

6 Лабораторная работа №6

Исследование способов профилирования программного обеспечения

6.1. Цель работы

Исследовать критические по времени выполнения участки программного кода и возможности их устранения. Приобрести практические навыки анализа программ с помощью профайлера VisualVM.

6.2. Постановка задачи

Разработать программу на основе библиотеки классов, реализованной и протестированной в предыдущей работе. Программа должна как можно более полно использовать функциональность класса. При необходимости для наглядности профилирования в методы класса следует искусственно внести задержку выполнения. Выполнить профилирование разработанной программы, выявить функции, на выполнение которых тратится наибольшее время. Модифицировать программу с целью оптимизации времени выполнения. Выполнить повторное профилирование программы, сравнить новые результаты и полученные ранее, сделать выводы.

6.3. Ход выполнения работы

Для выполнения данной лабораторной работы была разработана программа, состоящая из основного класса, представленного в листинге 1, и библиотечных классов, составленных в предыдущих лабораторных работах. Для наглядности профилирования в метод `Matrix#columnPositionWithLongestSeries` была добавлена задержка выполнения с помощью `Thread#sleep` с аргументом 100 миллисекунд.

Листинг 1 — Класс App

```
package org.cory7666.softwaretestingexample;
import java.io.File;
import org.cory7666.softwaretestingexample.task1.FileWithMatrix;
import org.cory7666.softwaretestingexample.task1.Matrix;
public class App
{
    public static void main (String[] args) throws Exception
    {
        if (args.length != 2)
        {
            System.err.println("Usage: program [file-with-matrix]"); System.exit(1);
        }
        else
        {
            final File file = new File(args[0]);
            System.out.printf("Получен файл \"%s\".\n", file.toPath());
            final Matrix matrix = new FileWithMatrix(file).parse();
            System.out.println("Данные из файла считаны.");
            int result = matrix.columnPositionWithLongestSeries();
            System.out.printf("Ответ: %d.\n", result);
            System.out.println("Выполнения програмы завершено.");
            System.exit(0);
        }
    }
}
```

Программа была скомпилирована. Для этой лабораторной работы была составлена матрица, имеющая 20 столбцов и 1 000 000 строчек.

Снимок выполнения программы с задержками представлен на рисунке, без задержек — на рисунке, результат сравнения двух снимков — на рисунке. Каждый из снимков содержит время выполнения функции по реальному времени/по времени процессора, количество вызовов.

Profiler Snapshot

View: Aggregation: Methods

Name	Total Time	Total Time (CPU)	Invocations
main	32 213 ms (100 %)	28 964 ms (100 %)	1
org.cory7666.softwaretestingexample.App.main (String[])	32 213 ms (100 %)	28 964 ms (100 %)	1
org.cory7666.softwaretestingexample.task1.Matrix.columnPositionWithLongestSeries ()	28 454 ms (88,3 %)	26 529 ms (91,6 %)	1
org.cory7666.softwaretestingexample.task1.Series.compareTo (Object)	14 631 ms (45,4 %)	14 749 ms (50,9 %)	19 047 593
Self time	10 021 ms (31,1 %)	9 901 ms (34,2 %)	19 047 593
org.cory7666.softwaretestingexample.task1.Series.compareTo (org.cory7666.softwaretestingexample.task1.Series)	4 609 ms (14,3 %)	4 848 ms (16,7 %)	19 047 593
Self time	7 693 ms (23,9 %)	6 759 ms (23,3 %)	1
org.cory7666.softwaretestingexample.task1.MatrixColumn.getAllPossibleSeries ()	6 130 ms (19 %)	5 020 ms (17,3 %)	20
org.cory7666.softwaretestingexample.task1.FileWithMatrix.parse ()	3 757 ms (11,7 %)	2 432 ms (8,4 %)	1
Self time	1,73 ms (0 %)	1,71 ms (0 %)	1

Name	Self Time (CPU)	Total Time (CPU)	Invocations
org.cory7666.softwaretestingexample.task1.Series.compareTo (Object)	9 901 ms (34,2 %)	14 749 ms (17,8 %)	19 047 593
org.cory7666.softwaretestingexample.task1.Matrix.columnPositionWithLongestSeries ()	6 759 ms (23,3 %)	26 529 ms (32 %)	1
org.cory7666.softwaretestingexample.task1.Series.compareTo (org.cory7666.softwaretestingexample.task1.Series)	4 848 ms (16,7 %)	4 848 ms (5,9 %)	19 047 593
org.cory7666.softwaretestingexample.task1.MatrixColumn.getAllPossibleSeries ()	4 759 ms (16,4 %)	5 020 ms (6,1 %)	20
org.cory7666.softwaretestingexample.task1.FileWithMatrix.parse ()	2 432 ms (8,4 %)	2 432 ms (2,9 %)	1
org.cory7666.softwaretestingexample.task1.Series.increase ()	261 ms (0,9 %)	261 ms (0,3 %)	952 387
org.cory7666.softwaretestingexample.App.main (String[])	1,71 ms (0 %)	28 964 ms (35 %)	1

Рисунок 6.1 — Выполнение программы с задержками


Profiler Snapshot

View: Aggregation: Methods

Name	Total Time	Total Time (CPU)	Invocations
main	29 957 ms (100 %)	27 186 ms (100 %)	1
org.cory7666.softwaretestingexample.App.main (String[])	29 957 ms (100 %)	27 186 ms (100 %)	1
org.cory7666.softwaretestingexample.task1.Matrix.columnPositionWithLongestSeries ()	27 025 ms (90,2 %)	25 641 ms (94,3 %)	1
org.cory7666.softwaretestingexample.task1.Series.compareTo (Object)	13 953 ms (46,6 %)	13 846 ms (50,9 %)	19 047 593
Self time	9 613 ms (32,1 %)	9 233 ms (34 %)	19 047 593
org.cory7666.softwaretestingexample.task1.Series.compareTo (org.cory7666.softwaretestingexample.task1.Series)	4 339 ms (14,5 %)	4 612 ms (17 %)	19 047 593
Self time	7 236 ms (24,2 %)	6 809 ms (25 %)	1
org.cory7666.softwaretestingexample.task1.MatrixColumn.getAllPossibleSeries ()	5 834 ms (19,5 %)	4 985 ms (18,3 %)	20
org.cory7666.softwaretestingexample.task1.FileWithMatrix.parse ()	2 486 ms (8,3 %)	1 543 ms (5,7 %)	1
Self time	445 ms (1,5 %)	1,31 ms (0 %)	1

Name	Self Time (CPU)	Total Time (CPU)	Invocations
org.cory7666.softwaretestingexample.task1.Series.compareTo (Object)	9 233 ms (34 %)	13 846 ms (17,7 %)	19 047 593
org.cory7666.softwaretestingexample.task1.Matrix.columnPositionWithLongestSeries ()	6 809 ms (25 %)	25 641 ms (32,8 %)	1
org.cory7666.softwaretestingexample.task1.MatrixColumn.getAllPossibleSeries ()	4 745 ms (17,5 %)	4 985 ms (6,4 %)	20
org.cory7666.softwaretestingexample.task1.Series.compareTo (org.cory7666.softwaretestingexample.task1.Series)	4 612 ms (17 %)	4 612 ms (5,9 %)	19 047 593
org.cory7666.softwaretestingexample.task1.FileWithMatrix.parse ()	1 543 ms (5,7 %)	1 543 ms (2 %)	1
org.cory7666.softwaretestingexample.task1.Series.increase ()	239 ms (0,9 %)	239 ms (0,3 %)	952 387
org.cory7666.softwaretestingexample.App.main (String[])	1,31 ms (0 %)	27 186 ms (34,8 %)	1

Рисунок 6.2 — Выполнение программы без задержек

Profiler Snapshot				
View:  Aggregation: Methods				
Name	Total Time	Total Time (CPU)	Invocations	
main	-2 256 ms	-1 778 ms	+0	
org.cory7666.softwaretestingexample.App.main (String[])	-2 256 ms	-1 778 ms	+0	
Self time	+444 ms	-0,398 ms	+0	
org.cory7666.softwaretestingexample.task1.FileWithMatrix.parse ()	-1 271 ms	-888 ms	+0	
org.cory7666.softwaretestingexample.task1.Matrix.columnPositionWithLongestSeries ()	-1 429 ms	-888 ms	+0	
org.cory7666.softwaretestingexample.task1.MatrixColumn.getAllPossibleSeries ()	-295 ms	-35,3 ms	+0	
org.cory7666.softwaretestingexample.task1.Series.increase ()	-38,0 ms	-22,1 ms	+0	
Self time	-257 ms	-13,2 ms	+0	
Self time	-456 ms	+49,6 ms	+0	
org.cory7666.softwaretestingexample.task1.Series.compareTo (Object)	-677 ms	-903 ms	+0	
org.cory7666.softwaretestingexample.task1.Series.compareTo (org.cory7666.softwaretestingexample.task1.Series)	-269 ms	-235 ms	+0	
Self time	-407 ms	-667 ms	+0	

Name	Self Time (CPU)	Total Time (CPU)	Invocations	
org.cory7666.softwaretestingexample.task1.Matrix.columnPositionWithLongestSeries ()	+49,6 ms	-888 ms	+0	
org.cory7666.softwaretestingexample.App.main (String[])	-0,398 ms	-1 778 ms	+0	
org.cory7666.softwaretestingexample.task1.MatrixColumn.getAllPossibleSeries ()	-13,2 ms	-35,3 ms	+0	
org.cory7666.softwaretestingexample.task1.Series.increase ()	-22,1 ms	-22,1 ms	+0	
org.cory7666.softwaretestingexample.task1.Series.compareTo (org.cory7666.softwaretestingexample.task1.Series)	-235 ms	-235 ms	+0	
org.cory7666.softwaretestingexample.task1.Series.compareTo (Object)	-667 ms	-903 ms	+0	
org.cory7666.softwaretestingexample.task1.FileWithMatrix.parse ()	-888 ms	-888 ms	+0	

Рисунок 6.3 — Сравнение двух снимков

Профайлер показывает, что горячей точкой программы является функция `Series#compareTo`. Если убрать задержку из функции `Matrix#columnPositionWithLongestSeries`, время выполнения функции уменьшится на 1429мс.

Выводы

При выполнении данной лабораторной работы были получены навыки работы с профайлером VisualVM. Определено, что профайлер следует использовать для определения «узких мест» программы, времени выполнения каждой из функций.

ОТВЕТЫ НА КОНТРОЛЬНЫЕ ВОПРОСЫ

1. В чем заключается процесс профилирования программного обеспечения?

После разработки программного кода, удовлетворяющего спецификации и модульным тестам, возникает задача оптимизации программы по таким параметрам, как производительность, расход памяти и т. п. Процесс анализа характеристик системы называется профилированием и выполняется с помощью специальных программных инструментов – так называемых профилировщиков, или профайлеров (profiler). Программа-профайлер выполняет запуск программы и анализирует характеристики выполнения. Процесс профилирования сводится к декомпозиции программы на отдельные выполняющиеся элементы и измерению времени, затрачиваемого на выполнение каждого элемента, и расходования памяти. В результате профилирования может быть построен граф вызовов, показывающий последовательность вызовов функций, а также определено время выполнения каждой функции и ее вклад в общее время выполнения программы. Анализ этой информации позволяет выявить критические участки программного кода (хотспоты, hot-spot), т. е. «узкие места», на которые следует в первую очередь обратить внимание при оптимизации программы.

2. Какие разновидности «узких мест» программы определяются с помощью профилирования?

Профилирование программы - это процесс анализа ее работы с целью выявления узких мест и других проблем, которые могут снижать производительность и качество работы программы. С помощью профилирования можно определить следующие разновидности узких мест программы:

Узкие места по времени - это части программы, которые занимают большую часть времени выполнения. Профилирование может помочь выявить, какие функции или методы занимают больше всего времени и требуют оптимизации.

Узкие места по памяти - это части программы, которые потребляют большое количество памяти. Профилирование может помочь выявить, какие объекты или структуры данных занимают больше всего памяти и требуют оптимизации.

Проблемы с производительностью ввода-вывода - это проблемы, связанные с чтением и записью данных на жесткий диск или другие устройства ввода-вывода. Профилирование может помочь выявить, какие операции ввода-вывода занимают больше всего времени и требуют оптимизации.

Проблемы с сетевым взаимодействием - это проблемы, связанные с передачей данных по сети. Профилирование может помочь выявить, какие операции сетевого взаимодействия занимают больше всего времени и требуют оптимизации.

Проблемы с многопоточностью - это проблемы, связанные с параллельной обработкой данных в многопоточной среде. Профилирование может помочь выявить, какие потоки занимают больше всего времени и какие синхронизационные механизмы требуют оптимизации.

Проблемы с использованием ресурсов - это проблемы, связанные с неправильным использованием ресурсов, таких как файлы, сокеты, базы данных и т.д. Профилирование может помочь выявить утечки ресурсов и другие проблемы, связанные с их использованием.

3. Что такое граф вызовов программы, для чего он строится?

В вершинах графа, соответствующих каждой из вызываемых функций, показано время выполнения в миллисекундах, а также относительный вклад (в процентах) этой функции в общее время выполнения вызывающей функции. Анализ этих данных позволяет выявить «узкие места» программы.

4. В чем заключается общий порядок профилирования программ с помощью EQATECProfiler?

Перейдем к собственно профилированию разработанной программы. Во вкладке Build программы EQATECProfiler необходимо выбрать папку, в которой

расположена скомпилированная программа (.exe-файл). После загрузки следует выбрать те модули, которые будем профилировать.

Затем нажимаем кнопку «Build». В результате профайлер добавит в нашу программу дополнительные служебные инструкции. После этого нажатие кнопки «Run app» запускает процесс профилирования. По его окончании создается отчет. Список отчетов показан во вкладке «Run». Выбрав нужный (последний по порядку) отчет, переходим на вкладку «View». Здесь в таблице представлены параметры выполнения функций программы. Ниже построен граф вызовов для выбранной функции (на рисунке 2.3 – для метода `MainClass.Main()`). В вершинах графа, соответствующих каждой из вызываемых функций, показано время выполнения в миллисекундах, а также относительный вклад (в процентах) этой функции в общее время выполнения вызывающей функции. Анализ этих данных позволяет выявить «узкие места» программы. Например, в нашем случае видно, что наибольшее время выполнения имеет метод `Rect.getPerimeter()`, а в этом методе, в свою очередь, узким местом является метод `Calc.Multiply()`.

С целью оптимизации времени выполнения программы перепишем метод `Rect.getPerimeter()`, заменив в нем операции умножения операциями сложения:

Скомпилировав приложение, повторив в EQATECProfiler операции «Build», «Run app» и просматривая результаты, видим, что такая модификация кода привела к сокращению общего времени выполнения программы. При этом уменьшился вклад функции `Rect.getPerimeter()` в общее время выполнения. Для сравнения профилей программы до и после оптимизации выбираем на вкладке Run два отчета и нажимаем кнопку «Compare». В результате строится таблица, в которой приводятся как абсолютные показатели старого и нового варианта программы (время выполнения Old Avg и New Avg, количество вызовов Old Calls и New Calls), так и изменения нового варианта по сравнению со старым (относительное изменение времени выполнения функций Speedup, абсолютное изменение времени выполнения Diff Avg, изменение количества вызовов функций Diff Calls).