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 2018

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 FindBugs）
- 动态代码分析（Java 命令行工具 jstat、jmap、jConsole、VisualVM）
- 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），请分别访问 http://checkstyle.sourceforge.net 和 https://spotbugs.github.io/获取更多帮助信息。（注：SpotBugs 是 FindBugs 的后续版本，你也可以继续使用 FindBugs 完成实验任务。）

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。

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。（实验报告中注明本次 commit 的版本号，前 6 位数字即可，下同）。

(2) 使用 CheckStyle 和 SpotBugs/FindBugs 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。

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 `Graph` 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 `Graph` 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），文件格式应符合 Lab3 中给出的语法规则。完成后，形成新版本提交 git 仓库。

(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。对你的图工厂方法进行改造，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。完成后提交 git 仓库形成新版本。

(3) 通过实验来度量不同实现方式的 I/O 性能差异：

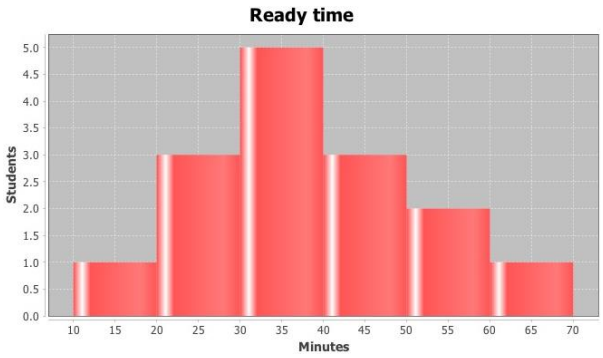
- 从 https://github.com/rainywang/Spring2018_HITCS_SC_Lab5 下载文本文件，你的程序读入文本文件，人工在程序中对该图做一系列操作（自定），进而将内存中的图数据写入文本文件；
- 使用人工代码注入的方式，在你的代码中增加某些代码，用于收集读文件和写文件环节分别消耗的时间，进行对比分析，将结果记录至下表中。（注：在做此对比时，一般仅考虑 I/O 环节，不考虑根据读入的数据构造图对象的时间；如果你的程序处理逻辑是以行为单位从文件中读入数据并加以处理，那么在对比时也可以将读入数据进行语法匹配并转为图结构的时间计算在内，但这会使 I/O 时间不准确。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。）

表 1 不同实现方式的 I/O 性能差异

		file1.txt	file2.txt	file3.txt	file4.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，并把第一列的“I/O 策略 i”替换为具体的 I/O 策略名称。

- （可选，额外计分）使用支持绘图的 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，但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 file1.txt 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、perm）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。

可参考：http://www.cnblogs.com/kongzhongqijing/articles/3625574.html

Option	Displays...
class	Statistics on the behavior of the class loader.
compiler	Statistics of the behavior of the HotSpot Just-in-Time compiler.
gc	Statistics of the behavior of the garbage collected heap.
gccapacity	Statistics of the capacities of the generations and their corresponding spaces.
gccause	Summary of garbage collection statistics (same as **-gcutil**), with the cause of the last and current (if applicable) garbage collection events.
gcnew	Statistics of the behavior of the new generation.
gcnewcapacity	Statistics of the sizes of the new generations and its corresponding spaces.
gcold	Statistics of the behavior of the old and permanent generations.
gcoldcapacity	Statistics of the sizes of the old generation.
gcpermcapacity	Statistics of the sizes of the permanent generation.
gcutil	Summary of garbage collection statistics.
printcompilation	HotSpot compilation method statistics.

(3) 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；

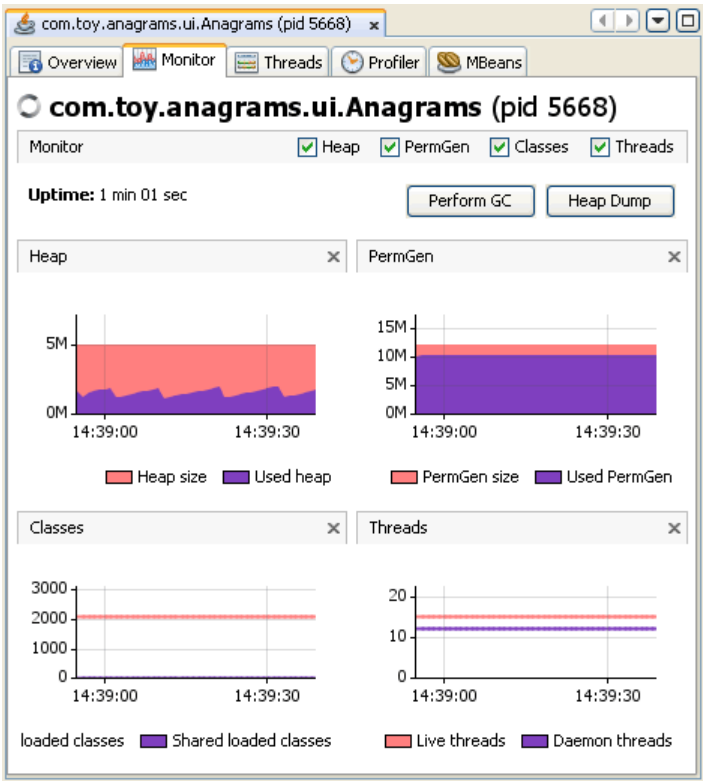
```
$ jmap -heap 29620
Attaching to process ID 29620, please wait...
Debugger attached successfully.
Client compiler detected.
JVM version is 1.6.0-rc-b100

using thread-local object allocation.
Mark Sweep Compact GC

Heap Configuration:
   MinHeapFreeRatio = 40
   MaxHeapFreeRatio = 70
   MaxHeapSize      = 67108864 (64.0MB)
   NewSize          = 2228224 (2.125MB)
   MaxNewSize       = 4294901760 (4095.9375MB)
   OldSize          = 4194304 (4.0MB)
   NewRatio         = 8
   SurvivorRatio    = 8
   PermSize         = 12582912 (12.0MB)
   MaxPermSize      = 67108864 (64.0MB)

Heap Usage:
New Generation (Eden + 1 Survivor Space):
   capacity = 2031616 (1.9375MB)
   used     = 70984 (0.06769561767578125MB)
   free     = 1960632 (1.8698043823242188MB)
   3.49396736391129058 used
Eden Space:
   capacity = 1835008 (1.75MB)
   used     = 36152 (0.03447723388671875MB)
```

(4) （选做，额外计分）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；

```
$ jmap -histo 29620
num   #instances    #bytes  class name
----------------------------------------
  1:      1414     6013016  [I
  2:       793      482888  [B
  3:      2502      334928  <constMethodKlass>
  4:       280      274976  <instanceKlassKlass>
  5:       324      227152  [D
  6:      2502      200896  <methodKlass>
  7:      2094      187496  [C
  8:       280      172248  <constantPoolKlass>
  9:      3767      139000  [Ljava.lang.Object;
 10:       260      122416  <constantPoolCacheKlass>
 11:      3304      112864  <symbolKlass>
 12:       160       72960  java2d.Tools$3
 13:       192       61440  <objArrayKlassKlass>
 14:       219       55640  [F
 15:      2114       50736  java.lang.String
 16:      2079       49896  java.util.HashMap$Entry
 17:       528       48344  [S
 18:      1940       46560  java.util.Hashtable$Entry
 19:       481       46176  java.lang.Class
 20:        92       43424  javax.swing.plaf.metal.MetalScrollButton
... more lines removed here to reduce output...
1118:         1           8  java.util.Hashtable$EmptyIterator
1119:         1           8  sun.java2d.pipe.SolidTextRenderer
Total    61297    10152040
```

(5) （选做，额外计分）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`。）

(6) 使用 `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、Permanent Generation 情况、装载的类实例统计等。



(7) 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。

(8) 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 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:PermSize=5m -XX:MaxPermSize=10m
-XX:+UseSerialGC
```

3.4 Dynamic Program Profiling

使用 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 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 file1.txt，执行以下动作：

(1) 在读取文件结束并生成 `Graph` 对象后，利用 `jmap`、`jconsole`、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。

(2) 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。

(3) 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。

(4) （选做，额外计分）使用 `jhat` 命令行工具分析内存导出文件，并使用 web 浏览器查看内存占用状态（各类实例的分布情况），使用 OQL 查询语言对其进行以下查询：

- `Graph` 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
- `Vertex`（及其每个子类）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 `10` 的 `Collections` 实例；
- …以及你感兴趣的其他查询。

可参考：

https://blog.csdn.net/gtuu0123/article/details/6039474
https://blog.csdn.net/gtuu0123/article/details/6039592

也可在 MAT 中进行基于 OQL 的查询以完成上述功能。

(5) （选做，额外计分）给程序输入若干条图操作指令，在每条指令执行时，利用 `jstack` 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：

- 删除某个节点
- 修改某条边的权值（针对 GraphPoet 应用）
- 增加一条边（针对 NetworkTopology 应用）
- …(你可自由设计)

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。

5 提交方式

截止日期：第 14 周周日（2018 年 6 月 3 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

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。