: ");

  break;

case "3":
  WriteStrategy w3 = new AtomStructureFileChannel();
  w3.writeFile("write_txt/AtomStructureWrite.txt", ast);
  ast.write(w3);
  System.out.println("成功写入文件. AtomStructureFileChannel时间为: ");

  break;
```

3.2.2 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. 收集程序 I/O 语法文件的时间。

因为 Input 阶段就采用了一次性将文件读完的策略，因此 Input 和 Output 收集时间的方式类似，在 Input 或 Output 开始阶段获取时间，在结束阶段再次获取时间，然后将两次时间相减即可。

具体操作：

long startTime = System.*currentTimeMillis*();

long finishTime = System.*currentTimeMillis*();

long totalTime = finishTime - startTime;

System.***out***.println(totalTime + "ms");

2.程序优化：

因为每次写入文件之前都需要在已经建立好轨道的基础上，因此即使我们一次性将文件读完，如果没有很好的建立轨道的策略，在读入大文件并进行 I/O 对比时，效率也是相当低的。

首先对 lab4 原本的设置进行修改，因为大文件中是存在非法情况的，因此我们不能像 lab4 中读到异常情况就重新选择文件，处理策略是读到异常情况，跳过该行的读取，继续读下一行。

因为程序在一定时间内难以读出 SocialNetworkCircle.txt 文件并建立轨道，因此主要的优化在 SocialNetworkCircle。

问题一：因为我们在读到每个 Friend 时，需要把建立的对象存储起来，存储方式是 Hashset，然而 hashSet 存在一个致命的缺陷，当数据量较小时还没有问题，可当数据量过大时 hashSet 的存储会出现问题。hashSet 集合存在大小，即存储的数据量上限，如果最初没有设置上限大小的话，默认的大小自然不可能有多大。当存储的数据量超出上限时，hashset 会自动倍数扩大，这时集合 set1 就被更大的 set2 替代，而原本 set1 的数据自然不会直接出现在 set2 中，于是在扩大时 set1 会把存储的所有数据取出放入内存中，再重新放入 set2 中。这一部分不仅耗费了时间同时占用内存，降低了代码性能。

优化一：根据 Friend 对象最大的数目，初始设置了 hashSet 的大小是 110000，这样向集合中添加的时候，不必一直 transfer 数据了。

问题二：即使我们有了存储 Friend 的 hashSet，在读 SocialTie 取 Friend 对象的时候，需要遍历一个很大的集合，根据姓名来取对象，效率依然很低，第一次尝试的时候构建轨道花了 2 个小时。

优化二：既然遍历集合效率很低，我们为什么不采用 hashMap 取数据的方式呢，时间复杂度只有 O(1)。因此我们重写 Friend 中 hashCode 方法，

```
public int hashCode() {
  int h = super.getName().hashCode();
  return h;
```

每次在 hashMap 中寻找对象的时候，只需要根据 name 的 hashCode 值来搜索，相当于查找一个数组中指定位置的元素，时间复杂度只有 O(1)

```
  return friendMap.get(name.hashCode());
```

如此优化后，程序的运行效率提高了很多，方便下面的统计时间了。

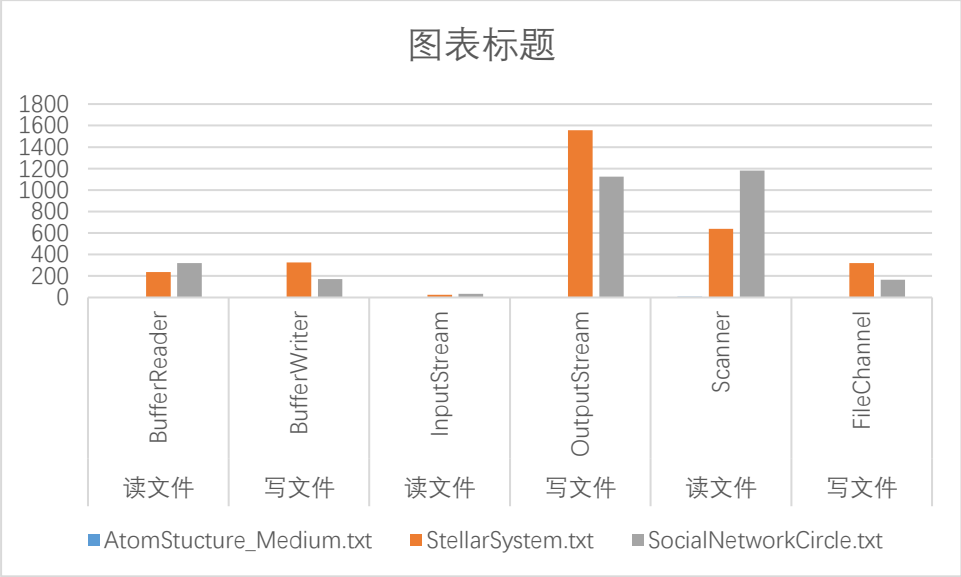
2.表格方式对比不同 I/O 的性能。

由于对读入文件中的异常情况进行了剔除，因此写入的文件并没有读入的文件大。

		AtomStucture_Medium.txt	StellarSystem.txt	SocialNetworkCircle.txt
读文件	BufferReader	0ms	238ms	320ms
写文件	BufferWriter	3ms	327ms	170ms
读文件	InputStream	0ms	24ms	35ms
写文件	OutputStream	2ms	1556ms	1123ms
读文件	Scanner	7ms	639ms	1180ms
写文件	FileChannel	2ms	319ms	165ms

图形对比不同 I/O 的性能。

由于 AtomicStructure_Medium.txt 文件较小，因此读此文件的时间较少，在图像上基本反映不出。

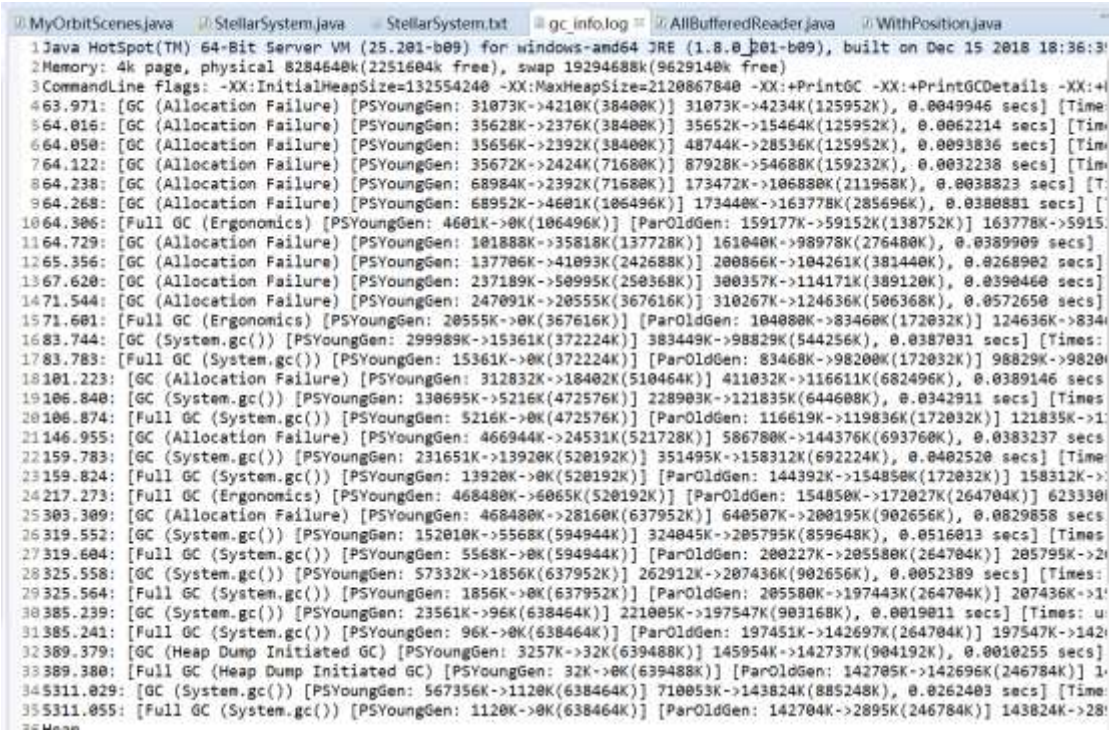


3.3 Java Memory Management and Garbage Collection (GC)

3.3.1 使用-verbose:gc 参数

配置参数：

日志文件中的部分内容如下：



```
MyOrbitScenes.java  StellarSystem.java  StellarSystem.txt  gc_info.log  AllBufferedReader.java  WithPosition.java
1 Java HotSpot(TM) 64-Bit Server VM (25.201-b09) for windows-amd64 JRE (1.8.0_201-b09), built on Dec 15 2018 18:36:3
2 Memory: 4k page, physical 8284640k(2251604k free), swap 19294688k(9629140k free)
3 CommandLine flags: -XX:InitialHeapSize=132554240 -XX:MaxHeapSize=2120867840 -XX:+PrintGC -XX:+PrintGCDetails -XX:+
4 63.971: [GC (Allocation Failure) [PSYoungGen: 31073K->4210K(38400K)] 31073K->4234K(125952K), 0.0049946 secs] [Time
5 64.016: [GC (Allocation Failure) [PSYoungGen: 35628K->2376K(38400K)] 35652K->15464K(125952K), 0.0062214 secs] [Tim
6 64.050: [GC (Allocation Failure) [PSYoungGen: 35656K->2392K(38400K)] 48744K->28536K(125952K), 0.0093836 secs] [Tim
7 64.122: [GC (Allocation Failure) [PSYoungGen: 35672K->2424K(71680K)] 87928K->54688K(159232K), 0.0032238 secs] [Tim
8 64.238: [GC (Allocation Failure) [PSYoungGen: 68984K->2392K(71680K)] 173472K->106880K(211968K), 0.0038823 secs] [T
9 64.268: [GC (Allocation Failure) [PSYoungGen: 68952K->4601K(106496K)] 173440K->16377BK(285696K), 0.0380881 secs] [
10 64.306: [Full GC (Ergonomics) [PSYoungGen: 4601K->0K(106496K)] [ParOldGen: 159177K->59152K(138752K)] 163778K->5915
11 64.729: [GC (Allocation Failure) [PSYoungGen: 101888K->35818K(137728K)] 161040K->98978K(276480K), 0.0389909 secs]
12 65.356: [GC (Allocation Failure) [PSYoungGen: 137706K->41093K(242688K)] 200866K->104261K(381440K), 0.0268902 secs]
13 67.620: [GC (Allocation Failure) [PSYoungGen: 237189K->50995K(250368K)] 300357K->114171K(389120K), 0.0390460 secs]
14 71.544: [GC (Allocation Failure) [PSYoungGen: 247091K->20555K(367616K)] 310267K->124636K(506368K), 0.0572650 secs]
15 71.601: [Full GC (Ergonomics) [PSYoungGen: 20555K->0K(367616K)] [ParOldGen: 104080K->83460K(172032K)] 124636K->834
16 83.744: [GC (System.gc()) [PSYoungGen: 299989K->15361K(372224K)] 383449K->98829K(544256K), 0.0387031 secs] [Times:
17 83.783: [Full GC (System.gc()) [PSYoungGen: 15361K->0K(372224K)] [ParOldGen: 83468K->98200K(172032K)] 98829K->9820
18 101.223: [GC (Allocation Failure) [PSYoungGen: 312832K->18402K(510464K)] 411032K->116611K(682496K), 0.0389146 secs
19 106.840: [GC (System.gc()) [PSYoungGen: 130695K->5216K(472576K)] 228903K->121835K(644608K), 0.0342911 secs] [Times
20 106.874: [Full GC (System.gc()) [PSYoungGen: 5216K->0K(472576K)] [ParOldGen: 116619K->119836K(172032K)] 121835K->1
21 146.955: [GC (Allocation Failure) [PSYoungGen: 466944K->24531K(521728K)] 586780K->144376K(693760K), 0.0383237 secs
22 159.783: [GC (System.gc()) [PSYoungGen: 231651K->13920K(520192K)] 351495K->158312K(692224K), 0.0402520 secs] [Time
23 159.824: [Full GC (System.gc()) [PSYoungGen: 13920K->0K(520192K)] [ParOldGen: 144392K->154850K(172032K)] 158312K->
24 217.273: [Full GC (Ergonomics) [PSYoungGen: 468480K->6065K(520192K)] [ParOldGen: 154850K->172027K(264704K)] 623330
25 303.309: [GC (Allocation Failure) [PSYoungGen: 468480K->28160K(637952K)] 640507K->200195K(902656K), 0.0829858 secs
26 319.552: [GC (System.gc()) [PSYoungGen: 152010K->5568K(594944K)] 324045K->205795K(859648K), 0.0516013 secs] [Times
27 319.604: [Full GC (System.gc()) [PSYoungGen: 5568K->0K(594944K)] [ParOldGen: 200227K->205580K(264704K)] 205795K->2
28 325.558: [GC (System.gc()) [PSYoungGen: 57332K->1856K(637952K)] 262912K->207436K(902656K), 0.0052389 secs] [Times:
29 325.564: [Full GC (System.gc()) [PSYoungGen: 1856K->0K(637952K)] [ParOldGen: 205580K->197443K(264704K)] 207436K->1
30 385.239: [GC (System.gc()) [PSYoungGen: 23561K->96K(638464K)] 221005K->197547K(903168K), 0.0019011 secs] [Times: u
31 385.241: [Full GC (System.gc()) [PSYoungGen: 96K->0K(638464K)] [ParOldGen: 197451K->142697K(264704K)] 197547K->142
32 389.379: [GC (Heap Dump Initiated GC) [PSYoungGen: 3257K->32K(639488K)] 145954K->142737K(904192K), 0.0010255 secs]
33 389.380: [Full GC (Heap Dump Initiated GC) [PSYoungGen: 32K->0K(639488K)] [ParOldGen: 142705K->142696K(246784K)] 1
34 5311.029: [GC (System.gc()) [PSYoungGen: 567356K->1120K(638464K)] 710053K->143824K(885248K), 0.0262403 secs] [Time
35 5311.055: [Full GC (System.gc()) [PSYoungGen: 1120K->0K(638464K)] [ParOldGen: 142704K->2895K(246784K)] 143824K->28
```

分析：

让我们来挑几条典型的日志进行分析：

第一条：`63.971: [GC (Allocation Failure) [PSYoungGen: 31073K->4210K(38400K)]`

```
31073K->4234K(125952K), 0.0049946 secs] [Times: user=0.05 sys=0.02, real=0.01 secs]
```

1. 63.971：gc 发生时，虚拟机运行了多少秒。
2. GC (Allocation Failure)： 发生了一次垃圾回收，这是一次 Minor GC 。注意它不表示只 GC 新生代，括号里的内容是 gc 发生的原因，这里的 Allocation Failure 的原因是年轻代中没有足够区域能够存放需要分配的数据而失败。
3. PSYoungGen: 使用的垃圾收集器的名字。
4. 31073K->4210K(38400K)指的是垃圾收集前->垃圾收集后(年轻代堆总大小)
5. 31073K->4234K(125952K),指的是垃圾收集前后，Java 堆的大小（总堆 125952K，堆大小包括新生代和年老代）

因此可以计算出年老代占用空间为 125952k-38400k = 87552k

6. 0.0049946 secs：整个 GC 过程持续时间
7. [Times: user=0.05 sys=0.02, real=0.01 secs]分别表示用户态耗时，内核态耗时和总耗时。也是对 gc 耗时的一个记录。

```
第二条：106.840: [GC (System.gc()) [PSYoungGen: 130695K->5216K(472576K)] 228903K->121835K(644608K), 0.0342911 secs] [Times: user=0.13 sys=0.00, real=0.03 secs]
```

先对于第一条的(Allocation Failure) 变成了(System.gc())，说明这是一次成功的垃圾回收。

```
第三条：83.783: [Full GC (System.gc()) [PSYoungGen: 15361K->0K(372224K)] [ParOldGen: 83468K->98200K(172032K)] 98829K->98200K(544256K), [Metaspace: 9989K->9989K(1058816K)], 0.3036213 secs] [Times: user=1.03 sys=0.00, real=0.30 secs]
```

对于 Full GC:

1 [PSYoungGen: 15361K->0K(372224K)] ：年轻代：垃圾收集前->垃圾收集后(年轻代堆总大小)

2 [ParOldGen: 83468K->98200K(172032K)] ：年老代：垃圾收集前->垃圾收集后(年老代堆总大小)

3 98829K->98200K(544256K)，：垃圾收集前->垃圾收集后(总堆大小)

4 [[Metaspace: 9989K->9989K(1058816K)], Metaspace 空间信息，同上

5 0.3036213 secs：整个 GC 过程持续时间

6 [Times: user=1.03 sys=0.00, real=0.30 secs] 分别表示用户态耗时，内核态耗时和总耗时。也是对 gc 耗时的一个记录。

```
第四条：71.601: [Full GC (Ergonomics) [PSYoungGen: 20555K->0K(367616K)] [ParOldGen: 104080K->83460K(172032K)] 124636K->83460K(539648K), [Metaspace: 9959K->9959K(1058816K)], 0.2146493 secs] [Times: user=0.64 sys=0.00, real=0.21 secs]
```

这里可以到 full gc 的 reason 是 Ergonomics，是因为开启了 UseAdaptiveSizePolicy，jvm 自己进行自适应调整引发的 full gc。

```
Heap
 PSYoungGen      total 638464K, used 19365K [0x00000000d5d80000, 0x00000000ffd00000, 0x0000000100000000)
  eden space 590336K, 3% used [0x00000000d5d80000,0x00000000d70696c0,0x00000000f9e00000)
  from space 48128K, 0% used [0x00000000f9e00000,0x00000000f9e00000,0x00000000fcd00000)
  to   space 47104K, 0% used [0x00000000fcf00000,0x00000000fcf00000,0x00000000ffd00000)
 ParOldGen       total 246784K, used 2895K [0x0000000081800000, 0x0000000090900000, 0x00000000d5d80000)
  object space 246784K, 1% used [0x0000000081800000,0x0000000081ad3dd8,0x0000000090900000)
 Metaspace       used 10174K, capacity 10562K, committed 11008K, reserved 1058816K
  class space    used 1119K, capacity 1236K, committed 1280K, reserved 1048576K
```

对于底下 Heap 的输出情况，和上面是完全一致的。
只是增加了堆中每个部分 total 总大小，used 使用情况。
PSYoungGen 分为 eden space 530336K, 3% used ， from space 48128K, 0% used ， to space 47104K, 0% used 三个部分，分别显示了它们的大小和 used 比例。
ParOldGen 分为 object space246784K, 1% used，显示了其大小和 used 比例。
Metaspace 中 used 大小为 10174k。

根据以上日志中显示出垃圾回收的异常情况，在报告的后面说明了处理方法。

3.3.2 用 jstat 命令行工具的-gc 和-gcutil 参数

```
C:\Users\admin>jps
38672 MyOrbitScenes
39836
41308 Jps
```

先用 jps 命令列出当前的 Java 进程列表，选择进程号 38672 的进程观察。

jstat -gc -h3 38672 2500 10
解释：监控 gc 的信息，每三行输出一次表头，监控的进程 pid 为 38672，每 2500ms 输出一次信息，一共输出 10 次。

```
C:\Users\admin>jstat -gc -h3 38672 2500 10
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC     MU    CCSC   CCSU   YGC     YGCT    FGC
FGCT     GCT
96256.0 109568.0 88243.8  0.0   160256.0 71919.8   163328.0   120631.4  6144.0 5691.8 768.0  617.0      20    0.653   2
     0.059    0.711
96256.0 109568.0 88243.8  0.0   160256.0 107176.1  163328.0   120631.4  6144.0 5691.8 768.0  617.0      20    0.653   2
     0.059    0.711
96256.0 109568.0 88243.8  0.0   160256.0 139227.9  163328.0   120631.4  6144.0 5691.8 768.0  617.0      20    0.653   2
     0.059    0.711
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC     MU    CCSC   CCSU   YGC     YGCT    FGC
FGCT     GCT
123392.0 109568.0  0.0   97021.7 160256.0  9614.6   163328.0   120631.4  6144.0 5691.8 768.0  617.0      21    0.729   2
     0.059    0.787
123392.0 109568.0  0.0   97021.7 160256.0 44868.3   163328.0   120631.4  6144.0 5691.8 768.0  617.0      21    0.729   2
     0.059    0.787
123392.0 109568.0  0.0   97021.7 160256.0 80122.1   163328.0   120631.4  6144.0 5691.8 768.0  617.0      21    0.729   2
     0.059    0.787
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC     MU    CCSC   CCSU   YGC     YGCT    FGC
FGCT     GCT
123392.0 109568.0  0.0   97021.7 160256.0 108966.1 163328.0   120631.4  6144.0 5691.8 768.0  617.0      21    0.729   2
     0.059    0.787
123392.0 109568.0  0.0   97021.7 160256.0 137809.8 163328.0   120631.4  6144.0 5691.8 768.0  617.0      21    0.729   2
     0.059    0.787
123392.0 137216.0 105021.7  0.0   149504.0  5980.0   163328.0   120631.4  6144.0 5691.8 768.0  617.0      22    0.820
2    0.059    0.879
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC     MU    CCSC   CCSU   YGC     YGCT    FGC
FGCT     GCT
123392.0 137216.0 105021.7  0.0   149504.0 38869.9   163328.0   120631.4  6144.0 5691.8 768.0  617.0      22    0.820
2    0.059    0.879
```

分析：

新生代（Young）：

S0C 第一个幸存者区的总大小；S1C 第二个幸存者区的总大小

S0U 第一个幸存者区的已使用的大小；S1U 第二个幸存者区的已使用的大小

EC 伊甸区的总大小；EU 伊甸区已使用的大小

老年代（Old）

OC 老年区的总大小；OU 老年区已使用的大小

Metaspace：

MC 方法区的总大小；MU 方法区已使用大小

CCSC 压缩类总大小；CCSU 压缩类已使用大小

其他：

YGC young gc 次数；YGCT young gc 总时间

FGC full gc 次数；FGCT full gc 总时间

GCT gc 的总时间

jstat -gcutil -h3 38672 2500 10

解释：监控 gc 的信息，每三行输出一次表头，监控的进程 pid 为 38672，每 2500ms 输出一次信息，一共输出 10 次。

```
C:\Users\admin>jstat -gcutil -h3 38672 2500 10
  S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT     GCT
  0.00  82.02  78.00  73.86  92.64  80.34     23    0.908      2    0.059    0.967
  0.00  82.02  96.00  73.86  92.64  80.34     23    0.908      2    0.059    0.967
 79.45   0.00  14.00  73.86  92.64  80.34     24    1.018      2    0.059    1.077
  S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT     GCT
 79.45   0.00  34.00  73.86  92.64  80.34     24    1.018      2    0.059    1.077
 79.45   0.00  56.00  73.86  92.64  80.34     24    1.018      2    0.059    1.077
 79.45   0.00  74.00  73.86  92.64  80.34     24    1.018      2    0.059    1.077
  S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT     GCT
 79.45   0.00  92.00  73.86  92.64  80.34     24    1.018      2    0.059    1.077
  0.00  77.22   4.00  73.86  92.64  80.34     25    1.120      2    0.059    1.178
  0.00  77.22   4.00  73.86  92.64  80.34     25    1.120      2    0.059    1.178
  S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT     GCT
  0.00  77.22   4.00  73.86  92.64  80.34     25    1.120      2    0.059    1.178
```

分析：

S0、S1 代表两个 Survivor 区；

E 代表 Eden 区；

O（Old）代表老年代；

M 代表 Metaspace

YGC（Young GC）代表 Minor GC；

YGCT 代表 Minor GC 耗时；

FGC（Full GC）代表 Full GC 耗时；

GCT 代表 Minor & Full GC 共计耗时

分析到这儿，程序的 gc 其实是发生了异常情况的，具体的解决和环境配置在后文说明。

3.3.3 使用 jmap -heap 命令行工具

```
C:\Users\admin>jmap -heap 38672
Attaching to process ID 38672, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.201-b09

using thread-local object allocation.
Parallel GC with 4 thread(s)
```

GC 使用的算法：thread-local object allocation.

```
Heap Configuration:
   MinHeapFreeRatio         = 0
   MaxHeapFreeRatio         = 100
   MaxHeapSize              = 2122317824 (2024.0MB)
   NewSize                  = 44564480 (42.5MB)
   MaxNewSize               = 707264512 (674.5MB)
   OldSize                  = 89653248 (85.5MB)
   NewRatio                 = 2
   SurvivorRatio            = 8
   MetaspaceSize            = 21807104 (20.796875MB)
   CompressedClassSpaceSize = 1073741824 (1024.0MB)
   MaxMetaspaceSize         = 17592186044415 MB
   G1HeapRegionSize         = 0 (0.0MB)
```

heap 的配置情况：以上多个参数的意义

MinHeapFreeRatio：JVM 堆最小空闲比率 0

MaxHeapFreeRatio：JVM 堆最大空闲比率 100

MaxHeapSize：JVM 堆的最大大小 2122317824

NewSize：JVM 堆的‘新生代’的默认大小 44564480

MaxNewSize：VM 堆的‘新生代’的最大大小 707264512

OldSize：JVM 堆的‘老生代’的大小 89653248

NewRatio：‘新生代’和‘老生代’的大小比率 2

SurvivorRatio：年轻代中 Eden 区与 Survivor 区的大小比值 8

MetaspaceSize=<value>: JVM 堆的 Metaspace 的初始大小 21807104

MaxMetaspaceSize=<value>:JVM 堆的 Metaspace 的最大大小

```
Heap Usage:
PS Young Generation
Eden Space:
   capacity = 142082048 (135.5MB)
   used     = 5683392 (5.42010498046875MB)
   free     = 136398656 (130.07989501953125MB)
   4.000077476360701% used
From Space:
   capacity = 168296448 (160.5MB)
   used     = 129955536 (123.93525695800781MB)
   free     = 38340912 (36.56474304199219MB)
   77.21822863427278% used
To Space:
   capacity = 181403648 (173.0MB)
   used     = 0 (0.0MB)
   free     = 181403648 (173.0MB)
   0.0% used
PS Old Generation
   capacity = 167247872 (159.5MB)
   used     = 123526552 (117.8041000366211MB)
   free     = 43721320 (41.695899963378906MB)
   73.85836992891605% used

3355 interned Strings occupying 266264 bytes.
```

堆的使用情况：
新生代区：Eden 区内存分布；From Space：其中一个 Survivor 区的内存分布；To Space：另一个 Survivor 区的内存分布
年老代区：内存分布

补充：jmap -histo 命令行工具
说明：instances（实例数）、bytes（大小）、classs name（类名）。它基本是按照使用使用大小逆序排列的。

```
C:\Users\admin>jmap -histo 38672

 num     #instances         #bytes  class name
----------------------------------------------
   1:       1615628       121679080  [C
   2:       1610139        38643336  java.lang.String
   3:        322822        23073928  [Ljava.lang.Object;
   4:        640927        20509664  java.util.HashMap$Node
   5:        320000        20480000  physicalobject.Planet
   6:        323158         7755792  java.util.ArrayList
   7:        320001         7680024  java.lang.Double
   8:        320000         7680000  track.Track
   9:            40         4269584  [Ljava.util.HashMap$Node;
  10:         25272         3361840  [I
  11:          3471         1382616  [Ljava.lang.String;
  12:          3121          848656  [Z
  13:          3565          256680  java.util.regex.Pattern
  14:          3565          228160  java.util.regex.Matcher
  15:          3566          199696  [Ljava.util.regex.Pattern$GroupHead;
  16:          4902          156864  sun.misc.FloatingDecimal$ASCIIToBinaryBuffer
  17:           461          150176  [B
  18:          1050          120408  java.lang.Class
  19:          3565           85560  java.util.regex.Pattern$Start
  20:          3565           85560  java.util.regex.Pattern$TreeInfo
  21:          3120           74880  java.util.regex.Pattern$BitClass
  22:          1783           71320  java.util.ArrayList$SubList
  23:           791           31640  java.util.TreeMap$Entry
  24:           890           28480  java.util.regex.Pattern$Curly
  25:           890           21360  java.util.regex.Pattern$GroupHead
```

其中：[C is a char[]　[I is a int[]　[B is a byte[]

3.3.4 使用 jmap -clstats 命令行工具

以管理员身份运行 cmd，可以得到如下图。

jmap -clstats 39864 打印 Java 类加载器的智能统计信息，对于每个类加载器而言，它的名称，活跃度，地址，父类加载器，它所加载的类的数量和大小都会被打印。此外，包含的字符串数量和大小也会被打印。

```
C:\WINDOWS\system32>jmap -clstats 39864
Attaching to process ID 39864, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.201-b09
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>     852     1664015   null          live    <internal>
0x00000000d5e0fbd0      4       4572      null          live    sun/misc/Launcher$ExtClassLoader@0x000000010000fc80
0x00000000d5e26598      22      33717   0x00000000d5e0fbd0      live    sun/misc/Launcher$AppClassLoader@0x000000010
000f8d8
0x00000000d617f328      0       0       0x00000000d5e26598      dead    java/util/ResourceBundle$RBClassLoader@0x000
000010009b7b0

total = 4       878     1702304     N/A         alive=3, dead=1     N/A
```

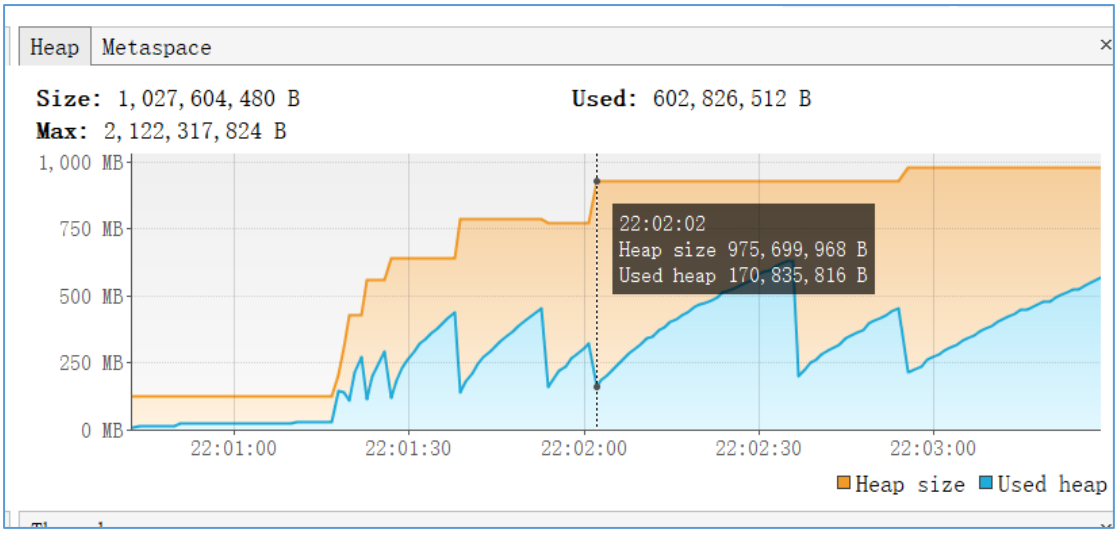
3.3.5 使用 jmap -permstat 命令行工具

JDK 版本在 8 之后，不使用该指令。

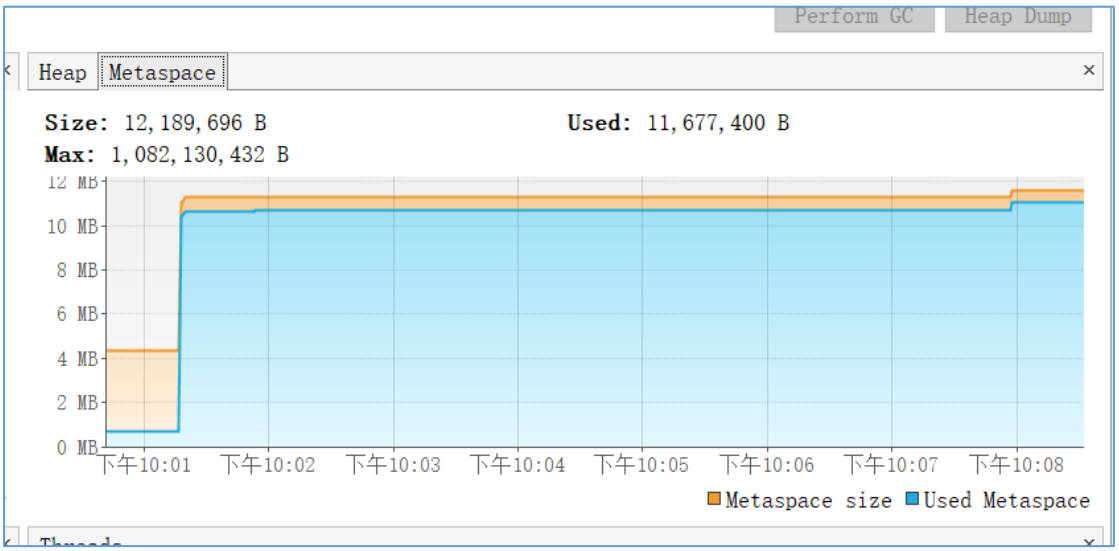
3.3.6 使用 JMC/JFR、jconsole 或 VisualVM 工具

堆的分配和占用:

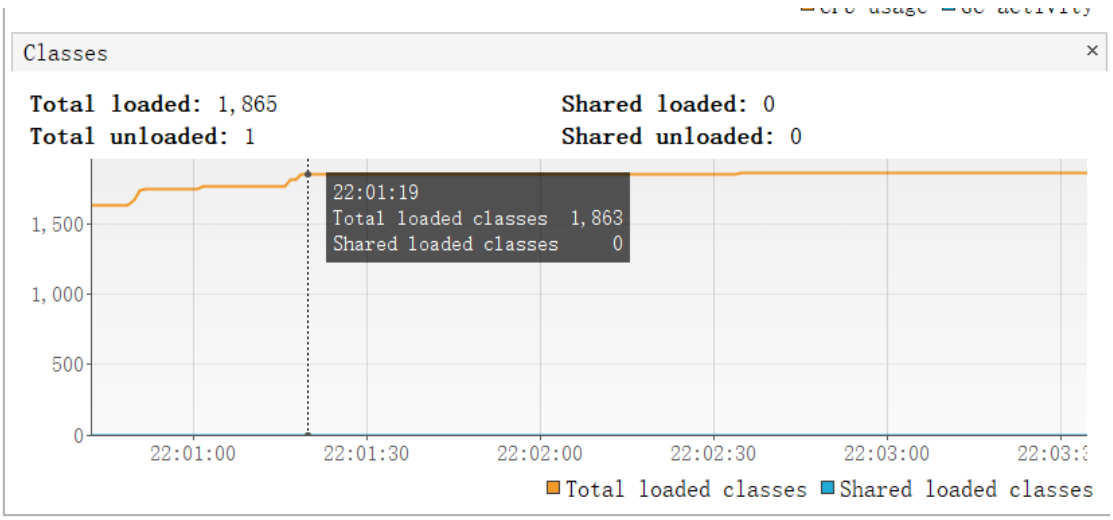
我们可以看出，在 readFile 开始运行的时候，Heap 的使用突然增大。由于构建的对象较多，整体上内存的 heap 的使用率也是比较高的。

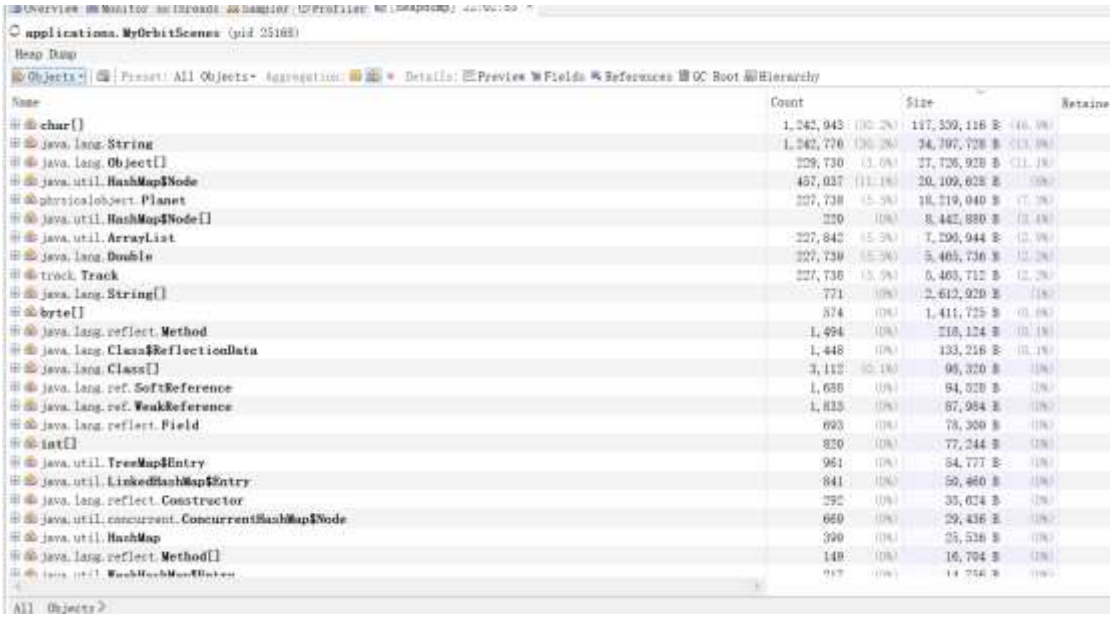


Metaspace 情况:



装载的类的实例统计:





3.3.7 分析垃圾回收过程

根据分代的垃圾回收策略，我们对垃圾回收的过程可以做出如下的总结。

首先从 jstat -gc 打印出的结果，我们可以清楚的看到年轻代分区的情况

1. 新生代内存按照 8:1:1 的比例分为一个 eden 区和 S0、S1 区。大部分对象在 Eden 区中生成，回收时先将 eden 区存活对象复制到一个 S0 区，然后清空 eden 区。
2. 当这个 S0 区也存放满了时，则将 eden 区和 S0 区存活对象复制到另一个 S1 区，然后清空 eden 和这个 S1 区，此时 S0 区是空的，然后将 S0 区和 s1 区交换，即保持 S1 区为空，如此往复。
3. 这里也解释了为什么命令行 jstat -gc 每个 2500ms 打印结果的时候，S0 和 S1 的大小交替变换。

然后再从日志分析结果，当我-XX:NewSize=500m：设置年轻代初始值为 500M 时。发现 gc 时大部分的的在年轻代会被回收，大约只有有 20M 大小的对象会进入老年代。

1. 一方面根据我程序实现时，对象的存活周期大多较短，另一方面，所有新生成的对象首先都是放在年轻代的。年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象。
2. 当 survivor1 区不足以存放 eden 和 S0 的存活对象时，就将存活对象直接存放到老年代。若是老年代也满了就会触发一次 Full GC，也就是新生代、老年代都进行回收
3. 新生代发生的 GC 也叫做 Minor GC，MinorGC 发生频率比较高(不一定等 Eden 区满了才触发)

我们再结合着 jmap -heap 打印出的结果来分析，以下主要讨论年老代和 Metaspace。

1. 在年轻代中经历了 N 次垃圾回收后仍然存活的对象，就会被放到年老代中。因此，可以认为年老代中存放的都是一些生命周期较长的对象。
2. 当老年代内存满时触发 Full GC，Full GC 发生频率比较低，老年代对象存活时间比较长，存活率标记高。
3. Metaspace 用于存放静态文件，如 Java 类、方法等。结合着日志，我们会发现 Metaspace 的大小变化总是不大，事实上 Metaspace 对垃圾回收没有显著影响。Metaspace 包含 JVM 用于描述应用程序中类和方法的元数据，是由 JVM 在运行时根据应用程序使用的类来填充的。

3.3.8 配置 JVM 参数并发现优化的参数配置

先说明一下参数配置的依据：

1. 在 gc 日志，我们很明显的看出 Minor gc 的时候，显示 Allocation Failure 的原因是年轻代中没有足够区域能够存放需要分配的数据而失败。因此我们需要增加年轻代的大小。刚开始-XX:NewSize=650m：设置年轻代初始值为 650M。发现还是不够，于是又加大了 1g。

2. 从 jstat -gc 的打印结果，发现每次 ygc 之后 survivor 空间基本是空的，说明新生对象产生快，生命周期也短，原本设计的 survivor 空间没有派上用场。因此考虑调整 young generation 分区的大小设置。

参数配置:

-XX:InitialHeapSize=2120867840
-XX:MaxHeapSize=2120867840
-XX:MaxNewSize=2147483648
-XX:NewSize=2147483648
-XX:+PrintGC
-XX:+PrintGCDetails
-XX:+PrintGCTimeStamps
-XX:+UseCompressedClassPointers

-XX:+UseCompressedOops
-XX:-UseLargePagesIndividualAllocation
-XX:+UseParallelGC

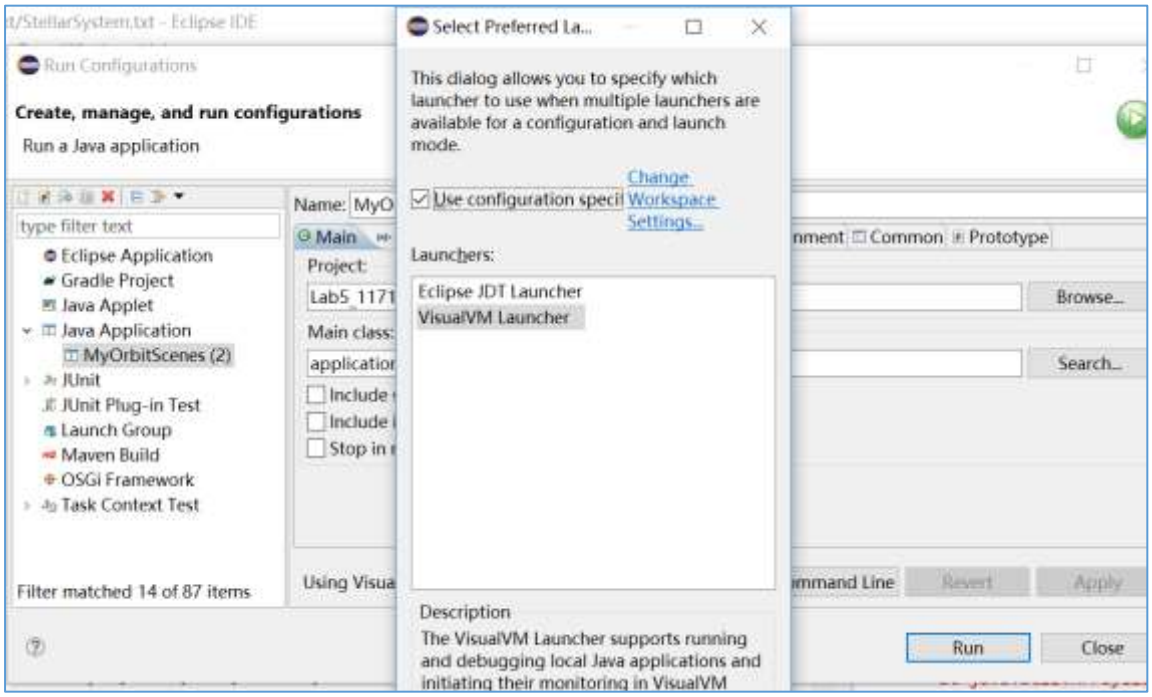
重新执行（1）：GC (Allocation Failure)和 Full GC (Ergonomics)终于没了，而且我们可以发现垃圾回收的时间减少了很多。

```
MyOrbitScenes.java   StellarSystem.java   gc_info.log
Java HotSpot(TM) 64-Bit Server VM (25.201-b09) for windows-amd64 JRE (1.8.0_201-b09), built on Dec 15 2018 18:36:
Memory: 4k page, physical 8284640k(2321288k free), swap 19294688k(8678300k free)
CommandLine flags: -XX:InitialHeapSize=2120867840 -XX:MaxHeapSize=2120867840 -XX:MaxNewSize=2147483648 -XX:NewSiz
47.645: [GC (System.gc()) [PSYoungGen: 1360895K->92614K(1813504K)] 1360895K->92630K(1814016K), 0.0850149 secs] [T
47.731: [Full GC (System.gc()) [PSYoungGen: 92614K->92350K(1813504K)] [ParOldGen: 16K->11K(512K)] 92630K->92362K(
102.830: [GC (System.gc()) [PSYoungGen: 920128K->132301K(1813504K)] 920139K->132321K(1814016K), 0.1442939 secs] [
102.974: [Full GC (System.gc()) [PSYoungGen: 132301K->132032K(1813504K)] [ParOldGen: 19K->15K(512K)] 132321K->132
160.597: [GC (System.gc()) [PSYoungGen: 922387K->168157K(1813504K)] 922402K->168181K(1814016K), 0.1443838 secs] [
160.742: [Full GC (System.gc()) [PSYoungGen: 168157K->167308K(1813504K)] [ParOldGen: 23K->506K(512K)] 168181K->16
216.022: [Full GC (System.gc()) [PSYoungGen: 787790K->196370K(1813504K)] [ParOldGen: 506K->496K(512K)] 788296K->1
247.181: [Full GC (System.gc()) [PSYoungGen: 432980K->196914K(1813504K)] [ParOldGen: 496K->494K(512K)] 433477K->1
268.862: [GC (Heap Dump Initiated GC) [PSYoungGen: 264329K->142413K(1813504K)] 264823K->142916K(1814016K), 0.1280
268.990: [Full GC (Heap Dump Initiated GC) [PSYoungGen: 142413K->142200K(1813504K)] [ParOldGen: 502K->496K(2048K)
2907.250: [GC (System.gc()) [PSYoungGen: 702058K->1952K(1813504K)] 702555K->2456K(1815552K), 0.0107346 secs] [Tim
2907.261: [Full GC (System.gc()) [PSYoungGen: 1952K->0K(1813504K)] [ParOldGen: 504K->1929K(2048K)] 2456K->1929K(1
Heap
 PSYoungGen      total 1813504K, used 37858K [0x0000000081880000, 0x0000000100000000, 0x0000000100000000)
  eden space 1554944K, 2% used [0x0000000081880000,0x0000000083d78980,0x00000000e0700000)
  from space 258560K, 0% used [0x00000000e0700000,0x00000000e0700000,0x00000000f0380000)
  to   space 258560K, 0% used [0x00000000f0380000,0x00000000f0380000,0x0000000100000000)
 ParOldGen       total 2048K, used 1929K [0x0000000081200000, 0x0000000081400000, 0x0000000081880000)
  object space 2048K, 94% used [0x0000000081200000,0x00000000813e2538,0x0000000081400000)
 Metaspace       used 10069K, capacity 10422K, committed 10752K, reserved 1058816K
  class space    used 1124K, capacity 1226K, committed 1280K, reserved 1048576K
```

重新执行（2）：
截图太多就不粘了，由于我们增大了 young generation，同时 old generation 会有一定的减小，但是观察 YGCT 代表 Minor GC 耗时；FGC（Full GC）代表 Full GC 耗时；GCT 代表 Minor & Full GC 共计耗时。这些耗时都有减少。

3.4 Dynamic Program Profiling

配置环境的部分在前面实验环境配置的部分已经说明，运行 VisualVM 的方法如下图：



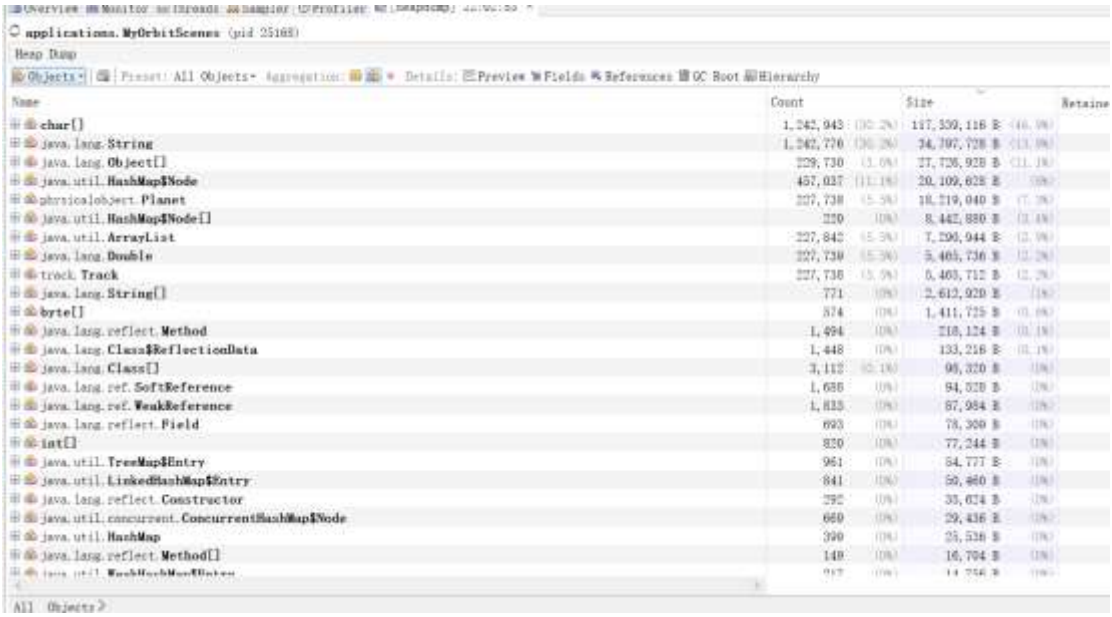
在选择读文件阶段，就将 VisualVM profiler 设定为“自动更新结果”，选择 CPU Profiling 或者 memory profiling，从头开始动态监控程序的性能。

在某一时刻截图如下**（关于 CPU 和 memory 后面分析）：**

在监视标签内，我们可以看到实时的应用程序内存堆以及 Metaspace 的使用情况,CPU 的使用率和垃圾回收活动。

此时 heapdump 情况如下：堆转储的类视图

堆转储的摘要包括转储的文件大小、路径等基本信息，运行的系统环境信息，也可以显示所有的线程信息。如图中可以看出 char[]和 java.lang.String 占用了较大内存

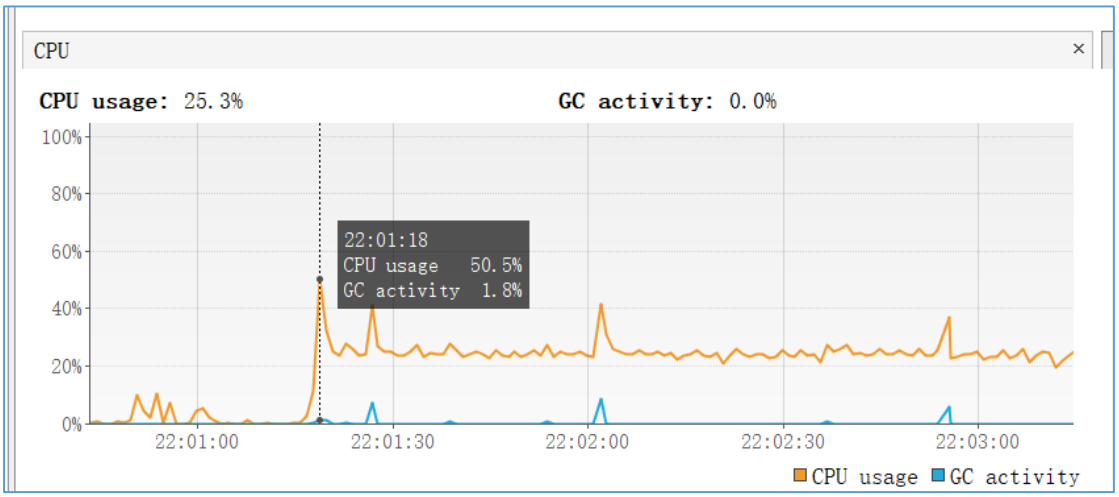


3.4.1 使用 JMC 或 VisualVM 进行 CPU Profiling

我们可以通过 VisualVM 的监视标签和 Profiler 标签对应用程序进行 CPU 性能分析。

一. 先看监视标签

在监视标签内，我们可以查看 CPU 的使用率以及垃圾回收活动对性能的影响。在程序刚启动时，CPU usage 达到了最高的 50.5%。在其他时候，过高的 CPU 使用率可能是由于我们的轨道系统构建中中存在低效的代码，整体上看，垃圾回收活动并不频繁，没有占用了较高的 CPU 资源。



二. 再看 CPU profiler 标签

在 CPU 性能分析 profiler 标签 ,VisualVM 会检测应用程序所有的被调用的方法。当进入一个方法时，线程会发出一个“method entry”的事件，当退出方法时同样会发出一个“method exit”的事件，这些事件都包含了时间戳。然后 VisualVM 会把每个被调用方法的总的执行时间和调用的次数按照运行时长展示出来。

Self time：我们按照 Total Time 进行排序之后，发现排在第一的是 Self time，这个其实对

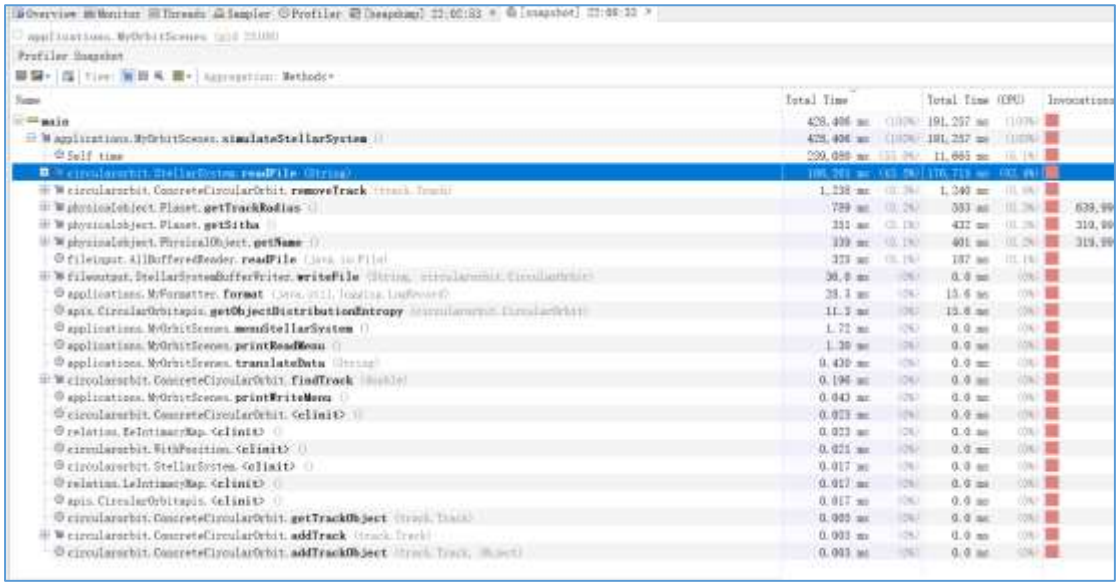
于我们不重要，解释起来，Self time 是在方法本身花费的挂钟时间（包括等待/休眠的时间），它是时间处理器时间，因此它不包括等待，睡眠等所花费的时间。

circularorbit.StellarSystem.readFile (String)：对于方法执行的时间，不出所料，耗时最长的是 readFile，它主要运行的时间不是在读入一篇 txt 文件，这个只需要几百 ms，而是在将物体添加到轨道上并且根据半径进行排序。简单地说就是通过读入的文件搭建轨道是个漫长的过程。

circularorbit.ConcreteCircularOrbit.removeTrack (track.Track)：移除一条轨道也需要花费不少的时间，因为我们需要先移除轨道上的物体，然后再删除轨道

fileoutput.StellarSystemBufferWriter.writeFile (String, circularorbit.CircularOrbit)：写文件运行所需要的时间也在考虑之中，毕竟有几十万行的数据。

其他方法：主要花费的时间在于没有优化彻底的查找方案，比如获得一个物体的某个属性，需要去 set 中遍历，造成了不少的时间消耗；另外在控制台的打印，写入日志文件也是需要一定时间的。

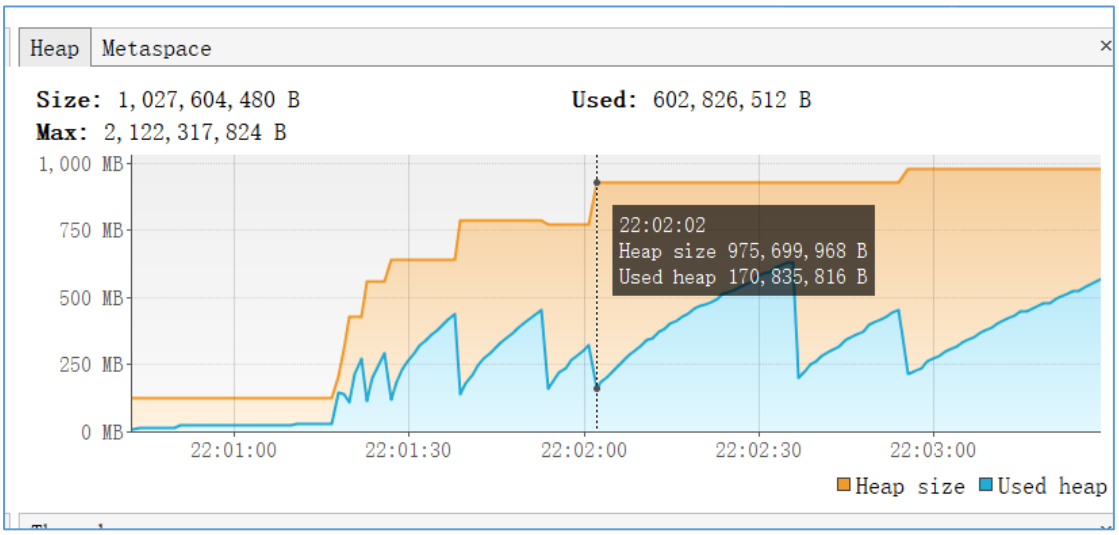


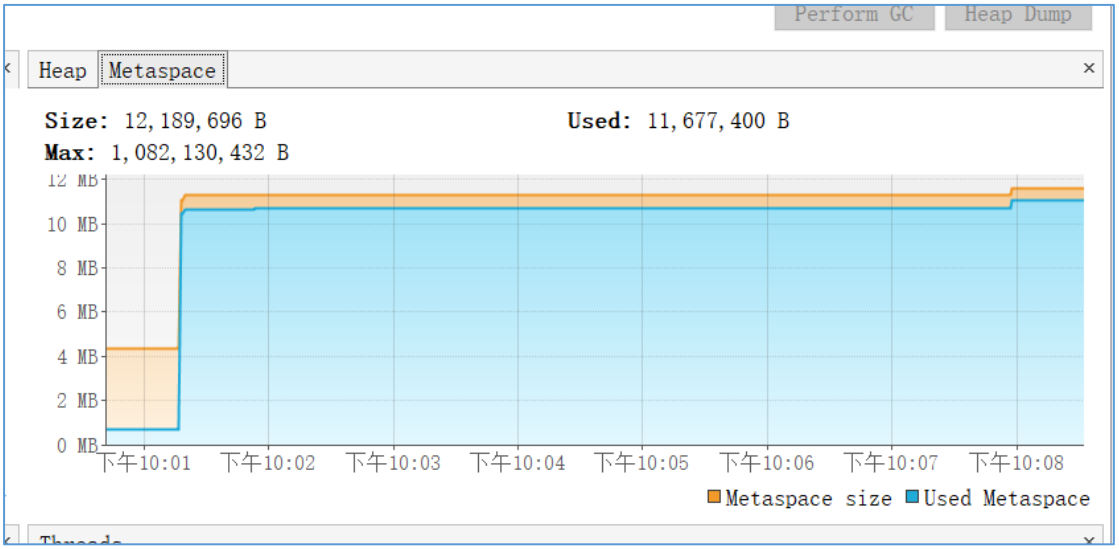
3.4.2 使用 VisualVM 进行 Memory profiling

一. 监视标签

在监视标签内，我们可以看到实时的应用程序内存堆以及 Metaspace 区域的使用情况。

我们可以看出，由于构建的对象较多，内存的 heap 的使用率也是比较高的



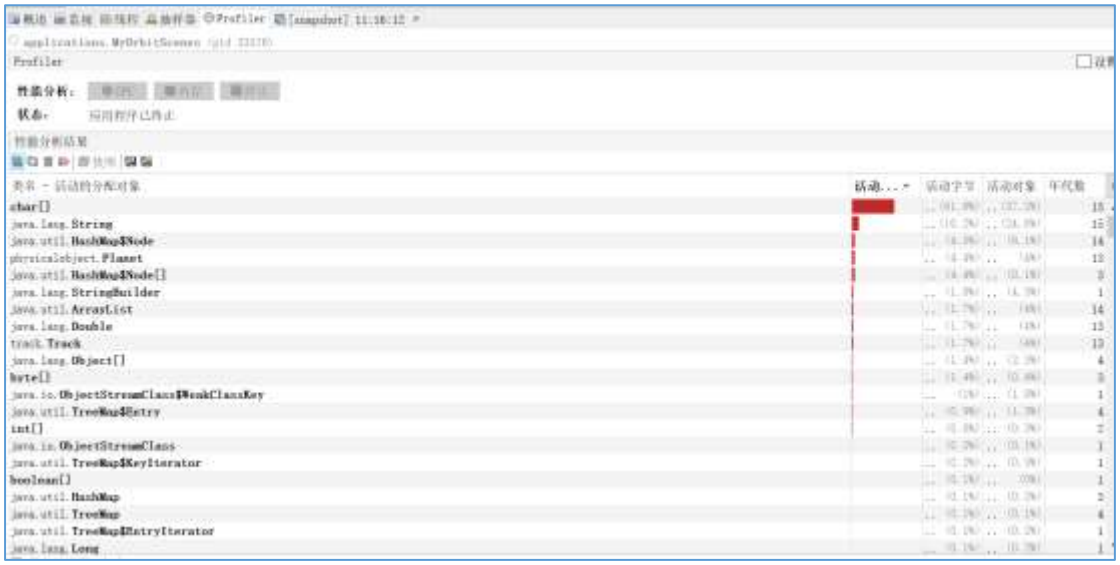


二. memory profiler

本以为 memory profiler 运行起来和 CPU profiler 一样，可是同样的操作却一直报错，或者就是已经在运行，却一直不显示活动的分配对象。

后来的处理方式是以管理员身份运行 cmd，此时打开 java visualvm，问题才得以解决。

```
C:\WINDOWS\system32>jvisualvm
```



分析：

通常考虑，字符串对象中就有一个 char[]数组，其他对象中也可能封装 char[]数组，先检查一下 String 对象有多少个，算上其他可能封装 char[]数组的对象，也差不多这个数值了。同理 int[]和 byte[]也是这个原因占用了很多内存。

但是事实上我们更应关注自己所创建的实例，因为我们解析的时候不断创建字符串，因此 String 对象毫无疑问有很多。从源码上来看,HashSet 实际上为(key,null)类型的 HashMap,因此我们使用 hashset 后会存在很多 java.util.HashMap$Node[]。

其余的内存占用居前的类型就很好理解了，因为 Planet、track 等正是我们构建轨道一直在使用的类型。

总体分析，这些耗费内存最多类型的执行正常。

3.5 Memory Dump Analysis and Performance Optimization

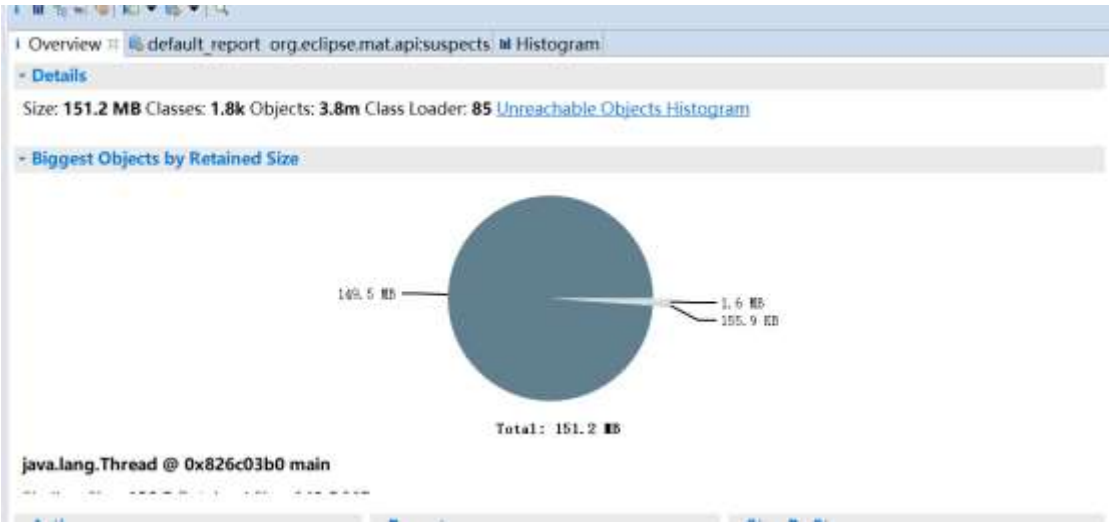
3.5.1 内存导出

使用 VisualVm 进行堆转储后，导出.hprof 文件，并将其另存为到本地。

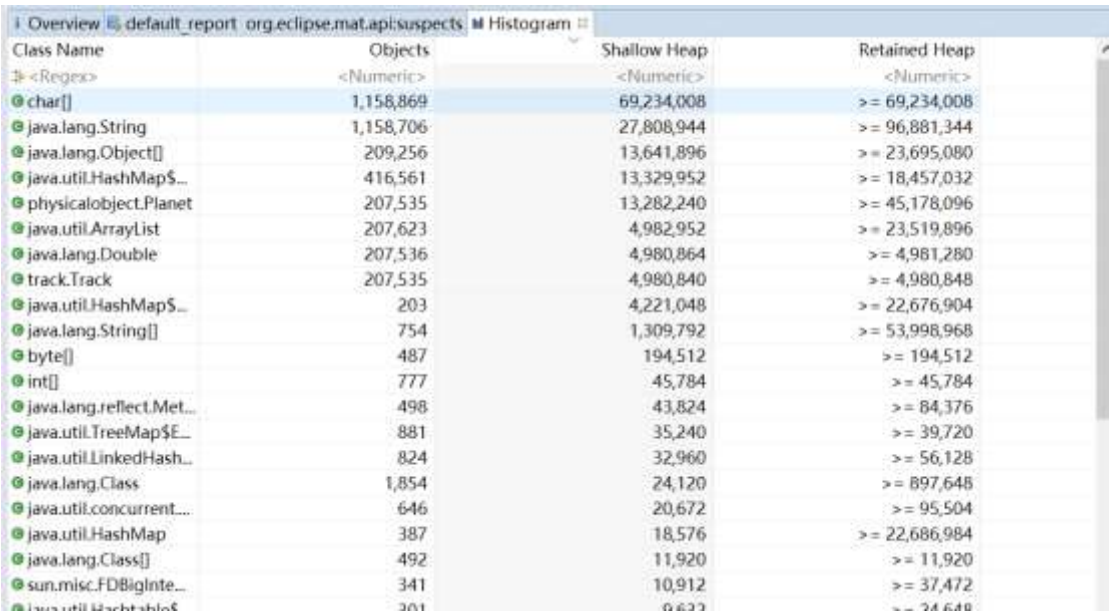
3.5.2 使用 MAT 分析内存导出文件

Overview：

内存在“<system class loader>”加载的“java.lang.Thread”的一个实例中累积。



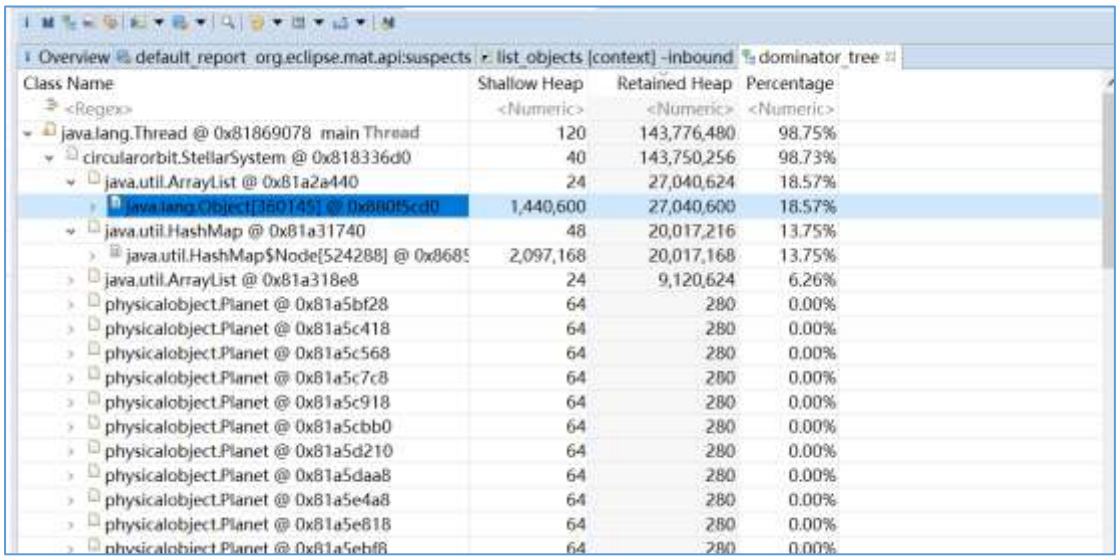
histogram 视图：展示了此刻内存中存储的各类型的实例数量以及所占用内存的情况。



Class Name	Objects	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
char[]	1,158,869	69,234,008	>= 69,234,008
java.lang.String	1,158,706	27,808,944	>= 96,881,344
java.lang.Object[]	209,256	13,641,896	>= 23,695,080
java.util.HashMap$...	416,561	13,329,952	>= 18,457,032
physicalobject.Planet	207,535	13,282,240	>= 45,178,096
java.util.ArrayList	207,623	4,982,952	>= 23,519,896
java.lang.Double	207,536	4,980,864	>= 4,981,280
track.Track	207,535	4,980,840	>= 4,980,848
java.util.HashMap$...	203	4,221,048	>= 22,676,904
java.lang.String[]	754	1,309,792	>= 53,998,968
byte[]	487	194,512	>= 194,512
int[]	777	45,784	>= 45,784
java.lang.reflect.Met...	498	43,824	>= 84,376
java.util.TreeMap$E...	881	35,240	>= 39,720
java.util.LinkedHash...	824	32,960	>= 56,128
java.lang.Class	1,854	24,120	>= 897,648
java.util.concurrent....	646	20,672	>= 95,504
java.util.HashMap	387	18,576	>= 22,686,984
java.lang.Class[]	492	11,920	>= 11,920
sun.misc.FDBigInte...	341	10,912	>= 37,472

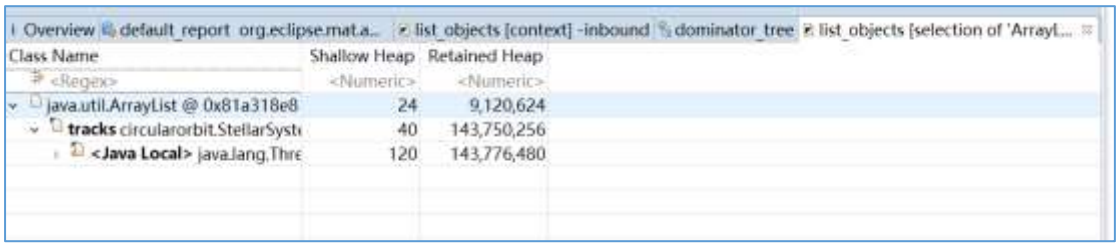
dominator tree 视图：

各实例之间的引用关系:



Class Name	Shallow Heap	Retained Heap	Percentage
<Regex>	<Numeric>	<Numeric>	<Numeric>
java.lang.Thread @ 0x81869078 main Thread	120	143,776,480	98.75%
circularorbit.StellarSystem @ 0x818336d0	40	143,750,256	98.73%
java.util.ArrayList @ 0x81a2a440	24	27,040,624	18.57%
java.lang.Object[360145] @ 0x880f5cd0	1,440,600	27,040,600	18.57%
java.util.HashMap @ 0x81a31740	48	20,017,216	13.75%
java.util.HashMap$Node[524288] @ 0x8685	2,097,168	20,017,168	13.75%
java.util.ArrayList @ 0x81a318e8	24	9,120,624	6.26%
physicalobject.Planet @ 0x81a5bf28	64	280	0.00%
physicalobject.Planet @ 0x81a5c418	64	280	0.00%
physicalobject.Planet @ 0x81a5c568	64	280	0.00%
physicalobject.Planet @ 0x81a5c7c8	64	280	0.00%
physicalobject.Planet @ 0x81a5c918	64	280	0.00%
physicalobject.Planet @ 0x81a5cbb0	64	280	0.00%
physicalobject.Planet @ 0x81a5d210	64	280	0.00%
physicalobject.Planet @ 0x81a5daa8	64	280	0.00%
physicalobject.Planet @ 0x81a5e4a8	64	280	0.00%
physicalobject.Planet @ 0x81a5e818	64	280	0.00%
physicalobject.Planet @ 0x81a5ebf8	64	280	0.00%

右键选择 List Objects -> with incoming reference，这可以列出 ArrayList 中对象的引用路径:

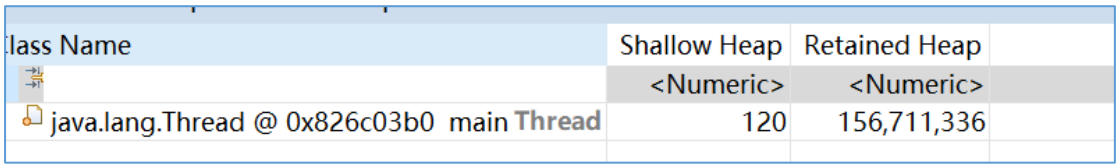


Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.util.ArrayList @ 0x81a318e8	24	9,120,624
tracks circularorbit.StellarSyste	40	143,750,256
<Java Local> java.lang.Thre	120	143,776,480

查看一个对象到 RC Roots 的引用链

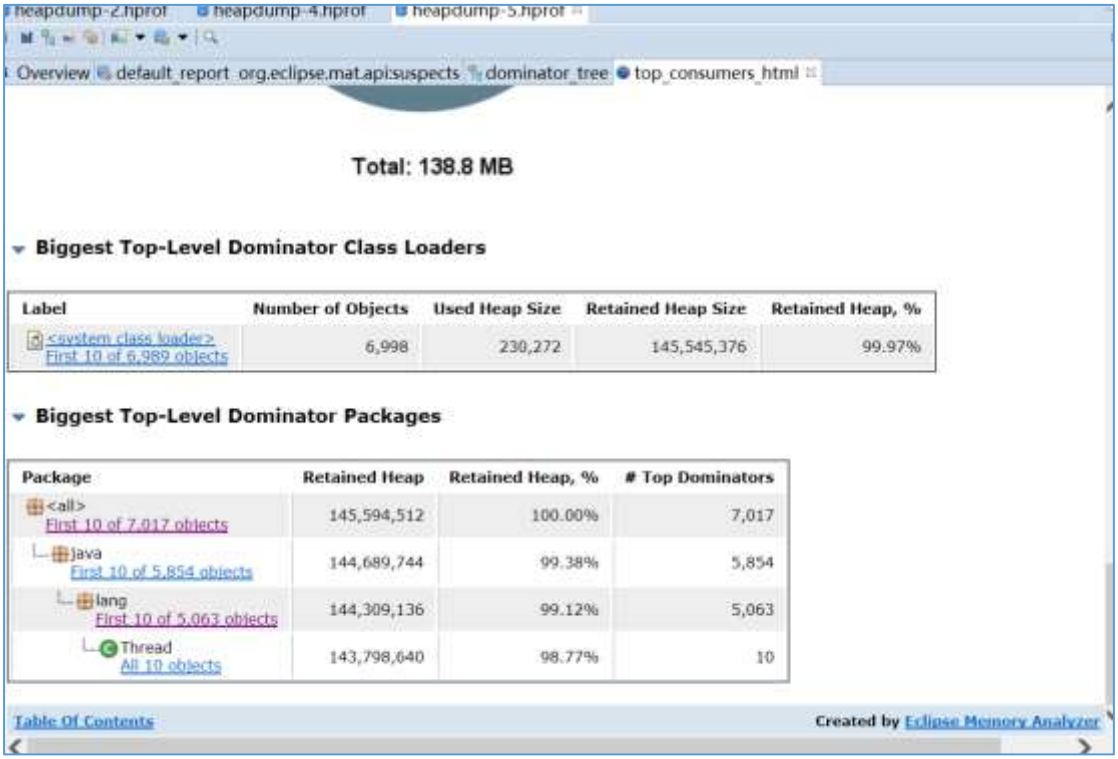
通常在排查内存泄漏的时候，我们会选择 exclude all phantom/weak/soft etc.references，意思是查看排除虚引用/弱引用/软引用等的引用链，因为被虚引用/弱引用/软引用的对象可以直接被 GC 给回收，我们要看的就是某个对象否还存在 Strong 引用链(在导出 HeapDump 之前要手动出发 GC 来保证)，如果有，则说明存在内存泄漏，然后再去排查具体引用。

据观察，几乎所有的 GC root 都是 Thread 或者 System Class。



lass Name	Shallow Heap	Retained Heap
	<Numeric>	<Numeric>
java.lang.Thread @ 0x826c03b0 main Thread	120	156,711,336

top consumer 视图：

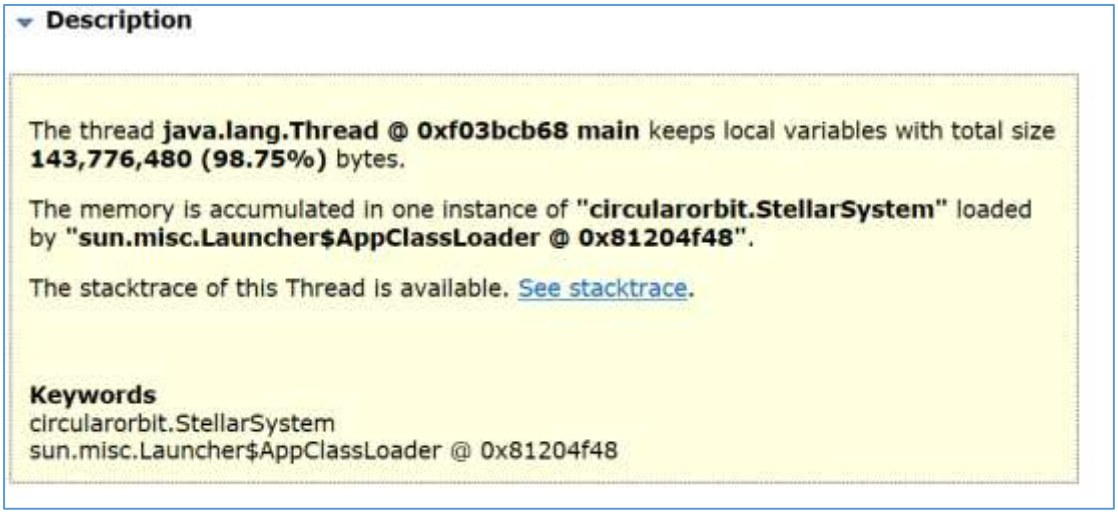


查看 leak suspects report：

刚开始我 java.lang.Thread @ 0xf03bcb68 main 占用堆的比例特别高，于是系统地对 leak suspects report 进行的分析，最后得出的结论是没有发生内存泄漏，主要的原因是我程序中对象的存活时间比较长，list 和 hashmap 常伴随着对象之间的映射关系，因此在它们 gc 的回收就会比较延后。

具体的分析过程如下：

1. 内存疑似泄漏的描述如下。



2. 主要看以下的两张图，首先就可以看到 Circularorbit.StellarSystem 的 shallow heap 和 retained heap 是非常不成比例的，这说明这个在其中持有了大量对象的引用，。于是我们根据 retained heap 的大小逐步对其支配树进行展开，可以发现 heap 占用较高的主要是 Arraylist、HashMap 和 Planet。当我们选择 List objects ->with incoming references，可以对它们的归属了解得更详细。

两个 Arraylist 主要是 track 的列表，和每个轨道上轨道物体嵌套列表；HashMap 是在 WithPositon 类中设计的每个物体映射一个角度产生的，而由于文件较大，产生较多的 Planet 的也是情理之中。

Class Name	Shallow Heap	Retained Heap	Percentage
java.lang.Thread @ 0xf03bcb68 main	120	143,776,480	98.75%
circularorbit.StellarSystem @ 0xf038c3e8	40	143,750,256	98.73%
java.util.ArrayList @ 0xf047a328	24	27,040,624	18.57%
java.util.HashMap @ 0xf047d140	48	20,017,216	13.75%
java.util.ArrayList @ 0xf038c540	24	9,120,624	6.26%
physicalobject.Planet @ 0x8127a7f0	64	280	0.00%
physicalobject.Planet @ 0x8127a9a8	64	280	0.00%
physicalobject.Planet @ 0x8127aec0	64	280	0.00%
physicalobject.Planet @ 0x8127b078	64	280	0.00%
physicalobject.Planet @ 0x8127b588	64	280	0.00%
physicalobject.Planet @ 0xf047a390	64	280	0.00%
physicalobject.Planet @ 0xf047a658	64	280	0.00%
physicalobject.Planet @ 0xf047aa78	64	280	0.00%
physicalobject.Planet @ 0xf047aff8	64	280	0.00%
physicalobject.Planet @ 0xf047b160	64	280	0.00%

Accumulated Objects by Class in Dominator Tree

Label	Number of Objects	Used Heap Size	Retained Heap Size
physicalobject.Planet First 10 of 320,000 objects	320,000	20,480,000	87,571,280
java.util.ArrayList All 3 objects	3	72	36,161,272
java.util.HashMap All 1 objects	1	48	20,017,216
relation.EeIntimacyMap All 1 objects	1	24	176
relation.LeIntimacyMap All 1 objects	1	24	176
centralobject.Stellar All 1 objects	1	32	96
Total: 6 entries	**320,007**	**20,480,200**	**143,750,216**

3.5.3 发现热点/瓶颈并改进、改进前后的性能对比分析

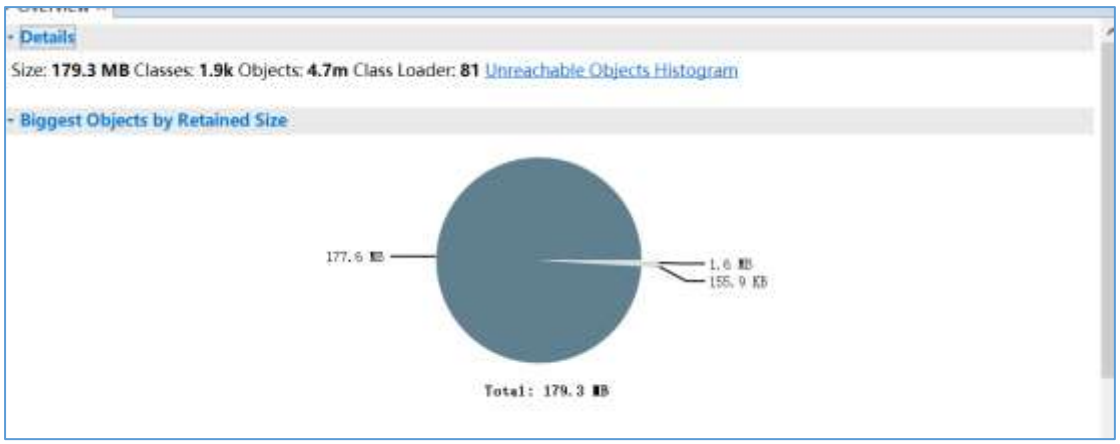
事实上在第二部分文件读写的时候，代码已经做出了很多的优化，这里只是针对执行时间长、内存分配大进行了程序的优化。

1. 在 lab4 异常处理的时候，我将所有出现过的行星名称存入一个 HashSet，以保证不会出现重复的行星名。MAT 同时打开两个堆转储文件，分别打开 Histogram 时，我发现这个 HashSet 中的对象在一直增多，回去看代码发现，我一直在往这个 HashSet 中添加对象，却从没有释放过，于是我在读完文件后：

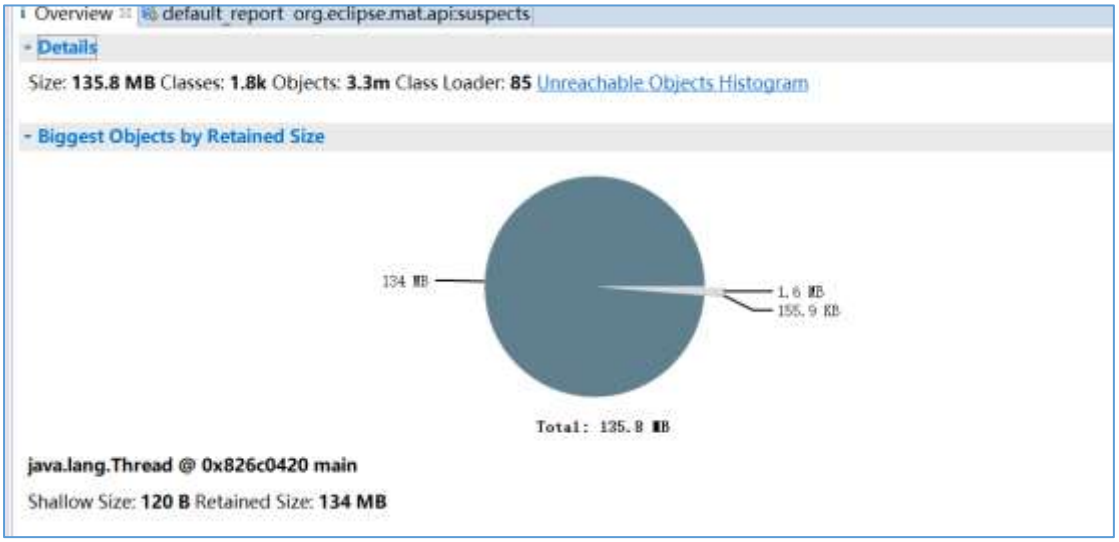
```
labelSet.clear();
System.gc();
```

2. 对于 labelSet 这个集合的对象，是独立于读文件方法外的实例变量，只有在 SteallarSystem 对象被回收，它才会变成不可达的，才会执行 gc。于是我们将其设置为局部变量，在读文件的最后，对其进行垃圾回收。
3. 在我们匹配正则表达式，分割字符串的时候，产生了很多字符数组，它们在方法体内是不会回收的，造成了局部的内存呢占用，于是我们可以在某个字符数组使用完后，将其变为 null，这样它就可以被回收了。
4. 对于 SocialNetworkCircle，其优化的方式和以上完全类似，都是尽量使一个或者是一个集合中的不再使用的对象尽快变成不可达的，从而进行 gc。

改进前的 Retained Size 为 179.3MB

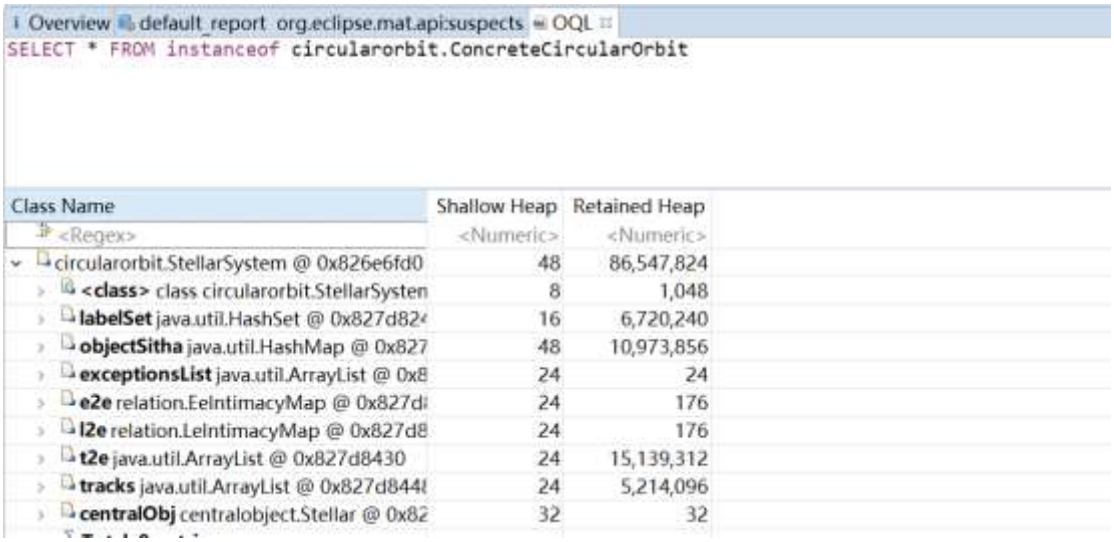


改进后的 Retained Size 为 135.8MB，比优化之前减少了 44MB，已经是个很不错的优化了。



3.5.4 在 MAT 内使用 OQL 查询内存导出

CircularOrbit 的所有对象实例；



大于长度 100 的 String 对象：

Overview OQL

```
SELECT s FROM java.lang.String s WHERE (s.toString().length() >= 100)
```

s
<Regex>
java.lang.String [id=0x8a35a120]
java.lang.String [id=0x81a34b20]
java.lang.String [id=0x819e2ae0]
java.lang.String [id=0x819de0a0]
java.lang.String [id=0x819dcd48]
java.lang.String [id=0x819ccce8]
java.lang.String [id=0x819ccb08]
java.lang.String [id=0x819c4eb0]
java.lang.String [id=0x819c1548]
java.lang.String [id=0x819c1050]
java.lang.String [id=0x819c0c80]
java.lang.String [id=0x81995fa8]
java.lang.String [id=0x819952e8]
java.lang.String [id=0x81991828]

大于特定大小的任意类型对象实例

Overview default_report org.eclipse.mat.api:suspects OQL

```
SELECT * FROM instanceof java.lang.Object o WHERE o.@usedHeapSize>=100
```

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
class sun.nio.cs.StandardCharsets @ 0x82780780 Sys	160	5,216
class java.lang.reflect.Modifier @ 0x8277fd78 Systen	104	176
class sun.net.www.ParseUtil @ 0x82773d60 System (	288	520
class sun.usagetracker.UsageTrackerClient @ 0x8277	168	2,736
class java.util.zip.ZipConstants @ 0x827725c0 Syster	184	200
class jdk.internal.org.objectweb.asm.ClassWriter @ 0	176	1,040
class jdk.internal.org.objectweb.asm.Frame @ 0x827	104	1,736
class sun.security.util.SecurityConstants @ 0x8276ce	136	2,232
class sun.util.calendar.BaseCalendar @ 0x8276c620 S	104	640
class java.net.URI @ 0x8276bc28 System Class	376	648
class com.sun.jmx.defaults.JmxProperties @ 0x82767	120	840
class sun.management.ManagementFactoryHelper @	104	864
class java.math.BigDecimal @ 0x827661e0 System C	144	160
class sun.misc.ProxyGenerator @ 0x8275c5e0 Systen	264	1,576
class sun.misc.FloatingDecimal @ 0x827581d0 Syste	144	1,376
class java.text.DateFormat @ 0x82757100 System Cl	104	144
class sun.util.calendar.ZoneInfoFile @ 0x82756a00 Sy	120	159,608

PhysicalObject（及其子类）的对象实例的数量和总占用内存大小

Overview | default_report org.eclipse.mat.api:suspects | OQL

```
SELECT * FROM instanceof physicalobject.PhysicalObject
```

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
physicalobject.Planet @ 0x8a040fa8	64	224
physicalobject.Planet @ 0x8a040dd0	64	224
physicalobject.Planet @ 0x8a040bf8	64	216
physicalobject.Planet @ 0x8a040a28	64	224
physicalobject.Planet @ 0x8a040850	64	216
physicalobject.Planet @ 0x8a040680	64	216
physicalobject.Planet @ 0x8a0404b0	64	224
physicalobject.Planet @ 0x8a0402d8	64	208
physicalobject.Planet @ 0x8a040110	64	208
physicalobject.Planet @ 0x8a03ff48	64	224
physicalobject.Planet @ 0x8a03fd70	64	216
physicalobject.Planet @ 0x8a03fba0	64	208
physicalobject.Planet @ 0x8a03f9d8	64	216
physicalobject.Planet @ 0x8a03f808	64	216
physicalobject.Planet @ 0x8a03f638	64	216
physicalobject.Planet @ 0x8a03f468	64	224

所有包含元素数量大于 100 的 Collections 实例

Overview | default_report org.eclipse.mat.api:suspects | OQL

```
SELECT * FROM instanceof java.util.ArrayList l WHERE l.size>=100
```

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.util.ArrayList @ 0x827d8448	24	5,214,096
java.util.ArrayList @ 0x827d8430	24	15,139,312
Σ **Total: 2 entries**		

3.5.5 观察 jstack/jcmd 导出程序运行时的调用栈

jstack 用于生成 java 虚拟机当前时刻的线程快照。线程快照是当前 java 虚拟机内每一条线程正在执行的方法堆栈的集合，但是由于调用每个方法的时候速度过快，我们很难在调用和的同时打印出调用栈，因此考虑在调用该方法时，使用 sleep()使程序暂停一段时间继续执

行，方便我们导出程序运行时的调用栈。

```
try{
    Thread.sleep(10000);
    }catch(Exception e){
    System.exit(0);//退出程序
    }
```

增加轨道：

```
"main" #1 prio=5 os_prio=0 tid=0x0000000002c73800 nid=0x4f64 waiting on condition [0x0000000002c6e000]
   java.lang.Thread.State: TIMED_WAITING (sleeping)
        at java.lang.Thread.sleep(Native Method)
        at circularorbit.ConcreteCircularOrbit.addTrack(ConcreteCircularOrbit.java:101)
        at applications.MyOrbitScenes.simulateAtomStructure(MyOrbitScenes.java:200)
        at applications.MyOrbitScenes.myMain(MyOrbitScenes.java:87)
        at applications.MyOrbitScenes.main(MyOrbitScenes.java:63)
```

删除轨道：

```
"main" #1 prio=5 os_prio=0 tid=0x0000000003533800 nid=0x22d4 waiting on condition [0x000000000311f000]
   java.lang.Thread.State: TIMED_WAITING (sleeping)
        at java.lang.Thread.sleep(Native Method)
        at circularorbit.ConcreteCircularOrbit.removeTrack(ConcreteCircularOrbit.java:132)
        at applications.MyOrbitScenes.simulateAtomStructure(MyOrbitScenes.java:228)
        at applications.MyOrbitScenes.myMain(MyOrbitScenes.java:87)
        at applications.MyOrbitScenes.main(MyOrbitScenes.java:63)
```

向特定轨道上增加物体：

```
"main" #1 prio=5 os_prio=0 tid=0x0000000003533800 nid=0x22d4 waiting on condition [0x000000000311f000]
   java.lang.Thread.State: TIMED_WAITING (sleeping)
        at java.lang.Thread.sleep(Native Method)
        at circularorbit.ConcreteCircularOrbit.addTrackObject(ConcreteCircularOrbit.java:160)
        at applications.MyOrbitScenes.simulateAtomStrcture(MyOrbitScenes.java:215)
        at applications.MyOrbitScenes.myMain(MyOrbitScenes.java:87)
        at applications.MyOrbitScenes.main(MyOrbitScenes.java:63)
```

判断轨道的合法性：

```
"main" #1 prio=5 os_prio=0 tid=0x00000000036d3800 nid=0x19a4 waiting on condition [0x00000000036ce000]
   java.lang.Thread.State: TIMED_WAITING (sleeping)
        at java.lang.Thread.sleep(Native Method)
        at legality.LegalStellarSystem.judgeLegality(LegalStellarSystem.java:53)
        at applications.MyOrbitScenes.simulateStellarSystem(MyOrbitScenes.java:407)
        at applications.MyOrbitScenes.myMain(MyOrbitScenes.java:93)
        at applications.MyOrbitScenes.main(MyOrbitScenes.java:64)
```

计算轨道系统的熵值：

```
"main" #1 prio=5 os_prio=0 tid=0x0000000003733800 nid=0x4f10 waiting on condition [0x000000000372f000]
   java.lang.Thread.State: TIMED_WAITING (sleeping)
        at java.lang.Thread.sleep(Native Method)
        at apis.CircularOrbitapis.getObjectDistributionEntropy(CircularOrbitapis.java:49)
        at applications.MyOrbitScenes.simulateStellarSystem(MyOrbitScenes.java:510)
        at applications.MyOrbitScenes.myMain(MyOrbitScenes.java:93)
        at applications.MyOrbitScenes.main(MyOrbitScenes.java:64)
```

3.5.6 使用设计模式进行代码性能优化

1. FlyWeight 设计模式：

因为 AtomicStructure 中 Electron 的对象过多，并且其实它们除了所处的 track，是等同的。因此我们使用 FlyWeight 设计模式设计 Electron 工厂，基本思想就是对于每一个轨道只有一个 Electron 实例。

但是有一处需要注意：因为以前的程序是依据引用（内存地址）判断相同的，在添加物体时的普遍规则是如果轨道系统中有相同的对象则报错，此处需要在子类中覆盖该方法，不进行对象相同检查。

实现 Flyweight 接口的具体享元电子：

```
public class ElectronFlyweight implements Flyweight {
  Track track;

  public ElectronFlyweight(Track t) {
    this.track = t;
  }

}
```

享元电子的工厂：

```
public class ElectronFactory {
  private HashMap<Track, ElectronFlyweight> eletronMap;

  public ElectronFactory() {
    eletronMap = new HashMap<>();
  }

  public ElectronFlyweight getFlyweight(Track t) {
    if (eletronMap.containsKey(t)) {
      return eletronMap.get(t);
    } else {
      ElectronFlyweight flyweight = new ElectronFlyweight(t);
      eletronMap.put(t, flyweight);
      return flyweight;
    }
  }
}
```

使用示例：

```
AtomStructure a = new AtomStructure();
String fileName = "txt/AtomicStructure.txt";
File file = new File(fileName);
ReadStrategy r = new AllBufferedReader();
a.readFile(r.readFile(file));

Track t = a.getTracks().get(0);
ElectronFactory factory = new ElectronFactory();
ElectronFlyweight e = factory.getFlyweight(t);  //获得电子
```

2. Prototype 设计模式：

对于该轨道系统，事实上我认为最适合 Prototype 设计模式的是享元电子的部分，我们可以以一个享元电子作为原型，使用 clone()方法来创建新的实例。

由于电子的 Track 是不同的，被复制的享元电子中引用了 Track 时，需要创建新的 Track(而不是共享)。因此我们需要使用深拷贝。

```
public class DeepCopy implements Cloneable {
  Track track = new Track(1);

  @Override
  public Object clone() throws CloneNotSupportedException {
    DeepCopy t = (DeepCopy) super.clone();
    t.track = new Track(1);
    return t;
  }

}
```

3. Singleton 设计模式

类似于 lab2 中构建一个图之后，getInstance 返回一个图的实例。我们实验中描述轨道物体之间关系、描述中心物体和轨道物体之间关系的图也可以这样返回图的一个单例，通过封装确保对象的复用，不用让客户端考虑。

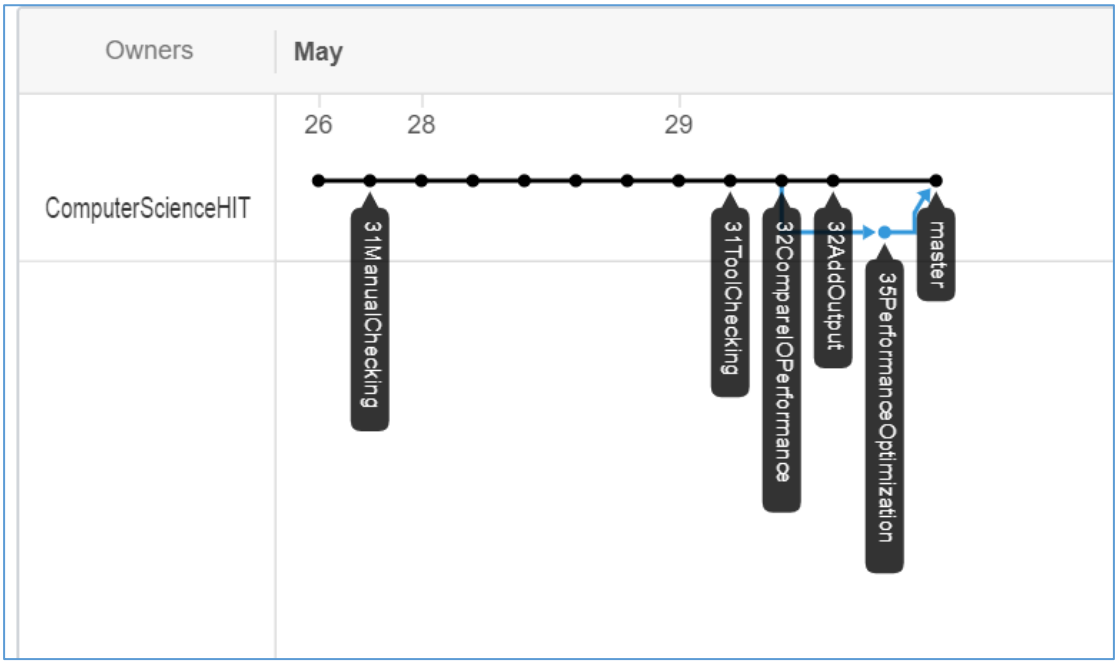
```
private static EeIntimacyMap instance = new EeIntimacyMap();

public static EeIntimacyMap getInstance() {
  return instance;
  }

EeIntimacyMap s = EeIntimacyMap.getInstance();
```

4. **对于 Object Pool 设计模式**，我的程序中并没有很合适的使用之处，但是我们可以产生一个不大的整形数的时候，使用 Integer num = 1；因为 Integer 类中，缓存了从-128 到 127 之间的所有的整数对象，这样也是内存友好的一种方式。

5. **其他**的性能改进：声明字符串变量的时候，使用 "" 而不是用 new String 方法。

3.6 Git 仓库结构

请在完成全部实验要求之后，利用 `Git log` 指令或 `Git` 图形化客户端或 `GitHub` 上项目仓库的 `Insight` 页面，给出你的仓库到目前为止的 `Object Graph`，尤其是区分清楚本实验中要求的多个分支和 `master` 分支所指向的位置。



Excluding merges, **1 author** has pushed **12 commits** to master and **12 commits** to all branches. On master, **200 files** have changed and there have been **2,995,987 additions** and **4,549 deletions**.

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
2019.5.20	14:00-18:00	阅读实验	按计划完成
2019.5.21	18:30-23:00	通读实验要求，配置环境	按计划完成
2019.5.22	15:30-23:00	手动修改代码	比预计晚 2 小时
2019.5.23	13:00-18:00	完成 checkstyle	比预计晚 1 小时
2019.5.23	19:00-23:00	I/O 读文件比较	未完成
2019.5.23	14:00-17:30	继续优化 input	按计划完成
2019.5.25	18:30-22:30	实现 3 种 I/O 策略	按计划完成
2019.5.26	14:00-17:00	分析垃圾回收，命令行工具使用	提前半小时完成
2019.5.26	17:30-23:00	设定 JVM 参数	按计划完成
2019.5.27	13:30-15:30	动态内存分析 VisualVM	未完成
2019.5.27	18:30-23:00	完成前一日内容，使用 MAT	按计划完成
2019.5.28	10:00-17:30	对 MAT 结果分析	比预计晚 2 小时
2019.5.28	18:30-23:00	使用设计模式优化代码	比预计晚 1 小时
2019.5.29	18:30-22:30	继续优化代码	按计划完成
2019.5.29	14:00-17:00	完成报告未写完的部分并提交	提前半小时完成

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
手动代码走查时不清楚代码规范	翻阅 Google 的代码规范
checkstyle 时需要修改的行数过多	修改 preference 即可实现大量代码，尤其是 tab 符的修改
读文件的速度可以统计，但由于建轨道过慢，导致写文件时间难以统计	持续优化搭建轨道的过程

使用-verbose gc 时总出现 Allocation failure	在网上查资料确定时年轻代空间不足，修改参数
命令行工具使用出现部分 error	以管理员身份打开 cmd
对 VisualVm 和 MAT 的使用不熟练，用法不了解	在网上查阅相关工具的使用方式，不断学习
设计模式掌握不熟练	重新翻译上课使用的课件，结合着 java 设计模式的相关书籍，加深印象

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

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？

答：联系和冲突都有的。很多代码看起来不美，但是可能运行起来很高效，但是我们依然最好遵守代码规范，这样方便我们和他人的阅读。个人认为都很重要，“运行起来很美”更重要一些。若编程时间有限，倾向于放在后者。

(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？

答：会。很多隐藏的小错误可能会导致很大的麻烦。

(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？

答：能搞清楚。有很多 I/O 策略可能在我们少量代码的测试中体现不出优势，但是它在其他时候会有更高效的用途。

(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？

答：JVM 的内存管理机制，和之前学的内存管理机制相比，特殊之处在于堆的管理。在 C 语言中，堆这部分空间是唯一一个程序员可以管理的内存区域。程序员可以通过 malloc 函数和 free 函数在堆上申请和释放空间。只不过和 C 语言中的不同，在 Java 中，程序员基本不用去关心空间释放的问题，Java 的垃圾回收机制会自动进行处理。因此这部分空间也是 Java 垃圾收集器管理的主要区域。另外，堆是被所有线程共享的，在 JVM 中只有一个堆。这也是 JVM 内存管理机制的一个新意。我认为这很好，但是仍有需要改进的地方。

(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？

答：在 C++中，堆这部分空间是唯一一个程序员可以管理的内存区域。程序员可以通过 malloc 函数和 free 函数在堆上申请和释放空间。只不过和 C 语言中的不同。

而在 Java 中，程序员基本不用去关心空间释放的问题，Java 的垃圾回收机制会自动进行处理。因此这部分空间也是 Java 垃圾收集器管理的主要区域。另外，堆是被所有线程共享的，在 JVM 中只有一个堆。还有改进的空间。

(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？

答：有很大的回报。内存分配更加合理，内存溢出的概率变小了。

(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？

答：体会到了。尤其是在我看到 gc 日志里的异常情况无法下手时，这些工具给了我很大的帮助。

(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？

答：必须得有耐心，这是普通的码农到高级程序员必经之路。

(9) 关于本实验的工作量、难度、deadline。

答：工作量看似不大，做起来也不轻松，难度不大，deadline 较合适。

(10)到目前为止，你对《软件构造》课程的意见与建议。

答：这是我非常喜欢的一门课，课程教会了我很多的知识，带我走进了软件构造的新世界。但是由于课程进度快，且 ppt 都是英文，因此课程难度较大，整个课程的设计应该还有可以完善的地方。因为课程 ppt 内容过多，建议多考核重点部分，不过多在意细枝末节。这个时候课程已经结束了，希望这门课越来越好！！