哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

Lab Manuals for

Software Construction

Lab-5
Static and Dynamic Code Analysis and Performance Optimization

School of Computer Science and Technology

Harbin Institute of Technology

Spring 2019

目录

1 实验目标

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

具体训练的技术包括：

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

2 实验环境

实验环境设置请参见 Lab-0 实验指南。

除此之外，本次实验需要你在 Eclipse IDE 中配置并运行 VisualVM 和 Memory Analyzer (MAT)（用于 Java 程序动态性能分析的工具）。请分别访问 https://visualvm.github.io 和 http://www.eclipse.org/mat/，获取更多信息。

还要配置并运行 CheckStyle 和 SpotBugs 工具（代码静态分析工具），请访问 http://checkstyle.sourceforge.net 和 https://spotbugs.github.io/获取更多帮助信息。

本次实验在 GitHub Classroom 中的 URL 地址为：

https://classroom.github.com/a/sizmGb2_

请访问该 URL，按照提示建立自己的 Lab5 仓库并关联至自己的学号。

由于本次实验是 Lab3 和 Lab4 的继续，请首先将你 Lab4 的最终代码复制一

份，推送到 Lab5 的仓库里，然后在其基础上完成本次实验任务。本次实验不能影响 Lab3 和 Lab4 仓库里的内容与 commit 历史。

本地开发时，本次实验只需建立一个项目，统一向 GitHub 仓库提交。实验包含的多项任务分别在不同的目录内开发，具体目录组织方式参见各任务最后一部分的说明。请务必遵循目录结构，以便于教师/TA 进行测试。

3 实验要求

3.1 Static Program Analysis

针对 Lab4 中提交的最新版本代码，进行静态代码分析。

(1) 选定某种特定的 Java 代码规范（例如 Google 或 Oracle 的规范），阅读规范，对你的 Lab4 代码进行人工代码走查（walkthrough and review），做出修改。主要关注命名、布局（空行、空格、缩进、分行等）、注释（java doc、RI、AF、safety from rep exposure、函数 spec 等）、文件/包的组织等。针对你所发现的问题，对代码进行修改，消除这些问题，并提交 git 仓库形成新版本，分支名为 31ManualChecking，然后切换回 master 分支并将 31ManualChecking 分支合并到 master，继续后续任务，但务必要保留 31ManualChecking 分支。

注：本手册后续部分若提及“形成新版本提交 git 仓库，分支名 XXX”，均与这里的要求一致，后续不再重复说明。

以下给出了 Git 的相关操作指南。

```
最初在 master 上工作…
git checkout -b 31ManualChecking    创建新分支
按上面(1)的要求进行代码修改…
git add *
git commit -m "31ManualChecking"     该分支上提交
git checkout master                  切换回 master 分支
git merge 31ManualChecking           合并修改
请不要使用 git branch -d 31ManualChecking
```

(2) 使用 CheckStyle 和 SpotBugs 工具对经过人工走查的 Lab4 代码进行自动的静态代码分析。通过查阅资料理解清楚这两个工具找出的所有问题，并手动进行修改，确保再次运行工具不会出现同样的问题。修改后的代码形成新版本，提交 git 仓库，分支名为 31ToolChecking。

3.2 Java I/O Performance Optimization

(1) Lab3/Lab4 要求程序从外部文本文件读取数据并构造 `CircularOrbit` 对象。在 Lab3/Lab4 的基础上，首先增加一个新功能：将经过各种人工操作之后的 `CircularOrbit` 数据写入一个新的文本文件进行持久化存储（存储格式仍遵循 Lab3 中规定的语法）。完成后，形成新版本提交到 git 仓库，分支名 32AddOutput。

(2) 在文本文件较大、图规模较大的时候，I/O 的效率将影响程序整体性能。为此，请采用至少 3 种不同的 Java I/O 策略对你程序中文件 I/O 进行改进，尽最大可能提高 I/O 效率，例如：

- `Stream`
- `Reader/Writer`
- `Buffer/Channel`
- `Scanner`
- `java.nio.file.Files`
- 你认为合理的其他 `I/O` 机制

如果你的 Lab3/Lab4 程序中已经使用了其中的某一种，只需要考虑其他未实现的机制即可。对你的工厂方法进行改造，采用 Strategy 设计模式，灵活切换读取文件的 I/O 实现策略。对你在本节(1)中实现的“写文本文件”的功能进行类似的改造。

(3) 通过实验来度量不同实现方式的 I/O 性能差异：

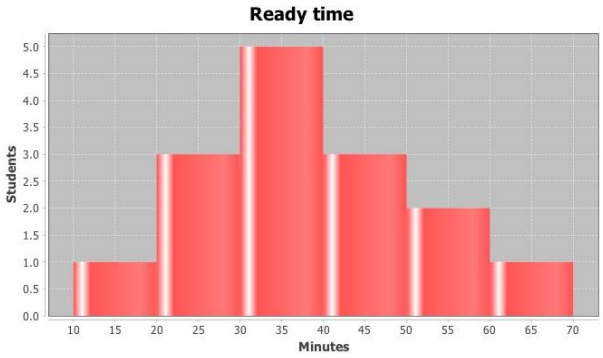
- 从 https://github.com/rainywang/Spring2019_HITCS_SC_Lab5 下载文本文件，你的程序读入文本文件，人工在程序中对该图做一系列操作（自定），进而将内存中的轨道对象数据写入文本文件。GitHub 上所提供的文本文件分别针对 Lab3 中的应用 1、2、4、5，在本实验后续部分，请选择你在 Lab3 中所实现的应用所对应的文件。
- 使用人工代码注入的方式，在你的代码中增加某些代码，用于收集读文件和写文件环节分别消耗的时间，进行对比分析，将结果记录至下表中。（注 1：在做此对比时，一般仅考虑 I/O 环节，不考虑根据读入的数据构造轨道对象的时间；如果你的程序处理逻辑是以“行”为单位从文件中读入数据并逐行加以处理，那么在对比时也可以将读入数据进行语法匹配并转为轨道结构的时间计算在内，但这会使 I/O 时间不准确。无论如何，确保在各种不同 I/O 策略下对比的“时间”具有统一的含义。注 2：在对比时请确保不同 I/O 策略执行时的 JVM 参数配置保持一致。）
- 如果对你的 I/O 效率不满意，请持续保持改进。完成后提交代码至 git 仓库，分支名为 32CompareIOPerformance。

表 1 不同实现方式的 I/O 性能差异

		file1.txt	file2.txt
I/O 策略 1	读文件		
	写文件		
I/O 策略 2	读文件		
	写文件		
…			
I/O 策略 n	读文件		
	写文件		

说明：该表中的 n 取决于你具体实现了多少种 I/O 策略，在省略号处自行增加行数，并把第一列中的“I/O 策略 i”替换为具体的 I/O 策略名称。最右侧两列的题目请修改为你所选择的文件名。空白格子里请填写具体耗费的时间。

- 使用手工方式（例如使用 Excel），或者支持绘图的 Java 第三方 API 对上表结果进行可视化，直观展示不同 I/O 策略的性能对比。绘图 API 可使用例如 JFreeChart (http://www.jfree.org /jfreechart) API，可采用柱状图(Histogram)方式，图形的具体形态和细节请自行设计。



3.3 Java Memory Management and Garbage Collection (GC)

让程序多次读取同一个数据文件并执行程序的各项功能，分别进行以下监控动作：

(1) 在启动你的程序的时候，使用`-verbose:gc` 参数，在控制台输出你的程序的 GC 情况或同时输出至 log 文本文件中（`-Xloggc`: 日志文件路径）。对控制台输出或 log 文件进行简要分析，观察：(a) Minor GC 和 Full GC 发生的频率；(2) 两种 GC 单次耗费的时间；(3) 每次 GC 前后 heap 中各区域占用情况的变化（`-XX:+PrintGCDetails`）。基于你的观察，对你的程序运行过程中内存变化情况进行简要分析。

(2) 使用 `jstat` 命令行工具的`-gc` 和`-gcutil` 参数，对程序的内存使用和垃圾回收情况进行周期性监控，包括对 heap 中各区域（young、old、metaspace）

的 size 和垃圾回收状况的监控与统计分析，根据输出结果判断你的程序的内存回收情况是否正常、存在何种异常。或者使用其他参数查看 heap 的不同区域的更细节的 GC 信息。注：可使用 `jps` 命令列出当前的 Java 进程列表，以下操作中均会用到。

使用 `jmap -heap` 命令行工具查询程序的内存使用信息，包括虚拟机当前所使用的 GC 策略、heap 的配置情况（各区域的大小等）、heap 的使用情况统计；

(3) 使用 `jmap -histo` 命令行工具查询当前装载进内存的各类的实例数目及内存占用情况；

(4) 使用 `jmap -clstats` 命令行工具查看 class loader 的统计信息，即程序执行期间装载的 class 和 method 相关信息。（注：在 JDK8 以前的版本中，使用的指令是 `jmap -permstat`）

(5) 使用 `JMC`、`jconsole` 或 `VisualVM` 工具，尝试着可视化查看程序当前的内存使用，包括 heap 的分配与占用、meta space 情况、装载的类实例统计等。

(6) 根据上述监控结果，分析你的 JVM 在程序执行过程中进行垃圾回收的过程，判断你的程序的内存使用情况与垃圾回收情况是否正常、发现异常情况。

使用你的 JVM 参数设定功能，利用 java 命令行参数或者在 Eclipse 中配置这些参数，对你的程序的 heap 参数进行不同的设置（例如：初始和最大 `heap size`、`young generation space`、`metaspace` 的初始和最大 `size`、`NewRatio`、`SuvivorRatio`、`MinHeapFreeRatio`、`MaxHeapFreeRatio`、GC 模式等），在不同参数设定情况下分别重新执行上述(1)(2)(3)(6)步骤，简要分析在不同内存尺寸参数设定情况下和不同 GC 模式下的 GC 性能变化情况，进而尝试着猜测你的程序的内存参数最优设定方案。

例如：

```
-Xmx32m -Xms4m -Xmn1m -XX:MetaspaceSize=5m
-XX:MaxMetaspaceSize=10m -XX:+UseConcMarkSweepGC
-XX:ParallelCMSThreads=6 ...
```

注意：`jcmd` 也可支持完成上述功能，请尝试着使用。

3.4 Dynamic Program Profiling

(1) 使用 `JMC` 或者 `VisualVM` 启动你的程序，让程序读入文件 `file1.txt`，

并对生成的图进行各类操作，使用 profiler 发现执行时间热点和内存消耗热点。将 `JMC` 或 `VisualVM` profiler 设定为“自动更新结果”。

(1) 选择 CPU Profiling，动态监控程序的执行时间性能：
- 设定要监控的类为你所开发的类（不包含 `JDK` 提供的公共类）；
- 查看程序中执行的各方法所耗费的时间、所占的比例；
- 着重关注耗费时间居前的各方法，根据你的程序结构来分析这些耗时最多的方法执行是否正常、合理；

(2) 选择 memory profiling，动态监控程序的内存空间性能：
- 查看内存空间中不同类型的对象的个数、所占内存空间大小、所占空间比例等；
- 着重关注内存占用居前的各类型，根据你的程序结构来分析这些耗费内存最多的类型执行是否正常、合理。

3.5 Memory Dump Analysis and Performance Optimization

让程序读取你在 3.2 节所用的某个大文件，执行以下动作：

(2) 在读取文件结束并生成 `CircularOrbit` 对象后，利用 `jmap`、`jconsole`、`jcmd`、`VisualVM` 或 Eclipse 的内存导出(memory dump)功能，导出当前时刻的 `HPROF` 文件（为此，你可能需要让你的程序在此刻停顿下来）。使用以上提及的任意一种工具即可。

(3) 使用 MAT 来分析 3.5 节中产生的内存导出文件`.HPROF`。
- 查看其 Overview、histogram 视图（当前时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）、dominator tree 视图（每个实例之所以未被 GC 的原因，即各实例之间的引用关系，以及与 root 之间的引用路径）、top consumers 视图（程序当前时刻的内存占用热点 hotspot）。
- 查看 leak suspects report，看是否存在可能的内存泄露。

(4) 对热点（执行时间长、占用内存大）的代码区域进行改造优化，重新进行内存导出和 MAT 分析。对比改造前的和改造后的程序，查看其内存消耗方面的改善程度。

(5) 在 MAT 中查看内存占用状态（各类实例的分布情况），使用 OQL 查询语言对其进行以下查询：
- `CircularOrbit` 的所有对象实例；
- 大于特定长度 `n` 的 `String` 对象；
- 大于特定大小的任意类型对象实例；
- `PhysicalObject`（及其子类）的对象实例的数量和总占用内存大小；

- 所有包含元素数量大于 `100` 的 `Collections` 实例；
- …以及你感兴趣的其他查询。

(6) 给程序输入若干条操作指令，在每条指令执行时，利用 `jstack` 或 `jcmd` 导出 java 程序运行时的调用栈（stack trace），观察其中展现出的类和函数调用关系。例如：

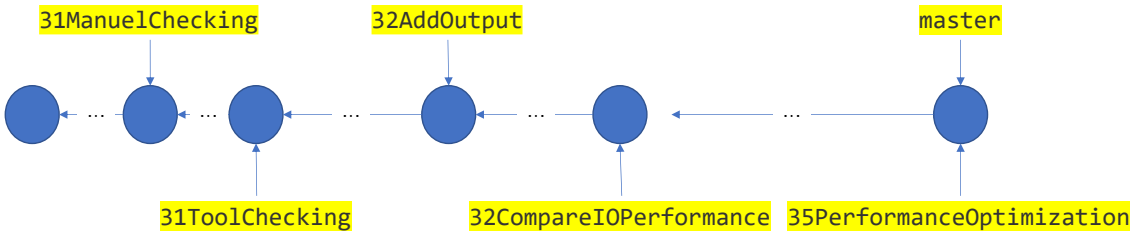
- 增加轨道、删除轨道；
- 向特定轨道上增加物体、从特定轨道上删除物体；
- 判断多轨道系统的合法性；
- …(你可自由选择)

(7) 基于以上 3.2 节、3.3 节、3.4 节的各种工具的观察结果，使用讲义 8.5 节中所介绍的各类方法（但不限于这些方法），对你的代码进行持续的性能优化。例如：

- 在 `AtomicStructure` 应用中，使用 Flyweight 设计模式实现“电子”：原子核周围的所有电子都是等同的，唯一区别在于其所处的 track；
- 基于 Prototype 设计模式的思想改造程序，支持 `clone`；
- 对程序中使用的各种临时性的集合类或其他构造代价高的对象，使用 Singleton 和 Object Pool 设计模式的思想进行改造；
- 对程序中使用的各种简单对象，使用 canonicalization 的思想对其进行改造，降低构造对象的数量；
- 其他更细节的性能改进。

完成后提交代码至 git 仓库，分支名为 35PerformanceOptimization。

注：经过以上各步骤之后，你的 Git 仓库中的 Object Graph 应类似于下图所示：



4 实验报告

针对上述任务，请遵循 CMS 上 Lab5 页面给出的**报告模板**，撰写简明扼要的实验报告。

实验报告的目的是记录你的实验过程，尤其是遇到的困难与解决的途径。不需要长篇累牍，记录关键要点即可，但需确保报告覆盖了本次实验所有开发任务。

注意：

- 实验报告不需要包含所有源代码，请根据上述目的有选择的加入关键源代码，作为辅助说明。
- 请确保报告格式清晰、一致，故请遵循目前模板里设置的字体、字号、行间距、缩进；
- 实验报告提交前，请“目录”上右击，然后选择“更新域”，以确保你的目录标题/页码与正文相对应。
- 实验报告文件可采用 Word 或 PDF 格式，命名规则：Lab5-学号-Report。

5 提交方式

截止日期：第 14 周周日夜间 23:55。截止时间之后通过 Email 等其他渠道提交实验报告和代码，均无效，教师和 TA 不接收，学生本次实验无资格。

源代码：从本地 Git 仓库推送至个人 GitHub 的 Lab5 仓库内。

实验报告：除了随代码仓库（doc）目录提交至 GitHub 之外，还需手工提交至 CMS 实验 5 页面下。

6 评分方式

TA 在第 12 周和第 13 周实验课上现场验收：学生做完实验之后，向 TA 提出验收申请，TA 根据实验要求考核学生的程序运行结果并打分。现场验收并非必需，由学生主动向 TA 提出申请。

Deadline 之后，教师使用持续集成工具对学生在 GitHub 上的代码进行测试。教师和 TA 阅读实验报告，做出相应评分。