



哈爾濱工業大學  
HARBIN INSTITUTE OF TECHNOLOGY

**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	冯郭晟
学号	1170300529
班号	1703005
电子邮件	fgsok@163.com
手机号码	18145193679

## 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	3
3.1.1 人工代码走查 (walkthrough)	3
3.1.2 使用 CheckStyle 和 SpotBugs 进行静态代码分析	5
3.2 Java I/O Optimization	8
3.2.1 多种 I/O 实现方式	8
3.2.2 多种 I/O 实现方式的效率对比分析	11
3.3 Java Memory Management and Garbage Collection (GC)	13
3.3.1 使用-verbose:gc 参数	13
3.3.2 用 jstat 命令行工具的-gc 和-gcutil 参数	15
3.3.3 使用 jmap -heap 命令行工具	16
3.3.4 使用 jmap -clstats 命令行工具	17
3.3.5 使用 jmap -permstat 命令行工具	17
3.3.6 使用 JMC/JFR、jconsole 或 VisualVM 工具	18
3.3.7 分析垃圾回收过程	20
3.3.8 配置 JVM 参数并发现优化的参数配置	21
3.4 Dynamic Program Profiling	22
3.4.1 使用 JMC 或 VisualVM 进行 CPU Profiling	22
3.4.2 使用 VisualVM 进行 Memory profiling	24
3.5 Memory Dump Analysis and Performance Optimization	25
3.5.1 内存导出	25
3.5.2 使用 MAT 分析内存导出文件	25
3.5.3 发现热点/瓶颈并改进、改进前后的性能对比分析	29
3.5.4 在 MAT 内使用 OQL 查询内存导出	29
3.5.5 观察 jstack/jcmd 导出程序运行时的调用栈	31
3.5.6 使用设计模式进行代码性能优化	32
4 实验进度记录	33

---

5 实验过程中遇到的困难与解决途径.....	34
6 实验过程中收获的经验、教训、感想.....	35
6.1 实验过程中收获的经验教训.....	35
6.2 针对以下方面的感受.....	35

## 1 实验目标概述

本次实验通过对 Lab4 的代码进行静态和动态分析，发现代码中存在的不符合代码规范的地方、具有潜在 bug 的地方、性能存在缺陷的地方（执行时间热点、内存消耗大的语句、函数、类），进而使用第 4、7、8 章所学的知识对这些问题加以改进，掌握代码持续优化的方法。

具体训练的技术包括

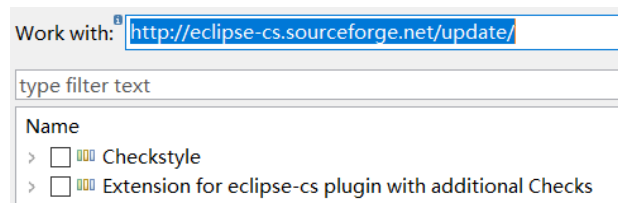
- 静态代码分析（CheckStyle 和 SpotBugs）
- 动态代码分析（Java 命令行工具 jstat、jmap、jcmd、VisualVM、JMC、JConsole 等）
- JVM 内存管理与垃圾回收（GC）的优化配置
- 运行时内存导出(memory dump)及其分析（Java 命令行工具 jhat、MAT）
- 运行时调用栈及其分析（Java 命令行工具 jstack）
- 高性能 I/O
- 基于设计模式的代码调优
- 代码重构

## 2 实验环境配置

- 配置静态代码风格规范检测工具 checkStyle

首先到 eclipse 中选择 help --> download new software

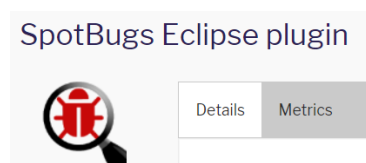
然后输入 checkStyle 的 URL：<http://eclipse-cs.sourceforge.net/update/>



然后按照之前的安装新的插件的步骤点选所有的选项进行安装，然后重启 eclipse 即可

- 安装 spotBugs 工具

直接输入官网的 URL 进入 eclipse 的 market 下载即可



<http://marketplace.eclipse.org/content/spotbugs-eclipse-plugin>

● 安装配置 VisualVM 和 Memory Analyzer 工具

a) 下载安装 VisualVM

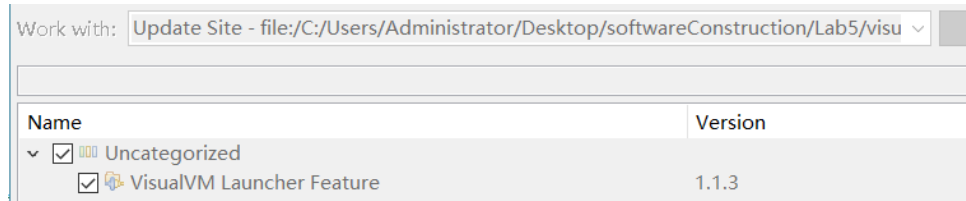
<https://visualvm.github.io/idesupport.html>

到官网的 IDE Integrations 页面，选择 eclipse 接口

Use the **Eclipse Launcher** to integrate VisualVM with the Eclipse IDE. The plugin enables starting VisualVM along with the executed application and automatically opens the application tab.

Installation: download the [plugin \(.zip, 39.1KB\)](#), unzip it and add as a local update site, then install the VisualVM Launcher Feature.

点击 plugin 下载接口到本地，然后在 eclipse 中配置本地接口开始下载



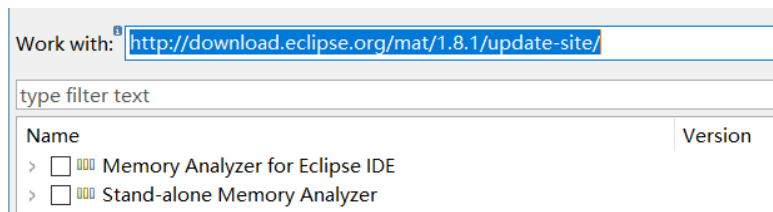
b) 下载安装 Memory Analyzer

到官网选择 download，查看下载 release 页面

Memory Analyzer 1.8.1 Release

- **Version:** 1.8.1.20180910 | **Date:** 19 September 2018 | **Type:** Released
  - **Update Site:** <http://download.eclipse.org/mat/1.8.1/update-site/>
  - **Archived Update Site:** [MemoryAnalyzer-1.8.1.201809100846.zip](#)
  - **Stand-alone Eclipse RCP Applications**
    - Windows (x86)
    - Windows (x86\_64)
    - Mac OSX (Mac/Cocoa/x86\_64)
    - Linux (x86/GTK+)
    - Linux (x86\_64/GTK+)
    - Linux (PPC64/GTK+)
    - Linux (PPC64le/GTK+)

选择 eclipse 的 update site 导入到 eclipse 中进行下载



GitHub Lab5 仓库的 URL 地址

Lab5-1170300529: [git@github.com:ComputerScienceHIT/Lab5-1170300529.git](https://github.com/ComputerScienceHIT/Lab5-1170300529.git)

或者: <https://github.com/ComputerScienceHIT/Lab5-1170300529.git>

## 3 实验过程

### 3.1 Static Program Analysis

#### 3.1.1 人工代码走查 (walkthrough)

选择 Google 的 Java 代码规范，然后通过人工代码走查方式，对源文件中不符合 Google 规范的代码部分，进行修改

注：这里只例举出我需要修改的部分，而其他的例如 4.3 一行一个语句 4.2 块缩进等已经由于编程习惯满足的部分则不一一例举

- package 语句不换行

将源代码中 package 语句换行的部分连接到第一行，保证每个 package 语句都只占用一行

- 3.3.1 import 不要使用通配符

将源代码中使用了通配符的 import 语句都替换成具体的包名

```
import centralObject.CentralObject;
import centralObject.CentralObjectFactory;
import centralObject.CentralUser;
import centralObject.Stellar;
import circularOrbit.AtomStructure;
import circularOrbit.SocialNetworkCircle;
import circularOrbit.StellarSystem;

import exceptions.ElementNameException;
import exceptions.ElementInputException;
import exceptions.NumberRepresentException;
import exceptions.closeNumException;
import exceptions.labelException;
import exceptions.numOfTrackUnmatch;
import exceptions.rotateDirectionInputException;
import exceptions.sameLabelException;
import exceptions.selfRelationException;
import exceptions.tieWithoutDefinitionException;

import physicalObject.Friend;
import physicalObject.PhysicalObject;
import physicalObject.PhysicalObjectFactory;
import physicalObject.Planet;
import track.Track;
```

#### 3.4.2.1 重载：永不分离

当一个类有多个构造函数，或是多个同名方法，这些函数/方法应该按顺序出现在一起，中间

- 不要放进其它函数/方法。

更换源文件中函数的顺序，按逻辑排序各个构造函数和函数

```
//constructors
public concretePhysicalObject(){
    setAngle(0);
}

public concretePhysicalObject(Track track,double angle){
    this.setTrack(track);
    this.setAngle(angle);
}

//-----getter and setters-----
public double getAngle() {
    return angle;
}
public void setAngle(double angle) {
```

大括号与 if, else, for, do, while 语句一起使用，即使只有一条语句(或是空)，也应该把大

- 括号写上。

把所有需要匹配大括号的关键词 if,else,for,do,while 都加上大括号

```
if (this.getObject(electron)==null||this.getTrack(track)==null) {
    return false;
}
```

#### 4.1.2 非空块：K & R 风格

- 对于非空块和块状结构，大括号遵循Kernighan和Ritchie风格
  - 左大括号前不换行
  - 左大括号后换行
  - 右大括号前换行
  - 如果右大括号是一个语句、函数体或类的终止，则右大括号后换行；否则不换行。

```
switch (systemChoice) {
case 1:
    AtomStructure atomStructure;
    try {
        atomStructure = GenerateFromFile
    } catch (ElementNameException e1) {
```

#### ● 4.6.1 垂直空白

以下情况需要使用一个空行：

1. 类内连续的成员之间：字段、构造函数、方法、嵌套类、静态初始化块、实例初始化块。
  - 例外：两个连续字段之间的空行是可选的，用于字段的空行主要用来对字段进行逻辑分组。
2. 在函数体内，语句的逻辑分组间使用空行。
3. 类内的第一个成员前或最后一个成员后的空行是可选的(既不鼓励也不反对这样做，视个人喜好而定)。
4. 要满足本文档中其他节的空行要求(比如3.3节：import语句)

子段逻辑分组，为成员变量按逻辑关系加入空行分组

```
static final Pattern signPattern = Pattern.compile("::=");

static final String LabelRegex = "([a-z]|[A-Z]|\\d)*";
static final String ElementLabelRegex = "[A-Z][a-z]";
static final String genderRegex = "[MF]*";
static final String closeNumRegex = "(0\\.\\d?\\d?\\d?)|1(\\.0*)?";
```

函数语句逻辑分组，为各个语句按功能逻辑关系进行分组

```
buffer = findAfterSign(buffer);
String[] elements = buffer.split(";");
//init
for (int i = 0; i < elements.length; i++) {
    String[] key_values = elements[i].split("/");//split the "d/

    Track track = new Track(Integer.parseInt(key_values[0]));
    track.checkRep();

    for (int j = 0; j < Integer.parseInt(key_values[1]); j++) {
```

#### ● 4.8.2.1 每次只声明一个变量

不要使用组合声明，比如 `int a, b;`。

将代码中有组合声明的部分分开来声明

```
int time1,time2 = 0;    -->    int time1 = 0;
                                int time2 = 0;
```

#### 4.8.4.3 default的情况要写出来

- 每个switch语句都包含一个 `default` 语句组，即使它什么代码也不包含。

为代码中的每一个 switch 语句都加上 default 语句

```
default:
    break;
```

### 5.2.4 常量名

- 常量命名模式为 `CONSTANT_CASE`，全部字母大写，用下划线分隔单词。

将所有的静态的 `final` 的成员变量命名设置为大写

```
public static final int GRAPH_WIDTH = 800;
public static final int GRAPH_HEIGHT = 800;
public static final int FPS = 40;
```

### 7.3 哪里需要使用Javadoc

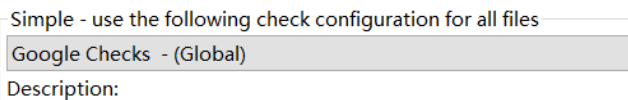
- 至少在每个 `public` 类及它的每个 `public` 和 `protected` 成员处使用 Javadoc，

对某些缺乏 `doc` 注释文档的方法和类加入 Javadoc

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

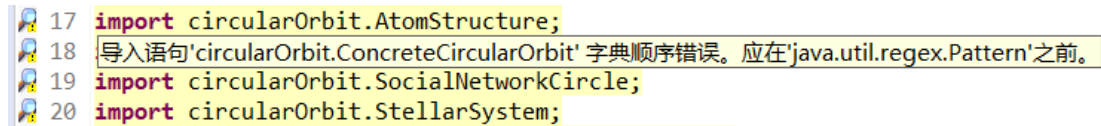
### ➤ 首先使用 check style 工具进行静态代码分析

首先配置 check style 的规约为 Google 规约



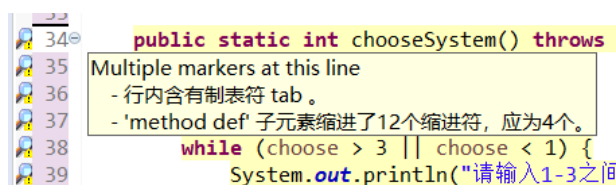
然后运行测试工具，依次找出不符合约定的地方并进行修改

- `import` 语句导入包的顺序不规范



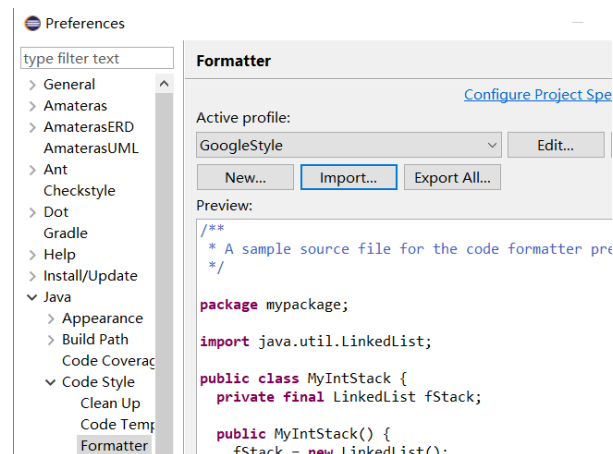
更改原代码中 `import` 语句的顺序，并进行分组

- 代码的空格缩进格式不规范



这里先导入 Java 的 Google formatter 的 xml 文件，然后设置 Google 的 formatter 为默认的格式





然后直接点击格式化 `format` 选项格式化代码即可，可以看到缩进自动调整为所需格式

```
33     scan
34     int
35     scan
36     while
37     System
```

## ● 某些类和方法缺少 Javadoc

```
>/
58 缺少 Javadoc。 ic void chooseAction(ConcreteCircularOrbit orbitSystem)
59      throws IOException, ParseException, rotateDirectionInputException {
60      int choose = scanner.nextInt();
```

依次为每个需要 `doc` 文档的类和方法都补上注释说明

```
* delete a certain Physical Object in this system
*
* @param obj the object waiting to be deleted
* @return true for succeed
*/
public boolean deleteObject(PhysicalObject obj);
```

## ● Javadoc 的格式不规范

主要包括以下几种情况

```
1. 18 Javadoc的第一句缺少一个结束时期。
   19 * delete a certain Physical
```

Javadoc 的第一行内容应该加上 `!!`

```
如所示 /**
        * get the cer
```

```
36 *
37 @标签应有非空说明。
38 * @return the object, null if not find
39 */
40 public PhysicalObject getObject(String label);
```

为每个要求有说明的标签都加上说明

```
* @param track the track to get
* @return track you need
```

### ● switch 块没有 break 导致分支落入下一个 switch 块

```

202         break;
203     }
204
205     从 switch 块的前一支路落入。
206     System.out.println("请输入轨道半径");

```

为每个 switch 块无论需不需要 break 都补上 break 语句，需要落入下一个块的就加上注释

```

    Actions.chooseAction(stellarSystem);
}
break;

case 3:
    SocialNetworkCircle socialNetworkCircle;

```

### ● 部分 Java 类、包、变量和方法命名不规范

```

344     * @param choose : the action that the user chose
345     * @throws IOException may not find file
346     */
347     Method name 'SocialNetworkAction' must match pattern '^([a-z]([a-z0-9]([a-zA-Z0-9])*)?)$' or
348     throws IOException {
349     switch (choose) {

501         File file = new File("src/./log/log.txt");
502     Multiple markers at this line
503     - Local variable name 'RAF' must match pattern '^([a-z]([a-z0-9]([a-zA-Z0-9])*)?)$'
504     - 名称 'RAF' 中不能出现超过 '2' 个连续大写字母。
505     StringBuffer outputLog = new StringBuffer();

```

修改命名的大小写，其中包括

- 1.包名要全部小写
- 2.类名和方法名开头小写
- 3.方法内成员变量命名开头小写

### ● 数组初始化不规范

```

410     String input = scanner.nextLine();
411     数组大括号位置错误。 users[] = input.split(",");
412     while (users.length != 2) {

```

大括号应该放在数组元素类型声明后面

```

String[] users = input.split(",");

```

### ● 行内字符超出限制

```

38     Pattern logTimePattern = Pattern.compile(
39     本行字符数 102个, 最多: 100个。 (?), (\\d\\d\\d\\d) (\\d\\d?):(\\d\\d?):(\\d\\d?) ([\\u4E00-\\u9FA5])午.*?);

```

必要时换行显示，但是要代码的易识别性

```

Pattern logTimePattern = Pattern.compile(
    "(\\d\\d?)月 (\\d\\d?)月 (\\d\\d\\d\\d) (\\d\\d?):(\\d\\d?):(\\d\\d?) ([\\u4E00-\\u9FA5])午.*?"
);

```

### ● 变量声明和使用相距太远，占用空间

```

53     变量'year'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)。
54     String hour = logTimeMatcher.group(4);

```

更改变量声明的时间，尽量在要使用的时候再声明

### ● 大括号的位置书写不规范

右大括号位置不规范

```
138     if (tracks.contains(track)) {  
139         throw new SameLabelException("相同半径的轨道已经有了" + String.valueOf(track.getRadius()));  
140     }  
141     // **Defensive Programming**
```

第 5 个字符 '}' 应该与当前多代码块的下一部分 (if/else-if/else, do/while 或 try/catch/finally) 位于同一行。

修改代码的大括号位置，if-else 语句中右大括号应该和 else 语句在同一行

```
        throw new SameLabelException  
    } else {  
        int index = 0;
```

左大括号位置不规范

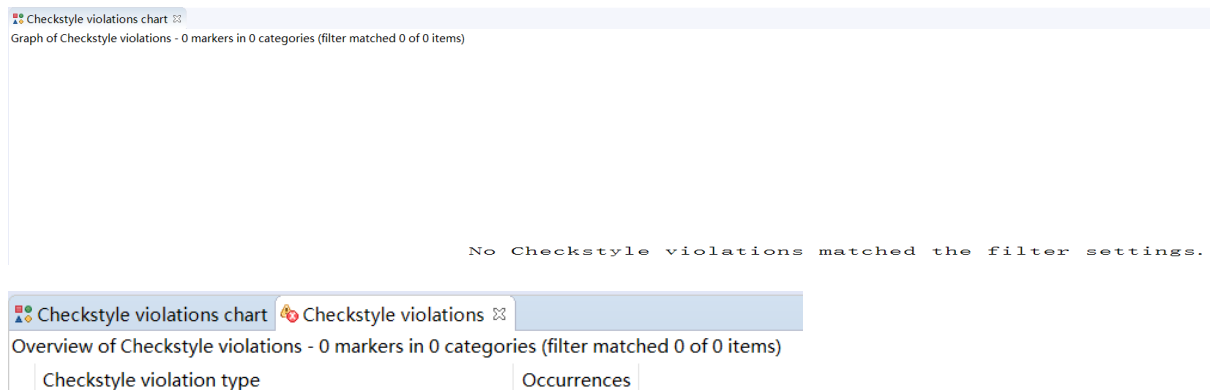
```
212  
213     Iterator<PhysicalObject> deleteIterator = tra  
214     while (deleteIterator.hasNext()) {
```

WhitespaceAround: '{' is not preceded with whitespace.

应该在左大括号前留一个空格字符与后面字符分隔开

```
.isEmpty()) {  
deleteIterator = tra
```

修改完上述类型的格式不规范后，再次运行工具，发现整个项目里面已经没有黄色突出显示的代码，说明代码已经完全符合 Google 规范



## 3.2 Java I/O Optimization

### 3.2.1 多种 I/O 实现方式

分别实现了 bufferedReader/Writer、NIO(mappedChannel)、StreamIO 三种 IO 实现方式，采用了 strategy 设计模式，并书写了专门的 IO 性能测试类来检测 IO 性能。

这里我把文件的写入和写出都简化成了字符串的读入和输出，也就是把文件读入和构造系统分开，用一个专门的 SystemFactory 来利用读入的信息初始化系统，而输出文件也是一样，先用一个 GetSystemInfo 类来构造按照 Lab3 格式需要输出的系统信息，然后利用 strategy

设计模式调用指定的 IO 输出方式写出到文件，使得系统的各个功能分开，符合 separate 的原则

### ● bufferedReader/Writer

#### bufferedReader

首先初始化一个 file reader 实例，然后采用 decorator 的模式初始化一个 bufferedReader

```
BufferedReader bufferedReader = new BufferedReader(new FileReader(file));
```

接着初始化一个 buffer 用于临时存储每一行读入的数据，最后设置一个 stringBuilder 类来连接读入的 buffer

```
String buffer;
StringBuffer stringBuilder = new StringBuffer();
```

设置一个 while 循环，逐行读入文件，存储到 stringBuilder 中，然后用 toString 方法传回读入的信息

```
while ((buffer = bufferedReader.readLine()) != null) {
    stringBuilder.append(buffer + "\n");
}
bufferedReader.close();

return stringBuilder.toString();
```

#### buffered Writer

构造一个 bufferedwriter 类来负责输出字符串

```
BufferedWriter bw = new BufferedWriter(new FileWriter(file));
```

然后把已经构造好的表示系统信息的字符串输出到指定的文件

```
bw.write(writeInfo);
bw.close();
```

### ● NIO(mappedChannel)

NIO 在没有内存映射加持的情况下（如直接使用）是没有普通的 bufferedReader 快的，因此我采用 map 配合 channel 的模式来进行 NIO 的输入输出

#### mappedChannelReader

首先初始化一个 FileChannel 来建立一个和文件的通道

```
FileChannel inchannel =
    FileChannel.open(atomFile.toPath(), StandardOpenOption.READ);
```

设置 OPTION 为只读的，避免因为写发生错误

然后建立一个从 channel 文件到内存的映射

```
MappedByteBuffer inMappedByteBuffer = inchannel.map(MapMode.READ_ONLY, 0, inchannel.size());
```

接着初始化一个大小和文件字节数相同的字节缓冲区

```
byte[] bytes = new byte[inMappedByteBuffer.limit()];
```

最后直接从映射的文件通道读取数据到缓冲区中

```
inMappedByteBuffer.get(bytes); //read
```

然后转化为字符串返回即可

```
String inputString = new String(bytes);  
return inputString;
```

### mappedChannelWriter

与读类似

首先初始化 channel

```
FileChannel outChannel =  
    FileChannel.open(outPath,  
        StandardOpenOption.READ, StandardOpenOption.WRITE, StandardOpenOption.CREATE);
```

然后利用 channel 将文件映射到内存

```
MappedByteBuffer outMappedByteBuffer =  
    outChannel.map(MapMode.READ_WRITE, 0, outputBytes.length);
```

而同时把参数传进的带有系统信息的字符串转化为字节数组

```
byte[] outputBytes = writeInfo.getBytes();
```

最后将数组的字节利用映射的内存输出到文件

```
outMappedByteBuffer.put(outputBytes);
```

## ● StreamIO

### InputStream

StreamReader 利用 fileInputStream 来建立从文件到内存的流

首先初始化一个 FileInputStream

```
InputStream inputStream = new FileInputStream(file);
```

然后初始化一个字节数组，长度为文件长度，用于存储文件信息

```
byte[] buffer = new byte[(int)file.length()];
```

最后直接利用 input Stream 把文件字节读入到缓冲区即可

```
while (inputStream.read(buffer) != -1) {  
}
```

然后转换为字符串返回

```
return new String(buffer);
```

### OutputStream

同样，首先建立一个与输出文件相关联的通道

```
OutputStream outputStream = new FileOutputStream(file);
```

然后把待输出的字符串转换为一个字节数组

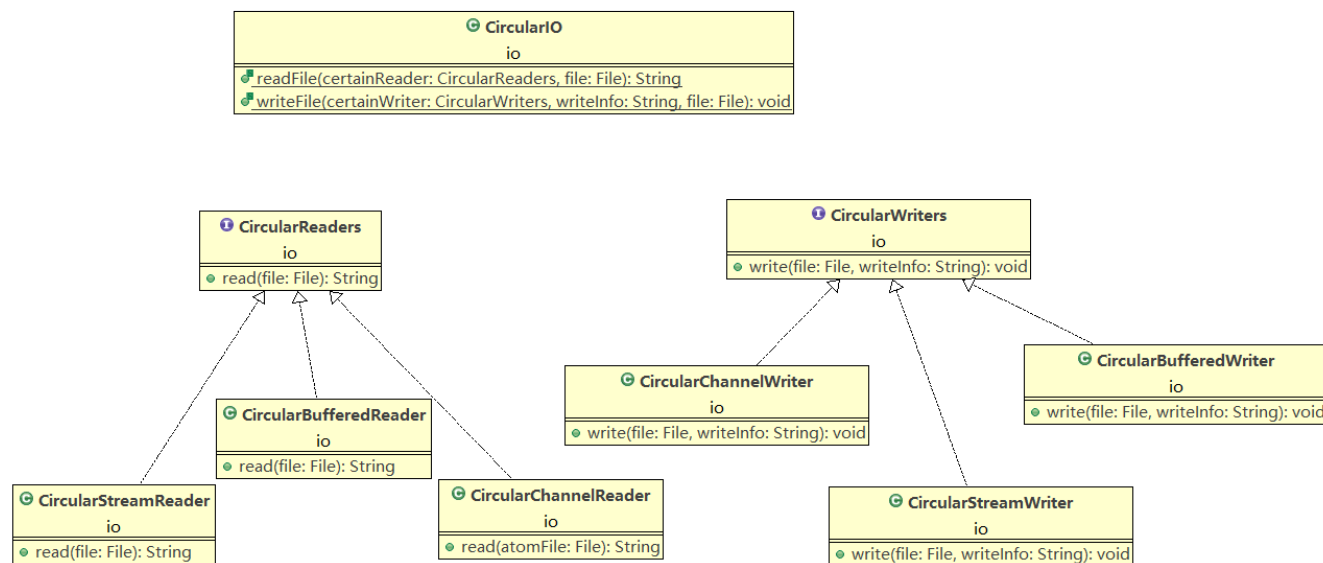
```
byte[] outputByte = writeInfo.getBytes();
```

最后调用 output Stream 的 write 方法输出到文件即可

```
outputStream.write(outputByte);
```

## ● strategy 设计模式的使用

各个 IO 的类图关系如下



首先，在 `circularIO` 类中可以选择是读还是写操作，而通过采用 `strategy` 设计模式，无论是选择读还是写，都必须传入一个指定的读或者写的策略

因此我建立了两个接口，分别负责标准化实现读和写，即无论是哪种策略都必须实现读和写的一般功能

而实现了接口的各个策略负责具体实现读和写的操作

这样，用户只需要知道 `CircularIO` 的方法和具体的读写策略就可以灵活地调用各种策略进行读写了

而基于独立的原则，我把读入的信息转换为系统和解析系统信息的操作由两个具体的类负责实现

`SystemFactory` 负责利用读入的信息字符串构造一个具体的系统

`GetSystemInfo` 负责将指定系统的信息按照 Lab3 的格式构造成一个字符串

比如，当用户需要读入一个原子系统时，只需这样调用方法即可

```
atomStructure = SystemFactory.generateAtomStructure(
    CircularIO.readFile(new CircularChannelReader(), atomFile));
```

### 3.2.2 多种 I/O 实现方式的效率对比分析

为了更好地测试 IO 实现的效率，我专门写了一个 `IOSpeed` 测试类

而通过 `System.currentTimeMillis()` 方法来分别获取读写操作前后的系统时间，通过插入代码到 IO 操作的前后即可得到准确的 IO 时间

比如，测试 `channelIO` 的性能

首先初始化 channel，这个时间不算在读取和写入的时间里面

```
FileChannel inchannel =  
    FileChannel.open(inPath, StandardOpenOption.READ);  
FileChannel outChannel =  
    FileChannel.open(outPath,  
        StandardOpenOption.READ, StandardOpenOption.WRITE, StandardOpenOption.CREATE);
```

然后建立映射

```
MappedByteBuffer inMappedByteBuffer = inchannel.map(MapMode.READ_ONLY, 0, inchannel.size());  
MappedByteBuffer outMappedByteBuffer = outChannel.map(MapMode.READ_WRITE, 0, inchannel.size());
```

最后插入代码到读取写入操作的前后用于获取时间

```
//read  
long readStartTime = System.currentTimeMillis();  
inMappedByteBuffer.get(bytes);  
String inputString = new String(bytes);  
//write  
long writeStartTime = System.currentTimeMillis();  
outMappedByteBuffer.put(bytes);  
long writeEndTime = System.currentTimeMillis();
```

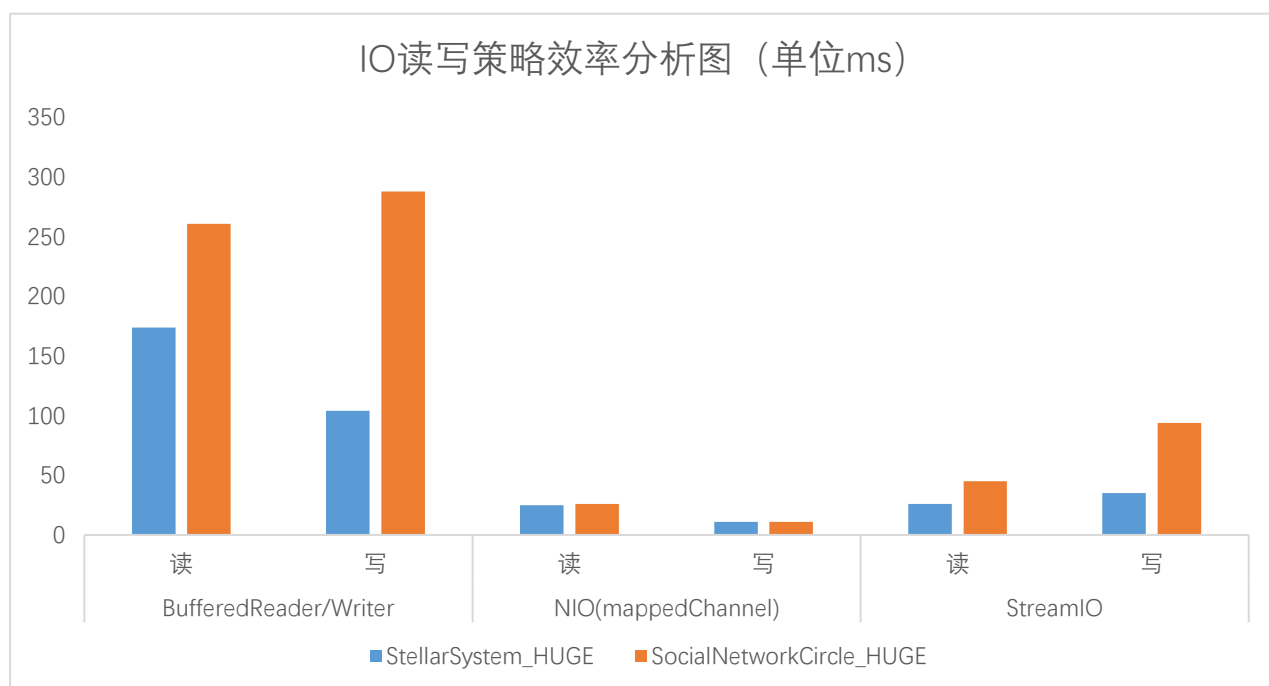
最后用 endTime 减去 beginTime 即可得到 IO 的时间

```
System.out.println("channelMappedNIOReader读HUGEstellar运行时间:"  
    + (readEndTime - readStartTime) + "ms");  
System.out.println("channelMappedNIOReader写HUGEstellar运行时间:"  
    + (writeEndTime - writeStartTime) + "ms");
```

这里采用读取给出的超大型文件来得到时间，分别测试 stellarSystem 和 socialNetWorkCircle 得到下面的测试结果

		StellarSystem_HUGE	SocialNetworkCircle_HUGE
Buffered Reader/Writer	读	174ms(0.174s)	261ms(0.261s)
	写	104ms(0.104s)	288ms(0.288s)
NIO (mappedChannel)	读	25ms(0.025s)	26ms(0.026s)
	写	11ms(0.011s)	11ms(0.011s)
StreamIO	读	26ms(0.026s)	45ms(0.045s)
	写	35ms(0.035s)	94ms(0.094s)





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

#### 3.3.1 使用-verbose:gc 参数

首先在 ini 文件中加入-verbose:gc 和-Xloggc 参数来观察 GC 的情况

```
-XX:+PrintGCTimeStamps
-XX:+PrintGCDetails
-verbose:gc
-Xloggc:C:\Users\Administrator\Desktop\
```

由于我的程序在做代码检查的时候进行了优化，垃圾处理现在基本看不到 fullGC，因此我采用上一个分支提交的程序来检测垃圾回收情况

看到输出的结果如下

```
[6.581s][info][gc] GC(7) Pause Young (Normal) (G1 Evacuation Pause) 38M->35M(256M) 13.850ms
[10.291s][info][gc] GC(8) Pause Young (Normal) (G1 Evacuation Pause) 87M->42M(256M) 12.160ms
[13.055s][info][gc] GC(9) Pause Young (Normal) (G1 Evacuation Pause) 110M->49M(256M) 14.482ms
[13.327s][info][gc] GC(10) Pause Young (Concurrent Start) (Metadata GC Threshold) 58M->52M(256M) 18.616ms
[13.328s][info][gc] GC(11) Concurrent Cycle
[13.483s][info][gc] GC(11) Pause Remark 56M->56M(256M) 13.613ms
[13.546s][info][gc] GC(11) Pause Cleanup 58M->58M(256M) 0.086ms
[13.546s][info][gc] GC(11) Concurrent Cycle 218.725ms
[14.668s][info][gc] GC(12) Pause Young (Concurrent Start) (G1 Evacuation Pause) 145M->51M(256M) 22.872ms
[14.668s][info][gc] GC(13) Concurrent Cycle
[14.810s][info][gc] GC(13) Pause Remark 57M->57M(256M) 18.644ms
[14.877s][info][gc] GC(13) Pause Cleanup 59M->59M(256M) 0.079ms
[14.878s][info][gc] GC(13) Concurrent Cycle 209.815ms
[16.341s][info][gc] GC(14) Pause Young (Normal) (G1 Evacuation Pause) 154M->58M(256M) 10.957ms
[16.949s][info][gc] GC(15) Pause Young (Normal) (G1 Evacuation Pause) 162M->65M(256M) 24.396ms
[19.494s][info][gc] GC(16) Pause Young (Normal) (G1 Evacuation Pause) 172M->73M(256M) 26.871ms
[22.064s][info][gc] GC(17) Pause Young (Normal) (G1 Evacuation Pause) 180M->84M(256M) 31.625ms
[24.490s][info][gc] GC(18) Pause Young (Normal) (G1 Evacuation Pause) 186M->92M(256M) 43.564ms
[28.377s][info][gc] GC(19) Pause Young (Normal) (G1 Evacuation Pause) 201M->100M(256M) 26.287ms
[29.690s][info][gc] GC(20) Pause Young (Normal) (G1 Evacuation Pause) 200M->107M(256M) 32.943ms
[34.802s][info][gc] GC(21) Pause Young (Concurrent Start) (G1 Humongous Allocation) 204M->130M(256M) 21.358ms
[34.802s][info][gc] GC(22) Concurrent Cycle
[34.837s][info][gc] GC(22) Pause Remark 137M->134M(256M) 19.430ms
[34.863s][info][gc] GC(22) Pause Cleanup 155M->155M(256M) 0.127ms
[34.864s][info][gc] GC(22) Concurrent Cycle 442.956ms
[34.935s][info][gc] GC(23) Pause Young (Prepare Mixed) (G1 Evacuation Pause) 255M->122M(263M) 20.856ms
[34.940s][info][gc] GC(24) Pause Young (Mixed) (G1 Evacuation Pause) 128M->113M(263M) 13.380ms
[36.456s][info][gc] GC(25) Pause Young (Normal) (G1 Evacuation Pause) 197M->116M(263M) 9.103ms
[36.483s][info][gc] GC(26) Pause Young (Concurrent Start) (G1 Humongous Allocation) 134M->122M(263M) 10.475ms
[36.483s][info][gc] GC(27) Concurrent Cycle
[36.506s][info][gc] GC(27) Pause Remark 128M->116M(263M) 19.861ms
```



由于我使用的是 G1 垃圾回收器，主要分 Eden, Survivor, Old, HUmongous 四个区域

观察上面的垃圾回收情况可以看到：

normal 代表的 minorGC 的发生次数比较多

```
[16.341s][info][gc] GC(14) Pause Young (Normal) (G1 Evacuation Pause) 154M->58M(256M) 10.957ms
[16.949s][info][gc] GC(15) Pause Young (Normal) (G1 Evacuation Pause) 162M->65M(256M) 24.396ms
[19.494s][info][gc] GC(16) Pause Young (Normal) (G1 Evacuation Pause) 172M->73M(256M) 26.871ms
[22.064s][info][gc] GC(17) Pause Young (Normal) (G1 Evacuation Pause) 180M->84M(256M) 31.625ms
[24.490s][info][gc] GC(18) Pause Young (Normal) (G1 Evacuation Pause) 186M->92M(256M) 43.564ms
[28.377s][info][gc] GC(19) Pause Young (Normal) (G1 Evacuation Pause) 201M->100M(256M) 26.287ms
[29.690s][info][gc] GC(20) Pause Young (Normal) (G1 Evacuation Pause) 200M->107M(256M) 32.943ms
```

这些 minorGC 的平均发生的持续时间在 29ms 左右，而每次大概会回收 100M 左右的内存空间

而 MajorGC 发生的频率要远小于 minorGC 的频率，下面是发生一次 majorGC 循环的情况

```
[13.328s][info][gc] GC(11) Concurrent Cycle
[13.483s][info][gc] GC(11) Pause Remark 56M->56M(256M) 13.613ms
[13.546s][info][gc] GC(11) Pause Cleanup 58M->58M(256M) 0.086ms
[13.546s][info][gc] GC(11) Concurrent Cycle 218.725ms
```

可以看到对 oldGeneration 的垃圾回收使用的是标记算法，执行一次回收的时间消耗是 218.725ms，这无疑是一个较大的时间消耗，消耗的时间是 minorGC 的 10 倍左右

同时 G1 中还有 HumongousGC

```
[348.021s][info][gc] GC(21) Pause Young (Concurrent Start) (G1 Humongous Allocation) 204M->130M(256M) 21.358ms
```

如果一个对象占用的空间超过了分区容量 50% 以上，G1 收集器就认为这是一个巨型对象。这些巨型对象，默认直接会被分配在年老代

HumongousGC 每次回收的占用时间大致为 20ms，与 MinorGC 相当

### heap 内存占用分析：

加入 -XX: +PrintGCDetails 参数后，初始化程序可以看到输出

可以看到输出如下

```
[20.337s][info] [gc,marking] GC(12) Concurrent Cleanup for Next Mark
[20.337s][info] [gc,marking] GC(12) Concurrent Cleanup for Next Mark 0.113ms
[20.337s][info] [gc] GC(12) Concurrent Cycle 161.886ms
[20.345s][info] [gc,start] GC(17) Pause Young (Normal) (G1 Evacuation Pause)
[20.345s][info] [gc,task] GC(17) Using 3 workers of 4 for evacuation
[20.345s][info] [gc,phases] GC(17) Pre Evacuate Collection Set: 0.0ms
[20.355s][info] [gc,phases] GC(17) Evacuate Collection Set: 6.5ms
[20.355s][info] [gc,phases] GC(17) Post Evacuate Collection Set: 3.3ms
[20.355s][info] [gc,phases] GC(17) Other: 0.1ms
[20.355s][info] [gc,heap] GC(17) Eden regions: 16->0(70)
[20.355s][info] [gc,heap] GC(17) Survivor regions: 3->3(3)
[20.355s][info] [gc,heap] GC(17) Old regions: 65->68
[20.355s][info] [gc,heap] GC(17) Humongous regions: 22->22
[20.355s][info] [gc,metaspace] GC(17) Metaspace: 9394K->9394K(1058816K)
[20.355s][info] [gc] GC(17) Pause Young (Normal) (G1 Evacuation Pause) 105M->91M(260M) 10.045ms
[20.355s][info] [gc,cpu] GC(17) User=0.05s Sys=0.00s Real=0.01s
[20.465s][info] [gc,start] GC(18) Pause Young (Normal) (G1 Evacuation Pause)
[20.465s][info] [gc,task] GC(18) Using 4 workers of 4 for evacuation
[20.481s][info] [gc,phases] GC(18) Pre Evacuate Collection Set: 0.0ms
[20.481s][info] [gc,phases] GC(18) Evacuate Collection Set: 15.1ms
[20.481s][info] [gc,phases] GC(18) Post Evacuate Collection Set: 0.2ms
[20.481s][info] [gc,phases] GC(18) Other: 0.3ms
[20.481s][info] [gc,heap] GC(18) Eden regions: 70->0(86)
[20.481s][info] [gc,heap] GC(18) Survivor regions: 3->10(10)
[20.481s][info] [gc,heap] GC(18) Old regions: 68->69
[20.481s][info] [gc,heap] GC(18) Humongous regions: 22->22
[20.481s][info] [gc,metaspace] GC(18) Metaspace: 9394K->9394K(1058816K)
[20.481s][info] [gc] GC(18) Pause Young (Normal) (G1 Evacuation Pause) 161M->101M(260M) 15.773ms
[20.481s][info] [gc,cpu] GC(18) User=0.06s Sys=0.00s Real=0.02s
[20.594s][info] [gc,start] GC(19) Pause Young (Normal) (G1 Evacuation Pause)
[20.594s][info] [gc,task] GC(19) Using 4 workers of 4 for evacuation
[20.618s][info] [gc,phases] GC(19) Pre Evacuate Collection Set: 0.0ms
[20.618s][info] [gc,phases] GC(19) Evacuate Collection Set: 23.2ms
[20.618s][info] [gc,phases] GC(19) Post Evacuate Collection Set: 0.2ms
[20.618s][info] [gc,phases] GC(19) Other: 0.1ms
```

其中,每次进行了 minorGC 后,Eden 区域中的元素都被清理,存活下来的被分配到了 survivor 区域和 old 区域

```
[gc,heap] GC(19) Eden regions: 86->0(78)
[gc,heap] GC(19) Survivor regions: 10->12(12)
[gc,heap] GC(19) Old regions: 69->79
[gc,heap] GC(19) Humongous regions: 25->25|
```

而当 survivor 区域满的时候就会直接分配到 old 区域

```
[gc,heap] GC(16) Eden regions: 18->0(16)
[gc,heap] GC(16) Survivor regions: 3->3(3)
[gc,heap] GC(16) Old regions: 62->65
[gc,heap] GC(16) Humongous regions: 22->22
```

由此可见,程序主要产生的是 eden 部分的垃圾,可能是读入的数据在实例化的时候生成的一些临时的实例对象,用过一次以后便不再使用,这些对象在 Eden 区域中大量产生,也被大量的回收

### 3.3.2 用 jstat 命令行工具的-gc 和-gcutil 参数

这里我使用的是优化后的程序进行读入来验证 gc 的工作是否正常

输入参数如下

```
C:\Users\Administrator>jstat -gc -h3 6292 250 10
```

然后读入 file1.txt 大文件,查看到 GC 的情况如下

S0C	S1C	S0U	S1U	EC	EU	OC	OU	MC	MU	CCSC	CCSU	YGC	YGCT	FGC	FGCT	CGC	CGCT	GCT
0.0	0.0	0.0	0.0	15360.0	5120.0	113664.0	18432.0	9728.0	9170.5	1024.0	841.0	0	0.000	0	0.000	0	0.000	0.000
0.0	3072.0	0.0	0.0	3072.0	18432.0	4096.0	107520.0	61993.6	9984.0	9416.3	1024.0	852.6	4	0.043	0	0.000	0	0.000
0.0	3072.0	0.0	0.0	3072.0	23552.0	7168.0	102400.0	76390.4	9984.0	9416.6	1024.0	852.6	10	0.086	0	0.000	4	0.002
S0C	S1C	S0U	S1U	EC	EU	OC	OU	MC	MU	CCSC	CCSU	YGC	YGCT	FGC	FGCT	CGC	CGCT	GCT
0.0	2048.0	0.0	0.0	2048.0	61440.0	46080.0	194560.0	88843.0	9984.0	9420.9	1024.0	852.6	15	0.124	0	0.000	6	0.004
0.0	10240.0	0.0	0.0	10240.0	74752.0	6144.0	173056.0	112436.3	9984.0	9420.9	1024.0	852.6	18	0.177	0	0.000	6	0.004
0.0	10240.0	0.0	0.0	10240.0	160768.0	97280.0	603136.0	122380.0	9984.0	9420.9	1024.0	852.6	19	0.208	0	0.000	6	0.004
S0C	S1C	S0U	S1U	EC	EU	OC	OU	MC	MU	CCSC	CCSU	YGC	YGCT	FGC	FGCT	CGC	CGCT	GCT
0.0	20480.0	0.0	0.0	20480.0	296960.0	155648.0	456704.0	135048.5	9984.0	9420.9	1024.0	852.6	20	0.241	0	0.000	6	0.004
0.0	36864.0	0.0	0.0	36864.0	316416.0	44032.0	420864.0	157090.0	9984.0	9420.9	1024.0	852.6	21	0.303	0	0.000	6	0.004
0.0	36864.0	0.0	0.0	36864.0	316416.0	195584.0	420864.0	157090.0	9984.0	9420.9	1024.0	852.6	21	0.303	0	0.000	6	0.004
S0C	S1C	S0U	S1U	EC	EU	OC	OU	MC	MU	CCSC	CCSU	YGC	YGCT	FGC	FGCT	CGC	CGCT	GCT
0.0	36864.0	0.0	0.0	36864.0	316416.0	195584.0	420864.0	157090.0	9984.0	9420.9	1024.0	852.6	21	0.303	0	0.000	6	0.004
0.0	36864.0	0.0	0.0	36864.0	316416.0	195584.0	420864.0	157090.0	9984.0	9420.9	1024.0	852.6	21	0.303	0	0.000	6	0.004
0.0	36864.0	0.0	0.0	36864.0	316416.0	195584.0	420864.0	157090.0	9984.0	9420.9	1024.0	852.6	21	0.303	0	0.000	6	0.004

可以看到,在 G1 中第一个幸存者区域 S1 和 Eden 区域的最大容量 S1C 和 EC 随时间和程序运行动态改变,而经过一次垃圾回收后,Eden 区域的使用量 EU 降低

S1C	S0U	S1U	EC	EU
2048.0	0.0	2048.0	61440.0	46080.0
10240.0	0.0	10240.0	74752.0	6144.0
10240.0	0.0	10240.0	160768.0	97280.0
S1C	S0U	S1U	EC	EU
20480.0	0.0	20480.0	296960.0	155648.0

old generation 的容量也随时间改变,而 Eden 中未被回收的一部分内容进入到老年代中

OC	OU
113664.0	18432.0
107520.0	61993.6
102400.0	76390.4

永久代中的方法区和压缩类的总容量和使用量都随程序运行增加,说明随着文件的读入,初始化的对象存储到对应的 orbit system 中,而由于还未被删除而一直存在,随着读入的增加,永久代使用量也增加

MC	MU	CCSC	CCSU
9728.0	9170.5	1024.0	841.0
9984.0	9416.3	1024.0	852.6
9984.0	9416.6	1024.0	852.6

后面显示的是 gc 的次数和时间，观察发现，读入过程中，程序并没有发生 full gc 说明程序整体运行良好，gc 的总时间 GCT 实际是 young gc 的总时间 YGCT

YGC	YGCT	FGC	FGCT	CGC	CGCT	GCT
15	0.124	0	0.000	6	0.004	0.128
18	0.177	0	0.000	6	0.004	0.181
19	0.208	0	0.000	6	0.004	0.213

而输入参数

```
C:\Users\Administrator>jstat -gcutil -h3 11376 250
```

得到的 heap 运行时的各个区域的占用比例的情况如下

S1	E	O	M	CCS	YGC	YGCT	FGC	FGCT	CGC	CGCT	GCT
0.00	26.67	0.00	-	-	0	0.000	0	0.000	0	0.000	0.000
0.00	33.33	16.22	94.04	82.13	0	0.000	0	0.000	0	0.000	0.000
100.00	33.33	57.66	94.09	83.26	4	0.039	0	0.000	0	0.000	0.039

和上面的分析一致，当程序实际运行读入行星的时候程序才有垃圾回收，从而为 S1 分配空间使得 Eden 的内容进入到 S1 中

### 3.3.3 使用 jmap -heap 命令行工具

直接使用 jmap -heap 命令发现这条命令已经失效了

区 Oracle 官网可以看到，对于 core 中运行的程序，应该使用 jhsdb 配合 jmap 来查看 heap 的情况

```
C:\Users\Administrator>jmap -heap 14680
Error: -heap option used
Cannot connect to core dump or remote debug server. Use jhsdb jmap instead
```

改正输入命令如下

```
C:\Users\Administrator>jhsdb jmap --heap --pid 14680
Attaching to process ID 14680, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 11.0.2+9-LTS
```

可以得到结果如下

```
Heap Configuration:
  MinHeapFreeRatio      = 40
  MaxHeapFreeRatio      = 70
  MaxHeapSize           = 2101346304 (2004.0MB)
  NewSize               = 1363144 (1.2999954223632812MB)
  MaxNewSize            = 1260388352 (1202.0MB)
  OldSize               = 5452592 (5.1999969482421875MB)
  NewRatio              = 2
  SurvivorRatio         = 8
  MetaspaceSize         = 21807104 (20.796875MB)
  CompressedClassSpaceSize = 1073741824 (1024.0MB)
  MaxMetaspaceSize      = 17592186044415 MB
  GLHeapRegionSize      = 1048576 (1.0MB)

Heap Usage:
  GL Heap:
    regions = 2004
    capacity = 2101346304 (2004.0MB)
    used = 162529280 (155.0MB)
    free = 1938817024 (1849.0MB)
    7.734530938123752% used
  GL Young Generation:
    Eden Space:
      regions = 15
      capacity = 213909504 (204.0MB)
      used = 15728640 (15.0MB)
      free = 198180864 (189.0MB)
      7.352941176470588% used
    Survivor Space:
      regions = 11
      capacity = 11534336 (11.0MB)
      used = 11534336 (11.0MB)
      free = 0 (0.0MB)
      100.0% used
  GL Old Generation:
    regions = 129
    capacity = 611319808 (583.0MB)
    used = 135266304 (129.0MB)
    free = 476053504 (454.0MB)
    22.126929674099486% used
```

其中首先显示的是垃圾回收的策略

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

可以看到回收策略采用 G1，为 4 线程的多线程模式，和电脑的 CPU 内核数相同  
然后显示的是 heap 分配的基本配置情况，包括最大最小的 free 比例，最大空间，初始化的空间，幸存者比例，metaspace 的大小等等

```
MinHeapFreeRatio      = 40
MaxHeapFreeRatio      = 70
MaxHeapSize           = 2101346304 (2004.0MB)
NewSize               = 1363144 (1.2999954223632812MB)
MaxNewSize            = 1260388352 (1202.0MB)
OldSize               = 5452592 (5.1999969482421875MB)
NewRatio              = 2
SurvivorRatio         = 8
MetaspaceSize         = 21807104 (20.796875MB)
CompressedClassSpaceSize = 1073741824 (1024.0MB)
MaxMetaspaceSize     = 17592186044415 MB
G1HeapRegionSize      = 1048576 (1.0MB)
```

然后是各个 heap 的使用情况，例如 Eden 当前的使用情况为

```
G1 Young Generation:
Eden Space:
  regions = 15
  capacity = 213909504 (204.0MB)
  used = 15728640 (15.0MB)
  free = 198180864 (189.0MB)
  7.352941176470588% used
```

总容量 204MB，使用了 15MB，剩余 189MB，使用比例为 7.35%

### 3.3.4 使用 jmap -clstats 命令行工具

输入指令 `C:\Users\Administrator>jmap -clstats 5612`

可以看到程序执行期间装载的 class 和 method 相关信息如下

Index	Super	InstBytes	ClassBytes	annotations	CpAll	MethodCount	Bytecodes	MethodAll	ROAll	RWAll	Total	ClassName
1	-1	729152	504	0	0	0	0	0	24	616	640	[B
2	-1	169920	504	0	0	0	0	0	24	616	640	[C
3	46	165656	672	0	22112	139	5682	37344	24616	37408	62024	java.lang.Class
4	46	157248	616	128	14216	109	4577	40688	18640	38312	56952	java.lang.String
5	46	119296	584	0	1384	7	149	1864	1152	3000	4152	java.util.HashMap\$Node
6	-1	114368	504	0	0	0	0	0	24	616	640	[Ljava.lang.Object;
7	46	91456	592	0	1360	9	213	2360	1488	3160	4648	java.util.concurrent.ConcurrentHashMap\$Node
8	-1	46544	504	0	0	0	0	0	32	616	648	[Ljava.util.HashMap\$Node;
9	-1	41576	504	0	0	0	0	0	24	616	640	[I
10	-1	28064	504	0	0	0	0	0	32	616	648	[Ljava.util.concurrent.ConcurrentHashMap\$Node;
11	46	20832	576	0	11520	89	4308	38168	15088	36536	51624	java.lang.invoke.MemberName
12	46	18496	528	120	5496	37	1783	13400	6552	13552	20104	java.lang.invoke.LambdaForm\$Name
13	939	16320	1024	0	7904	51	4065	24000	12672	21240	33912	java.util.HashMap
14	-1	13784	504	0	0	0	0	0	48	616	664	[Ljava.lang.String;
15	-1	13120	504	0	0	0	0	0	56	616	672	[Ljava.lang.Class;
16	-1	12376	504	0	0	0	0	0	24	616	640	[Ljava.lang.ref.SoftReference;
17	236	12160	560	0	848	4	83	2616	648	3584	4232	java.lang.invoke.MethodType\$ConcurrentWeakInternSet\$WeakEntry
18	46	11280	528	296	10224	72	2948	27736	12312	27528	39840	java.lang.invoke.MethodType
19	46	8496	552	0	2168	11	337	3360	1816	4600	6416	java.lang.module.ModuleDescriptor\$Exports
20	46	7176	528	0	280	1	5	168	152	984	1136	java.lang.invoke.ResolvedMethodName
21	-1	6816	504	0	0	0	0	0	24	616	640	[Z
22	980	6816	1040	88	1576	8	212	3200	1384	4880	6264	java.util.ImmutableCollections\$Set12

打印 Java 堆内存的永久保存区域的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。

### 3.3.5 使用 jmap -histo 命令行工具

(注，这里可能是报告小标题打错了，因为 -permstat 命令是 jdk8 以前的 -clstats 命令的替代形式，而指导书上面的 histo 命令行工具还没有使用，因此这里改成 histo 命令行工具的使用)

输入命令

```
C:\Users\Administrator>jmap -histo 12248
```

得到的打印输出如下

num	#instances	#bytes	class name (module)
1:	1432783	71187168	[B (java.base@11.0.2)
2:	1431333	34363992	java.lang.String (java.base@11.0.2)
3:	320824	28792304	Ljava.util.HashMap\$Node; (java.base@11.0.2)
4:	232955	26839512	[I (java.base@11.0.2)
5:	320000	25600000	physicalobject.Planet
6:	643728	20599296	java.util.HashMap\$Node (java.base@11.0.2)
7:	320341	15376368	java.util.HashMap (java.base@11.0.2)
8:	320000	7680000	track.Track
9:	319163	7659912	java.util.regex.Pattern\$BmpCharProperty (java.base@11.0.2)
10:	232102	7427264	java.util.regex.Pattern\$Curly (java.base@11.0.2)
11:	232141	5571384	java.util.regex.Pattern\$GroupHead (java.base@11.0.2)
12:	232141	5571384	java.util.regex.Pattern\$GroupTail (java.base@11.0.2)
13:	232121	5570904	java.util.regex.Pattern\$CharProperty (java.base@11.0.2)
14:	320198	5123168	java.util.HashSet (java.base@11.0.2)
15:	30893	4780744	Ljava.lang.Object; (java.base@11.0.2)
16:	261121	4177936	java.util.regex.Pattern\$\$Lambda\$17/0x00000001000d0040 (java.base@11.0.2)
17:	58021	4177512	java.util.regex.Matcher (java.base@11.0.2)
18:	115946	3710272	jdk.internal.math.FloatingDecimal\$ASCIIToBinaryBuffer (java.base@11.0.2)
19:	116389	3418768	[C (java.base@11.0.2)
20:	29020	2553760	java.util.regex.Pattern (java.base@11.0.2)
21:	29011	1624616	Ljava.util.regex.Pattern\$GroupHead; (java.base@11.0.2)
22:	58030	1392720	java.util.regex.Pattern\$BmpCharPropertyGreedy (java.base@11.0.2)
23:	58028	1392672	java.util.regex.Pattern\$Slice (java.base@11.0.2)
24:	180	1293960	Ljava.lang.String; (java.base@11.0.2)
25:	58023	928368	Ljava.util.regex.IntHashSet; (java.base@11.0.2)
26:	29040	696960	java.util.ArrayList (java.base@11.0.2)
27:	29014	696336	java.util.regex.Pattern\$Dollar (java.base@11.0.2)
28:	29013	464208	java.util.regex.Pattern\$Begin (java.base@11.0.2)
29:	1391	169712	java.lang.Class (java.base@11.0.2)
30:	2897	92704	java.util.concurrent.ConcurrentHashMap\$Node (java.base@11.0.2)
31:	64	28576	Ljava.util.concurrent.ConcurrentHashMap\$Node; (java.base@11.0.2)
32:	436	20928	java.lang.invoke.MemberName (java.base@11.0.2)
33:	578	18496	java.lang.invoke.LambdaForm\$Name (java.base@11.0.2)
34:	411	13248	Ljava.lang.Class; (java.base@11.0.2)
35:	182	12376	Ljava.lang.ref.SoftReference; (java.base@11.0.2)
36:	380	12160	java.lang.invoke.MethodType\$ConcurrentWeakInternSet\$WeakEntry (java.base@11.0.2)

可以看到各个 class 的创建的实例的数目和占用内存的数量，例如

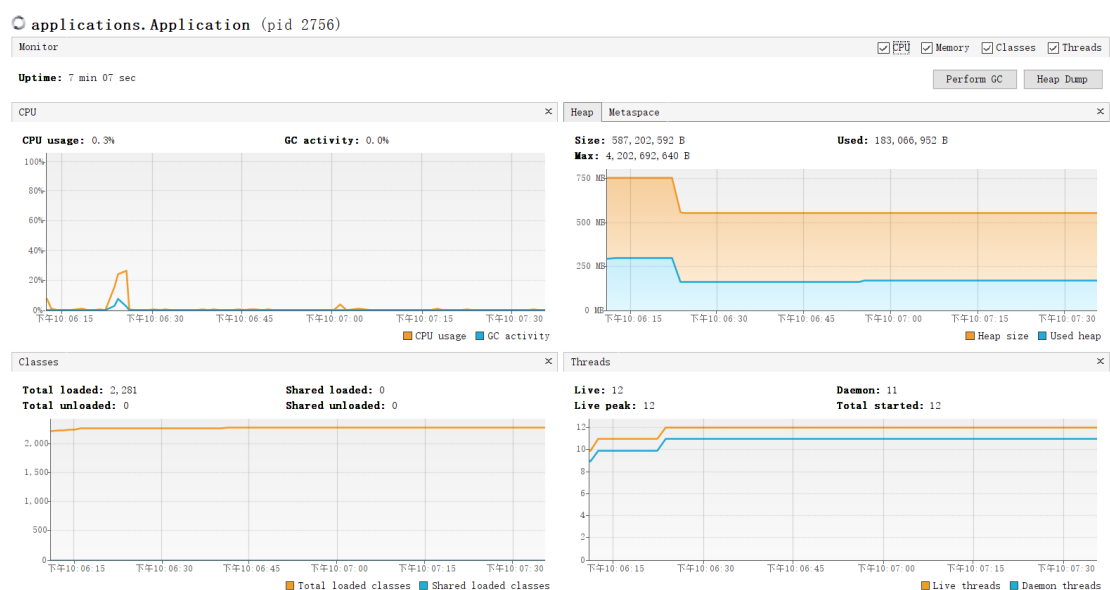
```
5:          320000          25600000  physicalobject.Planet
```

说明当前创建了 320000 个 planet 对象的实例，这些实例共占用了 25600000 字节的内存空间

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

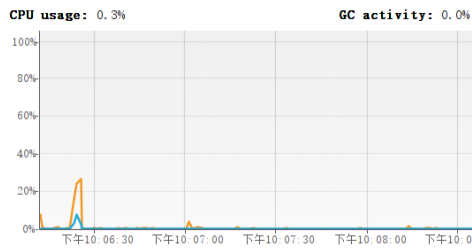
采用 VisualVM

运行程序后看到统计界面如下

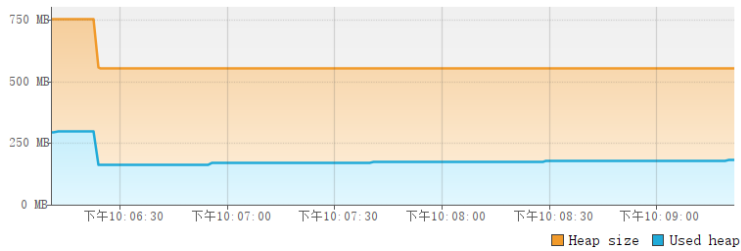


左上方可以看到程序的 CPU 占用率为%1.3，class 可以看到共装载了 2300 个左右的类



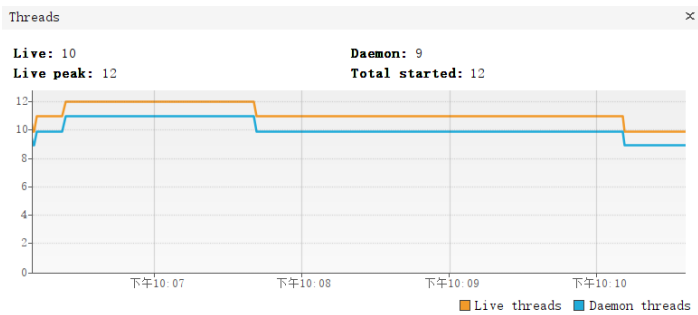


右上方是 heap 运行的占用情况

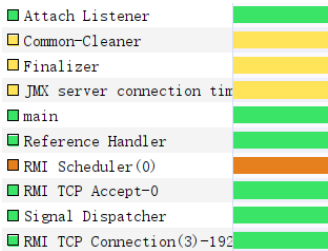


出现一个下降的陡坡是我强制执行了一次 perform GC 的结果，说明 GC 的实际效率还是很高的

而右下方可以看到程序的进程个数

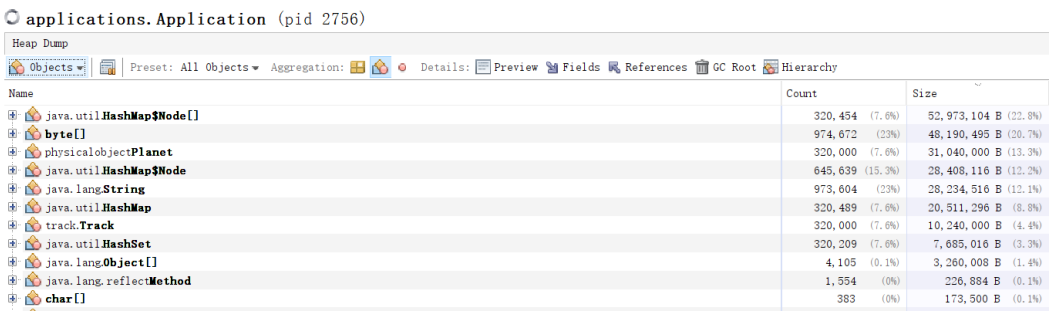


切换到 TREAD 界面，可以看到



说明程序本身实际运行的进程只有 main 进程，其他的都是垃圾回收器和附加的程序运行监听器等

点击 heap dump 可以查看当前的 heap 各个实例的分配情况

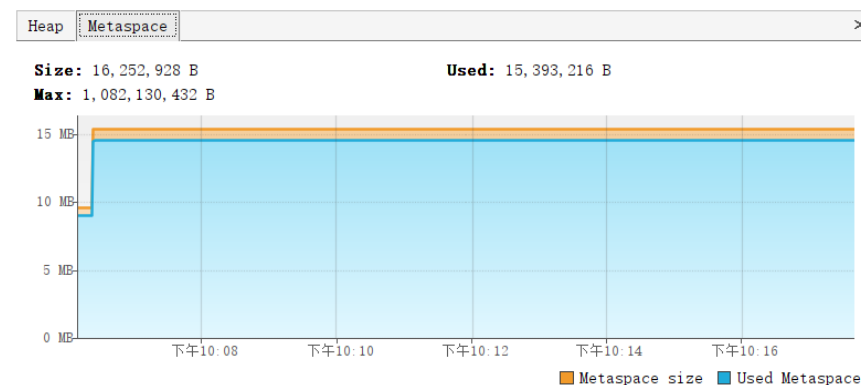


其中可以看到 planet 类实例化了 320000 个对象，和之前用 jmap 分析得到的结果是一致的总的统计结果如下

```

Basic Info:
Total Bytes: 232,563,785
Total Classes: 2,498
Total Instances: 4,230,056
Classloaders: 74
GC Roots: 2,043
Number of Objects Pending for Finalization: 0
    
```

可以看到，程序到目前一共装载了 2498 个类，创建了 4230056 个对象的实例  
 点击 metaspace 按钮可以切换到 metaspace 窗口



可以看到前面有一个增加的陡坡，是 perform GC 的结果，说明 heap 中有部分内容在 GC 后被转移到了 meta space 中

### 3.3.7 分析垃圾回收过程

通过上面的分析可以看到，程序的垃圾回收机制主要如下

- 使用 G1 垃圾回收器，将堆划分为若干个区域 (Region)，通过将对象从一个区域复制到另外一个区域，完成清理工作，每个 Region 被标记成 E、S、O、H，分别表示 Eden、Survivor、Old、Humongous，其中 E、S 属于年轻代，O 与 H 属于老年代
- 当 E 区不能再分配新的对象时，G1 回收器会触发垃圾回收，E 区的对象会移动到 S 区，当 S 区空间不够的时候，E 区的对象会直接移动到 O 区

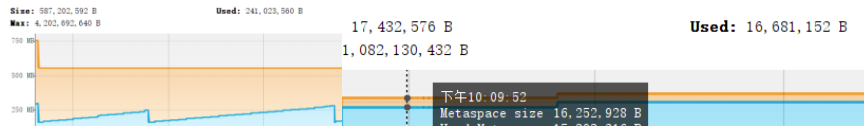
S1C	S0U	S1U	EC	EU
2048.0	0.0	2048.0	61440.0	46080.0
10240.0	0.0	10240.0	74752.0	6144.0
10240.0	0.0	10240.0	160768.0	97280.0

- 程序开始时，读入 plantFile1，创建大量的临时实例变量来构建具体的 Planet 和 track，此时的 Eden 区域呈现出增长，到一定时刻执行垃圾回收，一些还有使用的实例对象进入到 S 区域和 O 区域

```

Eden regions: 16->0(70)
Survivor regions: 3->3(3)
Old regions: 65->68
    
```

- 程序读入停止后，堆空间呈现稳定趋势，而垃圾回收执行后使用的堆空间会下降，同时因为读入的信息已经构造成了 planet 系统，存储到了系统 class 的成员变量中，因此 metaspace 在读入时增长，读入结束后呈现稳定的趋势



### 3.3.8 配置 JVM 参数并发现优化的参数配置

- 尝试减小一些初始和最大的 heap 尺寸

```
-Xmx32m
-Xms4m
```

结果导致 heap 容量太小内存溢出了，也出现了大量的 humogousGC 和 majorGC

```
[15.764s][info] [gc,metaspace] GC(10) Metaspace: 9089K->9089K(1058816K)
[15.764s][info] [gc] GC(10) Pause Full (G1 Humongous Allocation) 20M->20M(32M) 3.922ms
[15.764s][info] [gc,cpu] GC(10) User=0.00s Sys=0.00s Real=0.00s
[15.764s][info] [gc,marking] GC(7) Concurrent Mark From Roots 10.674ms
[15.764s][info] [gc,marking] GC(7) Concurrent Mark Abort
[15.764s][info] [gc] GC(7) Concurrent Cycle 11.099ms
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at java.base/java.util.Arrays.copyOfRange(Arrays.java:4030)
    at java.base/java.lang.StringCoding.decodeUTF8(StringCoding.java:710)
    at java.base/java.lang.StringCoding.decode(StringCoding.java:318)
```

- 增大一些初始和最大的 heap 尺寸

```
-Xmx4096m
-Xms2048m
```

发现 GC 的数量减少了，而且程序读入文件的消耗时间有所减少

```
[9.007s][info] [gc,heap] GC(8) Eden regions: 150->0(702)
[9.007s][info] [gc,heap] GC(8) Survivor regions: 17->21(21)
[9.007s][info] [gc,heap] GC(8) Old regions: 119->135
[9.007s][info] [gc,heap] GC(8) Humongous regions: 35->35
[9.007s][info] [gc,metaspace] GC(8) Metaspace: 9391K->9391K(1058816K)
[9.007s][info] [gc] GC(8) Pause Young (Normal) (G1 Evacuation) 0.000ms
[9.007s][info] [gc,cpu] GC(8) User=0.17s Sys=0.02s Real=0.19s
生成这个行星系统的速度为1802ms
```

- 更改垃圾回收的策略

#### 1. 串行垃圾处理器

```
-Xms2048m
-XX:+UseSerialGC
```

使用串行的垃圾回收器程序读入 file1 速度略有下降

```
[3.662s][info] [gc] GC(1) Pause Young (Allocation Failure)
[3.662s][info] [gc,cpu] GC(1) User=0.23s Sys=0.05s Real=0.33s
生成这个行星系统的速度为2037ms
```

#### 2. 并行垃圾处理器

```
[0.056s][info] [gc] Using Parallel
[0.056s][info] [gc,heap,coops] Heap at 0x0000000000000000
```

速度大幅下降

```
[6.956s][info] [gc] GC(1) Pause Young (All)
[6.956s][info] [gc,cpu] GC(1) User=0.61s Sys=1.00s Real=1.61s
生成这个行星系统的速度为3399ms
```

#### 3. 并发标记扫描垃圾回收器



[gc] Using Concurrent Mark Sweep

发现性能也并不好

```
[11.827s][info ][gc          ]
[11.827s][info ][gc,cpu       ]
生成这个行星系统的速度为2947ms
```

经过一番实验，我发现最好还是使用最新的 G1 垃圾处理器

- 增加处理策略的细节

**-XX:+UseStringDeduplication**

这样 G1 处理器对垃圾指针中重复的字符串可以进行处理

```
[2.133s][info ][gc,stringdedup] Concurrent String Deduplication 42.4M->42.1M(281.8K) avg 0.6% (1.624s,
```

但是效果并不理想

处理字符串的时候也大大增加了垃圾回收时间

```
Concurrent String Deduplication (4.224s)
Concurrent String Deduplication 3213.7K->1071.2K(2142.5K) avg 16.7% (4.224s, 4.259s) 34.587ms
```

最后发现，在使用 G1 回收器配合 4096 的分配内存下程序运行的效率是最高的

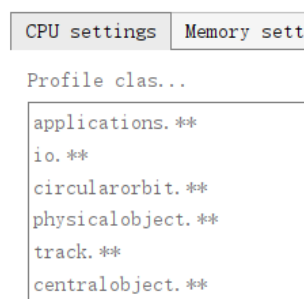
```
[9.007s][info ][gc,heap      ] GC(8) Eden regions: 150->0(702)
[9.007s][info ][gc,heap      ] GC(8) Survivor regions: 17->21(21)
[9.007s][info ][gc,heap      ] GC(8) Old regions: 119->135
[9.007s][info ][gc,heap      ] GC(8) Humongous regions: 35->35
[9.007s][info ][gc,metaspace ] GC(8) Metaspace: 9391K->9391K(1058K)
[9.007s][info ][gc          ] GC(8) Pause Young (Normal) (G1 Evacuation)
[9.007s][info ][gc,cpu       ] GC(8) User=0.17s Sys=0.02s Real=0.19s
生成这个行星系统的速度为1802ms
```

## 3.4 Dynamic Program Profiling

### 3.4.1 使用 JMC 或 VisualVM 进行 CPU Profiling

使用 VisualVM 进行观察

首先设置要进行观察的包和类的参数



点击 CPU 按钮开始记录，然后输入命令使程序读入 file1

得到的程序 CPU 占用结果如下

Name	Total Time	Total Time (CPU)	Invocations
main	3,753 ms (100%)	2,420 ms (100%)	6
io.SystemFactory.generateStellarSystem (String)	3,709 ms (98.8%)	2,373 ms (98.1%)	1
Self time	3,079 ms (82%)	1,837 ms (75.9%)	1
circularorbit.ConcreteCircularOrbit.addObject (physicalobject.PhysicalObject)	413 ms (11%)	353 ms (14.6%)	320,000
Self time	307 ms (8.2%)	308 ms (12.7%)	320,000
physicalobject.Plane.hashCode ()	105 ms (2.8%)	44.7 ms (1.8%)	320,000
physicalobject.Plane.hashCode ()	123 ms (3.3%)	60.3 ms (2.5%)	320,000
circularorbit.ConcreteCircularOrbit.addTrack (track.Track)	93.3 ms (2.5%)	122 ms (5.1%)	320,000
circularorbit.ConcreteCircularOrbit.<clinit> ()	0.354 ms (0%)	0.0 ms (0%)	1
circularorbit.ConcreteCircularOrbit.setCentralObject (centralobject.CentralObject)	0.005 ms (0%)	0.0 ms (0%)	1
io.CircularIO.readFile (io.CircularChannelReader)	42.5 ms (1.1%)	46.8 ms (1.9%)	1
io.CircularChannelReader.read ()	42.4 ms (1.1%)	46.8 ms (1.9%)	1
Self time	0.037 ms (0%)	0.0 ms (0%)	1
applications.Menu.showCommon ()	0.787 ms (0%)	0.0 ms (0%)	1
io.SystemFactory.<clinit> ()	0.363 ms (0%)	0.0 ms (0%)	1
applications.Menu.showStellarSystem ()	0.114 ms (0%)	0.0 ms (0%)	1

可以看到，其中使用内存映射的 NIOchannel 读入系统总共只有用了 42.4ms

io.CircularChannelReader.read ()	42.4 ms (1.1%)	46.8 ms (1.9%)
----------------------------------	----------------	----------------

程序读入的时候主要的时间消耗来源于系统的构建，这里已经是我优化后的系统构建程序，时间复杂度几乎为  $O(n)$ ，读入时间消耗在 0.37s 左右

main	3,753 ms (100%)	2,420 ms (100%)	6
io.SystemFactory.generateStellarSystem (String)	3,709 ms (98.8%)	2,373 ms (98.1%)	1
Self time	3,079 ms (82%)	1,837 ms (75.9%)	1
circularorbit.ConcreteCircularOrbit.addObject (physicalobject.PhysicalObject)	413 ms (11%)	353 ms (14.6%)	320,000

而通过观察可以发现，自己书写的 ADT 的时间消耗已经很小，总消耗几乎都在 0.1s 左右

physicalobject.Plane.hashCode ()	123 ms (3.3%)
circularorbit.ConcreteCircularOrbit.addTrack (track.Track)	93.3 ms (2.5%)
circularorbit.ConcreteCircularOrbit.<clinit> ()	0.354 ms (0%)
circularorbit.ConcreteCircularOrbit.setCentralObject (centralobject.CentralObject)	0.005 ms (0%)

最大的是 addObject, 总共消耗的时间是 0.4 秒

circularorbit.ConcreteCircularOrbit.addObject (physicalobject.PhysicalObject)	413 ms (11%)
Self time	307 ms (8.2%)
physicalobject.Plane.hashCode ()	105 ms (2.8%)

而分析程序可知，我使用的存储 object 的数据结构是 HashSet，而每次加入的时候由于 set 本身的特性，需要和前面加入的物体进行比对，由于是 HashSet，先根据 hashCode 值来找到对应的 box 来进行比较，而 object 的 equals 方法根据 label 比较，这样会使得程序在加入集合的时候消耗一定时间

```
@Override
public int hashCode() {
    return this.getLabel().hashCode();
}
```

重写的 hashCode 返回值是 JDK 自动生成的 hashCode 值，这样每次加入的时间复杂度几乎已经降低到最小

而通过上面分析，可以得知程序在读入文件的时候的时间消耗主要是正则表达式匹配的时间消耗，其他地方的消耗已经大幅度优化

而如果不使用 profiler，程序读入 31 万行的大型 planet 文件只需要 2s 左右

```
[9.007s][info][gc,heap] GC(8) Eden regions: 150->0(702)
[9.007s][info][gc,heap] GC(8) Survivor regions: 17->21(21)
[9.007s][info][gc,heap] GC(8) Old regions: 119->135
[9.007s][info][gc,heap] GC(8) Humongous regions: 35->35
[9.007s][info][gc,metaspace] GC(8) Metaspace: 9391K->9391K(1058K)
[9.007s][info][gc] GC(8) Pause Young (Normal) (G1 Evacuation)
[9.007s][info][gc,cpu] GC(8) User=0.17s Sys=0.02s Real=0.6s
生成这个行星系统的速度为1802ms
```

尝试进行一些操作，例如计算熵值

这个轨道系统物体分布的熵值为: 18.287712379572596

5月 27, 2019 10:47:24 上午 applications.Actions commonAction

信息: 计算出系统的熵值为 18.287712379572596

apis.CircularOrbitApis.getObjectDistributionEntropy(circularorbit.Circ	810 ms	(0.1%)
Self time	398 ms	(0%)
circularorbit.ConcreteCircularOrbigeTrack (int)	125 ms	(0%)
circularorbit.ConcreteCircularOrbigeTrackNum ()	105 ms	(0%)
circularorbit.ConcreteCircularOrbigeObjNum ()	98.3 ms	(0%)
track.Track.getObjNum ()	83.2 ms	(0%)
Self time	6.33 ms	(0%)

可以看到调用的 countEntropy 方法的时间消耗如上所示

消耗时间最大的是 getTrack 方法，如果 track 用 set 来存储，那么就需要遍历一次 set 来找到需要的 track，这里我优化后采用 arraylist 存储 track

```
private final List<Track> tracks = new ArrayList<Track>();
// the tracks in this system
// RI: should be Track type, CentralObject should not be nu
// ..
```

这样 getTrack 方法只需要 index 在 O(1)的时间复杂度就可以找到对应的 track，总共计算熵值只有消耗了不到 1s 的时间

验证其他操作，时间复杂度也在合理的限度内

删除整条轨道

5月 27, 2019 10:54:22 上午 applications.Actions commonAction

信息: 删除了第 6 条轨道上的所有物体

circularorbit.ConcreteCircularOrbideleteTrack (int)	2.77 ms	(0%)
Self time	2.11 ms	(0%)
physicalobject.PlanethashCode ()	0.651 ms	(0%)
track.Track.equals (Object)	0.006 ms	(0%)

计算一定时间后各个行星位置，更新行星位置

请输入系统运行的时间

1000

5月 27, 2019 10:55:48 上午 applications.Actions stellarAction

信息: 计算出系统运行时间为 1000.0 后的各个行星的位置

消耗时间为 0.114s

applications.ActionsstellarSystemAction (circularorbit.StellarSystem, in	2,413 ms	(0.1%)
Self time	2,299 ms	(0.1%)
circularorbit.StellarSystemmoveByTime (physicalobject.Planet, double	114 ms	(0%)

程序在各个操作以及读入系统的时候方法执行正常

### 3.4.2 使用 VisualVM 进行 Memory profiling

设置 memory 监控的类的参数

CPU settings	Memory settings	JDBC settings
Profile clas...		
applications.**		
io.**		
circularorbit.**		
physicalobject.**		
track.**		
centralobject.**		

运行程序读入文件 file1，可以看到监控显示如下

Profiling results

Results: | Collected data:

Name	Live Bytes	Live Objects	Allocated Objects	Generat...
physicalobjectPlanet	25,600,000 B (99%)	320,000 (96.9%)	320,000 (50%)	38
java.lang.Object<init> ()	25,600,000 B (99%)	320,000 (96.9%)	320,000 (50%)	38
physicalobject.ConcretePhysicalObject<init> ()	25,600,000 B (99%)	320,000 (96.9%)	320,000 (50%)	38
physicalobject.Plane<init> (String, String, String)	25,600,000 B (99%)	320,000 (96.9%)	320,000 (50%)	38
io.SystemFactory.generateStellarSystem (String)	25,600,000 B (99%)	320,000 (96.9%)	320,000 (50%)	38
applications.Applicationmain (String[])	25,600,000 B (99%)	320,000 (96.9%)	320,000 (50%)	38
track.Track	249,648 B (1%)	10,402 (3.1%)	320,000 (50%)	24
java.lang.Object<init> ()	249,648 B (1%)	10,402 (3.1%)	320,000 (50%)	24
track.Track<init> (double)	249,648 B (1%)	10,402 (3.1%)	320,000 (50%)	24
io.SystemFactory.generateStellarSystem (String)	249,648 B (1%)	10,402 (3.1%)	320,000 (50%)	24
applications.Applicationmain (String[])	249,648 B (1%)	10,402 (3.1%)	320,000 (50%)	24
circularorbitStellarSystem	32 B (0%)	1 (0%)	1 (0%)	1
centralObjectStellar	32 B (0%)	1 (0%)	1 (0%)	1
applicationsActions	0 B (0%)	0 (0%)	0 (0%)	0

其中可以看到，读入文件构建系统总共生成了 320000 个 planet 的实例，最后存活的有 320000 个

physicalobjectPlanet	25,600,000 B (99%)	320,000 (96.9%)	320,000 (50%)
track.Track	249,648 B (1%)	10,402 (3.1%)	320,000 (50%)

而对于 track，系统在生成的时候创建了 320000 个临时的 track，最后存活下来的只有 10402 个 track，剩余的 track 已经及时被垃圾回收器清理掉了

这说明系统实际上是有重复的轨道的，即有多个行星在同一个轨道上的情况

3, 7.32e6, 5640, CW, 287>	Find: 3.84e9
20, 1.49e8, 6570, CW, 76>	Replace with:
71, 7.16e8, 3862, CW, 122	Direction
13, 2.40e5, 364, CW, 61>	Scope
3, 84e9, 5349, CCW, 293>	
1 7 20e8 8701 CCW 261	

观测的结果和期望的结果是一致的

## 3.5 Memory Dump Analysis and Performance Optimization

### 3.5.1 内存导出

使用 visual VM 工具进行内存导出

[heapdump] 上午11:34:56	Summary
ote	Basic Info:
CoreDumps	Total Bytes: 174,394,410
pshots	Total Classes: 2,652
applications.Application (pid 13520), 上午1	Total Instances: 3,346,981
	Classloaders: 83
	GC Roots: 2,159
	Number of Objects Pending for Finalization: 0

得到内存导出文件

heapdump-1558928096362.hprof	2019/5/27 星期...	HPROF 文件	188,213 KB
------------------------------	-----------------	----------	------------

### 3.5.2 使用 MAT 分析内存导出文件

使用 MAT 进行内存导出文件解析

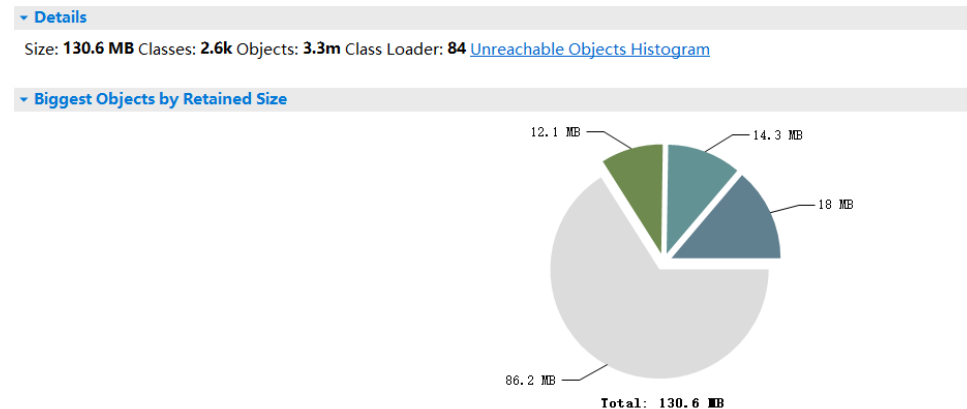
Opening...

Parsing heap dump from 'C:\Users\Administrator\Desktop\softwareConstruction\Lab5\1170300529-lab5\doc\heapdump-1558928096362.hprof'

Cancel

可以看到报告如下

## ● overview



**java.lang.Thread @ 0x82d74c88 main**

Shallow Size: **120 B** Retained Size: **18 MB**

可以看到，整个程序占用的内存空间大小为 130.6MB，共装载了 2600 个类，3.3m 个实例对象

而其中 Stellar system 的总引用对象占内存大小为 12.1MB

**circularorbit.StellarSystem @ 0x834fb948**

Shallow Size: **32 B** Retained Size: **12.1 MB**

main 线程的总引用对象占内存大小为 18MB

**java.lang.Thread @ 0x82d74c88 main**

Shallow Size: **120 B** Retained Size: **18 MB**

## ● histogram

Class Name	Objects	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
byte[]	974,359	44,181,352	>= 44,181,352
physicalobject.Planet	320,000	25,600,000	>= 71,680,000
java.lang.String	973,366	23,360,784	>= 46,981,536
java.util.HashMap\$Node	650,268	20,808,576	>= 21,170,696
org.netbeans.lib.profiler.server.Pr...	355,540	14,221,600	>= 14,221,600
java.util.HashMap\$Node[]	4,953	4,578,640	>= 25,741,864
java.lang.ref.WeakReference[]	2	2,065,304	>= 15,118,032
java.lang.Object[]	4,166	241,368	>= 437,456
java.util.HashMap	4,992	239,616	>= 25,968,696
char[]	385	185,800	>= 185,800
java.util.concurrent.ConcurrentHa...	3,468	110,976	>= 300,424
java.lang.Double	4,502	108,048	>= 108,432
track.Track	4,501	108,024	>= 13,012,192
int[]	693	89,104	>= 89,104
java.lang.ref.WeakReference	2,478	79,296	>= 79,296
java.util.HashSet	4,710	75,360	>= 25,438,448
java.lang.invoke.LambdaForm\$Na...	1,884	60,288	>= 118,176
java.lang.reflect.Method	624	54,912	>= 84,896
java.lang.invoke.MemberName	921	36,840	>= 73,784
java.util.LinkedHashMap\$Entry	887	35,480	>= 51,544
java.util.concurrent.ConcurrentHa...	86	32,512	>= 382,536
java.lang.Class	2,571	32,200	>= 17,702,424

从 histogram 视图中可以看到，当前类中占用内存大小的情况（包括自身占用内存 shallow heap 和引用对象占用内存 retained heap）

其中占用内存空间最大的类是 byte[]数组，有 974359 个实例对象，自身就占用了 44181352 的内存空间，是需要排查的隐患

byte[]	974,359	44,181,352	>= 44,181,352
physicalobject.Planet	320,000	25,600,000	>= 71,680,000

接下来占用空间较大的是 String 类

java.lang.String	973,366	23,360,784	>= 46,981,536
------------------	---------	------------	---------------

使用 list object 后发现是程序构建系统时建立的临时 String，可以进行优化

java.lang.String @ 0x9898efe8 Dark	24	48
java.lang.String @ 0x9898efb8 Liquid	24	48
java.lang.String @ 0x9898ef88 p289261	24	48
java.lang.String @ 0x9898ee78 Yellow	24	48
java.lang.String @ 0x9898ee48 Gas	24	48
Total: 44 of 973,366 entries; 973,322 more		

最后一个较大的是 hashmap 类

java.util.HashMap\$Node	650,268	20,808,576	>= 21,170,696
-------------------------	---------	------------	---------------

使用 list object 功能查看发现，里面主要存储的是构建好系统的系统中的各个 object 和对应的信息

java.util.HashMap\$Node @ 0x98990388	32	64
<class> class java.util.HashMap\$Node @	8	32
value java.lang.Object @ 0x82d7a610	16	16
next java.util.HashMap\$Node @ 0x9005	32	32
key physicalobject.Planet @ 0x989902c6	80	224
Total: 4 entries		
java.util.HashMap\$Node @ 0x98990368	32	32
java.util.HashMap\$Node @ 0x98990218	32	32

## dominator tree

Class Name	Shallow Heap	Retained Heap	Percentage
<Regex>	<Numeric>	<Numeric>	<Numeric>
java.lang.Thread @ 0x82d74c88 main Thread	120	18,888,336	13.79%
class org.netbeans.lib.profiler.server.ProfilerRuntimeObjLiveness	16	15,036,376	10.98%
circularorbit.StellarSystem @ 0x834fb948	32	12,647,176	9.23%
class org.netbeans.lib.profiler.server.ProfilerRuntime @ 0x8317	32	1,200,096	0.88%
class java.time.zone.ZoneRulesProvider @ 0x82d00440 System Class	16	206,184	0.15%
class sun.util.calendar.ZoneInfoFile @ 0x82cce8e0 System Class	120	151,128	0.11%
class org.netbeans.lib.profiler.server.ProfilerInterface @ 0x8317	200	93,672	0.07%
class sun.util.locale.provider.LocaleProviderAdapter @ 0x82cb	32	68,488	0.05%
java.util.HashSet @ 0x82c5f130	16	63,152	0.05%
char[28672] @ 0x830d6418 \ufffd\ufffd\ufffd\ufffd\ufffd	57,360	57,360	0.04%
class sun.nio.cs.GBK @ 0x82d6beb0 System Class	32	55,584	0.04%
class org.netbeans.lib.profiler.server.ProfilerRuntimeMemory @	80	55,280	0.04%
class sun.util.resources.Bundles @ 0x82cb1340 System Class	24	53,712	0.04%
java.util.zip.ZipFile\$Source @ 0x82db35f8	64	52,024	0.04%
char[256] @ 0x830bc110	1,040	51,440	0.04%
class sun.util.cldr.CLDRLocaleDataMetaInfo\$TZCanonicalID	8	51,040	0.04%
class java.lang.CharacterData00 @ 0x82c2e900 System Class	40	50,232	0.04%
com.sun.management.internal.DiagnosticCommandImpl @ 0x82	40	48,744	0.04%
class java.io.ObjectStreamClass\$Caches @ 0x82c729a0 System Class	16	44,232	0.03%
class java.lang.invoke.MethodType @ 0x8306c7b8 System Class	48	39,928	0.03%
class jdk.internal.loader.BuiltinClassLoader @ 0x82d5a8f0 System Class	16	38,280	0.03%
java.util.zip.ZipFile\$Source @ 0x82dcb290	64	33,896	0.02%
sun.security.provider.Sun @ 0x82c2a828	104	32,168	0.02%
class java.lang.System @ 0x82c2be00 System Class	40	27,960	0.02%

支配树视图以实例对象的维度展示当前堆内存中 Retained Heap 占用最大的对象，以及依赖这些对象存活的对象的树状结构

从中可以看到，当前占用内存最大的对象是 main 线程，占用比例为 13.79%

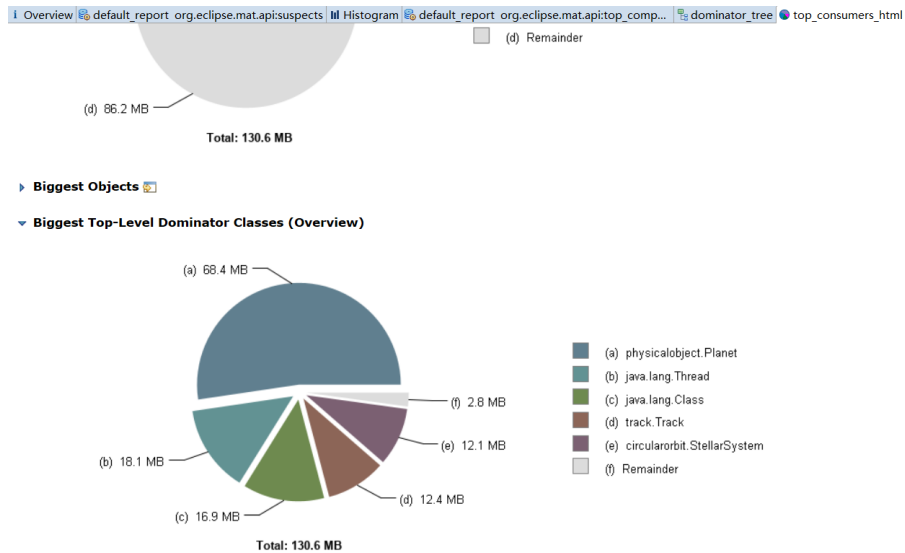
将 main 点开 Dominator Tree 实例对象左侧的“+”，展示出下一层，这样层层展开后发现占用内存的其实是读入文件后的临时 byte 数组，还没有释放掉，可以作为重点优化对象



java.lang.Thread @ 0x82d74c88: main Thread	120	18,888,336
java.lang.ThreadLocal\$ThreadLocalMap @ 0x830bbbc8	24	18,885,504
java.lang.ThreadLocal\$ThreadLocalMap\$Entry[32] @ 0x833dd870	144	18,885,480
java.lang.ThreadLocal\$ThreadLocalMap\$Entry @ 0x830ec820	32	18,884,280
java.lang.StringCoding\$Result @ 0x830ec840	24	18,884,248
byte[18884208] @ 0x84800000: Stellar ::= <ThreeBody,6.96392e5,1.9885e30>.NumOfTracks ::= 32000	18,884,224	18,884,224

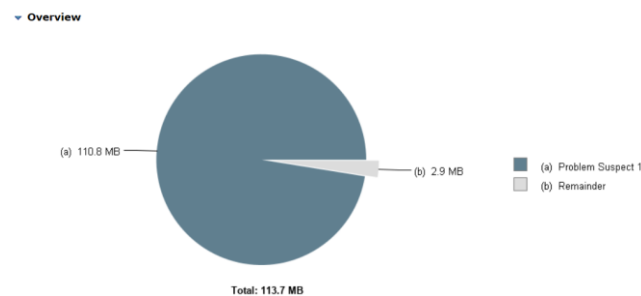
而次之的是 netbean 的 profiler，这个关闭 visual VM 的分析程序关联后可以关闭再往后是系统对象 stellarSystem，这里面包含系统的行星和恒星的各项信息，以及它们对应的轨道数目，轨道半径，对应的映射关系等等

### ● top consumers



top consumers 视图中可以看到，当前占用内存最大的是 planet 对象，存储在 system 里面，是从文件读入信息后构建的，和预期一致  
而第二个 Thread 占用的大小为 18.1MB，比预期要大，猜测是刚才分析看到的读入文件时产生的临时 byte 数组，还没有被垃圾回收释放掉

### ● Leak Suspects



(注：为避免 VM 的 profile 影响，我退出了 VM 重新导出了一份 dump 文件)

#### ▼ Problem Suspect 1

The thread **java.lang.Thread @ 0x82d723a8 main** keeps local variables with total size **116,227,400 (97.47%)** bytes.

The memory is accumulated in one instance of **"circularorbit.StellarSystem"** loaded by **"jdk.internal.loader.ClassLoaders\$AppClassLoader @ 0x82d85cb0"**.

The stacktrace of this Thread is available. [See stacktrace.](#)

#### Keywords

circularorbit.StellarSystem  
jdk.internal.loader.ClassLoaders\$AppClassLoader @ 0x82d85cb0  
[Details >](#)

内存泄漏分析报告中可以看到，主线程 main 里面的 byte 数组占用了 94.47%空间，很可能出现了内存泄漏，而对比之前的分析，猜测可能就是读入文件时产生的临时 byte 数组

### 3.5.3 发现热点/瓶颈并改进、改进前后的性能对比分析

对比上面的分析可知，当前的主要内存占用和泄漏出现在读入文件时构建的临时 byte 数组，定位到 IO 方法内部，发现

```
byte[] bytes = new byte[inMappedByteBuffer.limit()];
inMappedByteBuffer.get(bytes); //read
String inputString = new String(bytes);
return inputString;
```

这里读入文件使用的是 NIO 的 channel 配合文件内存映射，读入的时候需要临时构建一个 byte 数组，而实际需要的是转换成 String 后的 String 对象，byte 数组被 bytes 所引用，还有指针指向，因此无法回收

在这里取消 bytes 的引用，人工调用系统垃圾回收函数进行垃圾回收，释放临时数组占用的空间

```
String inputString = new String(bytes);
bytes = null;
System.gc();
return inputString;
```

将构建 planet 时的临时变量也移动到循环外部来重复利用

```
if (planetMatcher.find()) {
    double size = Double.parseDouble(sizeStr);
    double beginAngle = Double.parseDouble(beginAngleStr);
    double velocity = Double.parseDouble(velocityStr);
    String direction = planetMatcher.group(1);
    boolean booleanDirection = Boolean.parseBoolean(direction);
    String type = planetMatcher.group(2);
    String form = planetMatcher.group(3);
    String color = planetMatcher.group(4);
    Planet planet = new Planet(size, beginAngle, velocity, direction, booleanDirection, type, form, color);
```

### 3.5.4 在 MAT 内使用 OQL 查询内存导出

- 查询当前实例化的所有行星对象

```
SELECT * FROM instanceof physicalobject.Planet
```

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
> physicalobject.Planet @ 0x8b86cf20	80	224
> physicalobject.Planet @ 0x8b86ce20	80	224
> physicalobject.Planet @ 0x8b86cd20	80	224
> physicalobject.Planet @ 0x8b86cc20	80	224
> physicalobject.Planet @ 0x8b86cb20	80	224
> physicalobject.Planet @ 0x8b86ca20	80	224
> physicalobject.Planet @ 0x8b86c920	80	224
> physicalobject.Planet @ 0x8b86c820	80	224
> physicalobject.Planet @ 0x8b86c720	80	224
> physicalobject.Planet @ 0x8b86c620	80	224
> physicalobject.Planet @ 0x8b86c520	80	224
> physicalobject.Planet @ 0x8b86c420	80	224
> physicalobject.Planet @ 0x8b86c320	80	224



可以看到当前每一个实例化的 planet 对象的占用空间和引用空间的大小

- 查询长度大于 1000 的字符串

```
SELECT * FROM java.lang.String s WHERE s.value.@length > 1000
```

可以看到当前长度大于 1000 的字符串有很多是 Unicode 码，推测是输出的菜单的字符串

>	java.lang.String @ 0x82f4f2c0	C:\Users\Administrator\Desktop	24	1,360
>	java.lang.String @ 0x82f4e838	C:\Program Files\Java\jdk-11.0.2	24	2,448
>	java.lang.String @ 0x82e6ff40	af af-NA af-ZA agq agq-CM ak a	24	4,200
>	java.lang.String @ 0x82cb7d80	\u0000\u0000\u0000\u0000\u0000	24	11,816
>	java.lang.String @ 0x82cb5d48	\u0000\u00000 0@P`p\u0080\u00	24	4,136
>	java.lang.String @ 0x82cb4e48	\u4800\u100fu4800\u100fu48	24	3,792

- 查询占用空间大小大于 10000 字节的对象

```
SELECT * FROM instanceof java.lang.Object s WHERE s.@usedHeapSize > 10000
```
















可以看到当前长度大于 10000 字节的对象主要是 char 数组对象，其中一部分是临时数组和菜单输出

[illegible]

- 查看 physicalObject 及其子类的实例数量以及占用空间大小

```
SELECT * FROM instanceof physicalobject.ConcretePhysicalObject
```

输出如下

>	 physicalobject.Planet @ 0x8b86cf20	80	224
>	 physicalobject.Planet @ 0x8b86ce20	80	224
>	 physicalobject.Planet @ 0x8b86cd20	80	224
>	 physicalobject.Planet @ 0x8b86cc20	80	224
>	 physicalobject.Planet @ 0x8b86cb20	80	224
>	 physicalobject.Planet @ 0x8b86ca20	80	224
>	 physicalobject.Planet @ 0x8b86c920	80	224
>	 physicalobject.Planet @ 0x8b86c820	80	224
>	 physicalobject.Planet @ 0x8b86c720	80	224
>	 physicalobject.Planet @ 0x8b86c620	80	224
>	 physicalobject.Planet @ 0x8b86c520	80	224
>	 physicalobject.Planet @ 0x8b86c420	80	224
>	 physicalobject.Planet @ 0x8b86c320	80	224
>	 physicalobject.Planet @ 0x8b86c220	80	224
>	 physicalobject.Planet @ 0x8b86c120	80	224

Σ+ Total: 33 of 320.000 entries: 319.967 more

可以看到总共实例化了 320000 个 physical Object 对象，每个对象本身占用 80 字节的堆内

存空间

### 3.5.5 观察 jstack/jcmd 导出程序运行时的调用栈

首先输入命令 `jps -l` 查询当前的进程，然后用 `jstack` 跟踪当前进程

```
C:\Users\Administrator>jps -l
17376 applications.Application
2316 jdk.jcmd/sun.tools.jps.Jps
3580

C:\Users\Administrator>jstack 17376
2019-05-27 23:05:05
Full thread dump Java HotSpot(TM) 64-Bit Server VM (11.0.2+9-LTS mixed mode):
```

jstack 可以输出生成 java 虚拟机当前时刻的线程快照，而输入命令后发现 jstack 输出了所有线程的快照，我们只需关注 main 线程即可

```
"main" #1 prio=5 os_prio=0 cpu=593.75ms elapsed=15.42s tid=0x000001a45f5b6800 nid=0x1f14 runnable [0x00000019567fe000]
  java.lang.Thread.State: RUNNABLE
    at java.io.FileInputStream.readBytes(java.base@11.0.2/Native Method)
    at java.io.FileInputStream.read(java.base@11.0.2/FileInputStream.java:279)
    at java.io.BufferedInputStream.read1(java.base@11.0.2/BufferedInputStream.java:290)
    at java.io.BufferedInputStream.read(java.base@11.0.2/BufferedInputStream.java:351)
    - locked <0x000000008a947588> (a java.io.BufferedInputStream)
    at sun.nio.cs.StreamDecoder.readBytes(java.base@11.0.2/StreamDecoder.java:284)
    at sun.nio.cs.StreamDecoder.implRead(java.base@11.0.2/StreamDecoder.java:326)
    at sun.nio.cs.StreamDecoder.read(java.base@11.0.2/StreamDecoder.java:178)
    - locked <0x000000008a57da60> (a java.io.InputStreamReader)
    at java.io.InputStreamReader.read(java.base@11.0.2/InputStreamReader.java:185)
    at java.io.Reader.read(java.base@11.0.2/Reader.java:189)
    at java.util.Scanner.readInput(java.base@11.0.2/Scanner.java:882)
    at java.util.Scanner.next(java.base@11.0.2/Scanner.java:1592)
    at java.util.Scanner.nextInt(java.base@11.0.2/Scanner.java:2258)
    at java.util.Scanner.nextInt(java.base@11.0.2/Scanner.java:2212)
    at applications.Actions.chooseSystem(Actions.java:44)
    at applications.Application.main(Application.java:39)
```

进入系统后的 main 线程函数的调用关系如上，可以看到，从下往上函数是层层调用的关系，依次为 main 函数调用 choosesystem 函数，choosesystem 函数调用 scanner 函数读取用户输入，然后 scanner 函数继续调用更底层的函数，这样依次类推，直到调用到最底层的 Native Method 的 readBytes 函数

读入系统的某一时刻的函数调用关系如下

```
"main" #1 prio=5 os_prio=0 cpu=2062.50ms elapsed=20.51s tid=0x0000024c2f9e4800 nid=0x3fb4 runnable [0x0000006f241fe000]
  java.lang.Thread.State: RUNNABLE
    at java.util.regex.Pattern.compile(java.base@11.0.2/Pattern.java:1753)
    at java.util.regex.Pattern.<init>(java.base@11.0.2/Pattern.java:1427)
    at java.util.regex.Pattern.compile(java.base@11.0.2/Pattern.java:1068)
    at java.util.regex.Pattern.matches(java.base@11.0.2/Pattern.java:1173)
    at io.SystemFactory.generateStellarSystem(SystemFactory.java:143)
    at applications.Application.main(Application.java:68)
```

随机输入一些指令，增加轨道，删除物体，计算一定时间后各个行星运动的位置，可视化输出等，可以看到函数调用关系和预期一致

```
"main" #1 prio=5 os_prio=0 cpu=4312.50ms elapsed=688.64s tid=0x000001a45f5b6800 nid=0x1f14 waiting on condition [0x00000019567ff000]
  java.lang.Thread.State: TIMED_WAITING (sleeping)
    at java.lang.Thread.sleep(java.base@11.0.2/Native Method)
    at circularguis.CircularFrame$DrawCircularOrbit.run(CircularFrame.java:62)
    at circularguis.CircularFrame.launchFrame(CircularFrame.java:42)
    at applications.Actions.stellarSystemAction(Actions.java:477)
    at applications.Actions.chooseAction(Actions.java:83)
    at applications.Application.main(Application.java:90)
```

### 3.5.6 使用设计模式进行代码性能优化

- 使用 Flyweight 设计模式进行代码优化

分析可知, electron 对象的 label 可以初始化为一样的, 因为每一个 electron 都是一样的, 而轨道的差异作为特性可以用一个 map 在

首先设计 `FlightWeightElectronFactory` 类来存储共有的 electron 对象

在成员变量区域设置一个已经初始化完成的电子

```
private Electron electron = new Electron(1);
//the electron used in all the system
```

而同时用一个 map 来建立各个轨道和轨道上的电子数量的对应关系

```
private Map<Track,Integer> electronNum = new HashMap<Track, Integer>();
//the number of electron that each track contains
```

而 putElectron 方法只需在 map 中的对应 track 的 value 加 1 即可, 如果没有这条轨道, 则加入 track 初始化 value 为 1

```
public void putElectron(Track track) {
    if (electronNum.containsKey(track)) {
        int electrons = electronNum.get(track);
        electrons++;
        electronNum.remove(track);
        electronNum.put(track, electrons);
    } else {
        electronNum.put(track, 1);
    }
}
```

这样 getElectron 方法只需要先判断 map 中有没有这条 track, 如果有则将公共电子的参数设置成 track 然后返回即可

```
/**
 * get the instance of a electron according to the trackIndex
 * @param track the track this electron stays
 * @return a Electron
 */
public Electron getElectron(Track track) {
    if (electronNum.containsKey(track)) {
        electron.setTrack(track);
        return electron;
    } else {
        return null;
    }
}
```

这样输出只需输出轨道上有几个电子即可

请选择接下来操作

中心点物体: Rb

第1条轨道: 半径1.0 有2个物体

第2条轨道: 半径2.0 有8个物体

第3条轨道: 半径3.0 有18个物体

第4条轨道: 半径4.0 有8个物体

第5条轨道: 半径5.0 有1个物体

- 对 track 对象采用 object pool 模式

在 `ConcreteCircularOrbit` 类里面再加入一个 Map 来建立轨道半径和轨道的一一映射

```
private final Map<Double,Track> findTrack = new HashMap<Double, Track>();
```

这样, 每次加入新的轨道的时候就同时建立一条这样的映射, 而每次加轨道前都要判断映射

是否已经存在，如果存在就从映射里面直接取出 track 而不建立临时的 track 对象实例

```
if (findTrack.containsKey(track.getRadius())) {
    return false;
}
findTrack.put(track.getRadius(), track);
return tracks.add(track);
```

这样读入文件建立系统的时候就不需要重复遍历 list 来判断有没有相同的轨道，直接找键值即可，复杂读从  $O(n)$  变成  $O(1)$

- 在 social network circle 中加入一个 Map

```
private final Map<String, Friend> findFriend = new HashMap<String, Friend>();
//use to quickly find a certain friend
```

由于 social network circle 在读入文件的时候先解析的是 friend 对象，然后再根据 social tie 来建立 friend 之间的关系，这样每次建立关系都要遍历一遍 set 来找 friend

考虑采取新的策略在加入 friend 的时候同时也在一个 map 中加入  $\langle \text{String}, \text{Friend} \rangle$  的键值对，这样每次要取 friend 都可以再  $O(1)$  时间直接取到

```
public Friend getFriend(String name) {
    return this.findFriend.get(name);
}
```

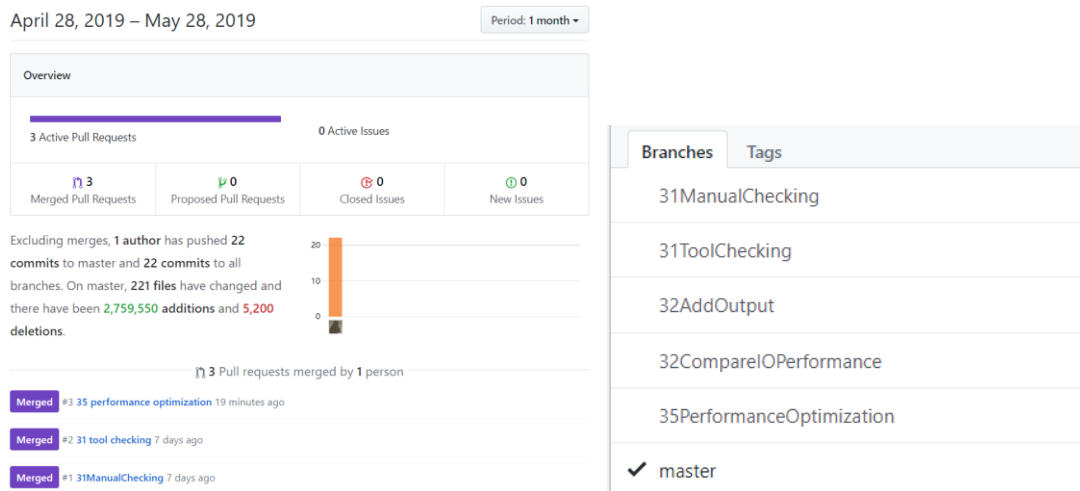
同时的其他方法也需要改造

这样做使得创建 social network circle 的时间从之前的半个小时多骤减到了 6 秒，性能提升非常明显，而输出显示读入的系统正常

```
[8.226s][info    ][gc,metaspace
[8.226s][info    ][gc
[8.226s][info    ][gc,cpu
生成这个社交系统的速度为6318ms
```

### 3.6 Git 仓库结构

gitHub 的 Insight 页面如下



其中还有 32CompareIOPerformance 的 merge 是在本地进行，查看日志可以看到 merge 记录

```
commit 3cd811ad76b83d259da5af6a0118c701fece17ec
Merge: 73f440c 5645049
Author: elenathFGS <fgsok@163.com>
Date: Tue May 21 21:34:04 2019 +0800
```

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。

## 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
2019/5/21	17:25-20:00	手工修改代码，静态走查	完成
2019/5/22	19:23-22:00	用 Google 规范修改代码	完成
2019/5/24	20:00-23:23	写三种输出函数	Channel 输出未完成
2019/5/25	20:12-00:15	完成 channel 输出，测时间	完成
2019/5/26	20:25-22:30	重构系统，优化建系统时间	优化成功，系统构建快速
2019/5/27	20:00-00:30	完成调优，利用各种工具检测	检测后发现热点和其他小问题
2019/5/28	18:00-00:30	完成实验	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
代码走查发现大量不符合谷歌规范的地方，尤其是缩进	网上查，发现可以用 <code>formatter</code>
NIO 的 channel 输出输入不会写	翻阅各种大佬的博客，自己尝试一点点写
系统构建太费时间	找到热点消耗，优化结构，降低循环次数，使用设计模式来优化函数

## 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教训

- 1.代码的规范性很重要，使用严格的规范不但方便自己查看和重构，更是和他人合作时必不可少的
- 2.程序的重构需要耐心，需要缜密的思考，一次重构可能牵一发而动全身，因此除了要对 ADT 设计遵循各种设计原则以外还要考虑程序各个函数类的引用，仔细审查重构的问题
- 3.程序的优化是必须的，我的程序一开始读入构建超大型系统 30min 都整不出来，一看发现自己复杂度高达  $O(n^3)$ ，优化后构建系统仅仅只需 2 秒即可，性能提升是显著的

### 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？

我认为代码看起来美观简洁，可以方便程序猿发现内部问题，及时修改，也方便重构，使得代码“运行起来很美”

我认为应该先放在看起来很美，然后后期调优注意“运行起来很美”

- (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？

会的，使得我的代码看起来很美了，然后大大方便后面的重构

- (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？

因为不同场景需要不一样

查阅后得到

JDK1.0 的时候，所有与输入相关的类都继承于 `InputStream`，所有与输出相关的类都继承于 `OutputStream`。JDK1.1 的时候，增加了面向字符的 IO 类，包括 `Reader` 和 `Writer`。JDK1.4 的时候，增加了 NIO

- (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？

JVM 内存管理比系统的内存管理在更高一层实现了更多的细节和改进，有些地方比如 G1 的分区垃圾回收使得系统内存管理更加高效，我认为 JVM 当前的处理机制已经很高效了

- (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？

我认为 JVM 当前的处理机制已经基本成熟，但是仍然可以面向不同应用场景来优化

- (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？

在程序 GC 不显著，规模较小的时候回报可能较少，但是如果增大程序规模，那么回报是显著的

- (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？

人工分析和运行分析不能很好的揭露程序内部特别是运行的时候的一些问题，例如内存泄漏，垃圾未回收，而通过使用这些工具可以大大方便程序员对程序进行优化

- (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？

我会加油的

- (9) 关于本实验的工作量、难度、deadline。

我觉得阔以

- (10) 到目前为止，你对《软件构造》课程的意见与建议。

希望实验课老师可以给一些设计展示和优秀演示，促进同学们互相交流共同进步