Practicum – Gegevensstructuren en Algoritmen

Het analyseren en verbeteren van de performantie van Java-applicaties

Inleiding

Optimalisatie van programmacode is een probleem van alle tijden. De allereerste computerprogrammeur, Ada Lovelace, hield zich er al in het jaar 1842 mee bezig:

> 'In almost every computation a great variety of arrangements for the succession of the processes is possible, and various considerations must influence the selection amongst them for the purposes of a Calculating Engine. One essential object is to choose that arrangement which shall tend to reduce to a minimum the time necessary for completing the calculation.'

132 jaar later werd er al aanzienlijk genuanceerder over gedacht door Donald Knuth:

> 'We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified.'

Wat professor Knuth hier bedoelt, is dat in verreweg de meeste tijd waarin we software ontwikkelen we ons niet willen bezighouden met optimalisatie van kleinigheden. Andere belangrijkere overwegingen vereisen al onze volledige aandacht. Optimalisatie moet gelden voor de "critical 3%" van de code. De cruciale vraag is hier natuurlijk: *welke 3% is kritiek?* Zolang die 3% niet geïdentificeerd is, verspillen wij onze tijd met optimalisatie. Het gevolg van nodeloze optimalisatie is dat de leesbaarheid, onderhoudbaarheid en herbruikbaarheid van broncode achteruitgaan: een hoge prijs zonder opbrengst.

Dit practicum heeft als doel je enkele technieken aan te bieden die je kunnen helpen bij het analyseren en verbeteren van de performantie van je Java programma's. Dit onderwerp kan later goed van pas komen bij het project van deze cursus. Op het einde van dit practicum zou je in staat moeten zijn om:

- Java programma's te profilen;
- Enkele veel voorkomende efficiëntieproblemen in Java te identificeren;
- Deze problemen op te lossen a.d.h.v. enkele optimalisaties in de code.

Profiling

Profiling is het geheel van informatie omtrent timings en geheugengebruik verzameld tijdens het uitvoeren van een programma. Er bestaan verschillende tools om Java programma's te

profilen en de kwaliteit en instellingen van de profiler zijn zeer belangrijk om doeltreffend op zoek te kunnen gaan naar kritieke code.

In dit practicum kan je gebruik maken van de profiler die ingebouwd is in de Ultimate edition van IntelliJ. Helaas is deze profiler niet voorzien in de Community edition, maar je kan gratis de Ultimate edition gebruiken met een *educatieve licentie* die je moet aanvragen op `https://www.jetbrains.com/community/education/#students`.

Standaard wordt IntelliJ Ultimate meegeleverd met de volgende profilers:

- De **Java Flight Recorder** (`jfr`) is ingebouwd in OracleJDK en OpenJDK, maar is deprecated en standaard uitgeschakeld in IntelliJ. Dit programma is in principe meer dan een profiler: het kan gebruikt worden als algemene tool om Java applicaties te monitoren en is geconfigureerd om een zo klein mogelijke impact te hebben op de uitvoeringstijd van een programma. Voor korte programma's verzamelt deze profiler standaard te weinig informatie om een duidelijk beeld te krijgen.
- De **IntelliJ Profiler** (vroeger: **Async Profiler**) combineert functionaliteiten van de Java Flight Recorder met het gebruik van de `perf_events` van Unix kernels, bijgevolg werkt deze dus niet op Windows. De standaardconfiguratie van deze profiler is normaal wel goed geschikt om dit practicum te profilen. Om van deze profiler gebruik te maken kan het zijn dat je enkele enkele opties in je kernel moeten aanzetten. IntelliJ zal duidelijk aangeven hoe je dat moet doen.
- Op Windows voorziet IntelliJ ook een **IntelliJ Profiler** (vroeger: **Windows Async Profiler**), maar die bevat wel iets minder functionaliteit. Je kan er bijvoorbeeld niet de geheugenallocaties mee zien.

Profilers verzamelen twee zaken die je achteraf kan analyseren:

- De **CPU samples** zijn steekproeven die genomen werden tijdens de uitvoering van het programma waarbij telkens werd opgeslagen welke methode de JVM was aan het uitvoeren. Je kan dit gebruiken om te achterhalen in welke methoden je programma het meeste tijd spendeert.
- De **Memory Allocations** is hoeveel geheugenallocaties er werden gemaakt. Dit is nuttig om te weten waar je programma het meeste geheugen voor nodig heeft en waar er veel nieuwe objecten worden aangemaakt.

Omdat verschillende profilers andere technieken toepassen om een uitvoering te analyseren, kunnen de resultaten verschillen tussen beide profilers. We raden je aan om de IntelliJ Profiler te gebruiken.

Je dient je er echter wel bewust van te zijn dat het profilen er vaak voor zorgt dat je programma merkbaar trager zal lopen. Dit komt door het toevoegen van instrumentatiecode voor de tijdsmetingen in het programma. Het heeft dus geen zin om absolute doorlooptijden te meten met een profiler.

Opmerking: het gebruik van een profiler zal voor dit practicum niet zoveel meerwaarde hebben aangezien de meeste opgaven – naast de `main()`-methode – maar uit een methode bestaan. Dus zelfs zonder een profiler te gebruiken zullen jullie onmiddellijk weten in welke methode de bottleneck zich bevindt. Het is in dit practicum vooral de bedoeling om jullie te laten kennismaken met de profiler zodat jullie die later kunnen gebruiken voor grotere programma's (bijvoorbeeld tijdens jullie project) waar het niet zo duidelijk is wat de bottleneck is.

Tijdsmetingen

Een vlugge manier om de tijd te meten van een gedeelte (bijvoorbeeld een methode) van je Java programma zonder deze overhead is het aanroepen van een van de volgende methoden op gepaste plaatsen in je code:

`System.currentTimeMillis()`: Dit geeft de huidige tijd met een resolutie van milliseconden. Het is een prima methode, zolang de te meten periode tenminste groter is dan een tiende seconde.

`System.nanoTime()`: Dit geeft de waarde terug van een hardware klokregister met een resolutie van nanoseconden. Deze waarde hoeft evenwel niet gerelateerd te zijn aan de werkelijke tijd. De teruggegeven waarde kan zelfs *negatief* zijn! De enige betekenisvolle manier om deze waarde te gebruiken is als een verschil met de waarde van een andere aanroep van dezelfde functie.

De benodigde tijd voor een for-lus is bijvoorbeeld als volgt te meten:

```
long start = System.currentTimeMillis();
for (int i = 0; i < somethingBig; ++i) {
    // Get some work done ...
}
long end = System.currentTimeMillis();
long elapsed = end - start;
System.out.printf("Time elapsed: %d ms\n", elapsed);
```

Oefening 1: Het samenstellen van een string in Java

Tracht te begrijpen wat er precies gebeurt in het programma StringCat.java. Welke tijdscomplexiteit in de lengte van de string zou je verwachten? Voer tijdsmetingen uit om na te gaan of dit effectief het geval is. Tracht je vaststellingen te verklaren. Wat gebeurt er precies zodat de vastgestelde tijdscomplexiteit niet deze is die je verwacht?

Je zal merken dat IntelliJ slim genoeg is om het probleem te herkennen en zelfs een voorstel kan doen hoe je dit kan oplossen. Pas deze suggestie toe en voer vervolgens opnieuw tijdsmetingen uit. Is de tijdscomplexiteit nu wel zoals je zou verwachten?

Oefening 2: Het markeren van bezochte elementen

In veel toepassingen is het belangrijk om bezochte elementen te markeren, zodat je niet meerdere keren hetzelfde element bezoekt. Beschouw voor deze oefening het Java-bestand RandomVisit.java dat gebruikmaakt van een dergelijke markering.

In methode `visitElements` simuleren we dat we een driedimensionale structuur hebben. In de buitenste for-lus willen we een zekere berekening $iterations$ keer uitvoeren. Deze berekening bestaat uit het $internalIterations$ keer bezoeken van een willekeurig punt van onze driedimensionale structuur voor een $punt$-berekening. Echter, als het punt al bezocht is dan

voeren wij deze $punt$-berekening niet nog eens uit. Het totaal aantal verschillende bezochte punten, wat gelijk is aan het aantal gesimuleerde $punt$-berekeningen, wordt uiteindelijk teruggegeven.

Om onduidelijke redenen heeft deze berekening erg veel tijd nodig. Onderzoek waarom. Wellicht zijn er verschillende optimalisaties nodig. Onderzoek welke optimalisaties succesvol zijn door $verschillende$ versies van methode `visitElements` toe te voegen. Geef telkens voor elke versie de benodigde rekentijd weer. Door herhaaldelijk verder optimaliseren in nieuwe versies moet uiteindelijk een factor achthonderd gewonnen kunnen worden.

Oefening 3: Elementen opvragen uit een `Collection`

In RandomSum.java wordt eerst een `Collection` opgevuld, waarna er verschillende elementen uit die collection opgezocht worden. Voeg tijdsmetingen toe en meet hoeveel tijd er wordt besteed aan het opvullen van de `Collection` en hoeveel aan het opvragen van de elementen. Hoe kan dit op een veel efficiëntere manier gebeuren?

Oefening 4: De fractal generator

In deze oefening wordt gebruik gemaakt van een fractal generator. Hierbij worden de volgende bronbestanden gebruikt:

- ComplexNumber.java: Stelt een complex getal voor.
- Mandelbrot.java: Gebruikt `ComplexNumbers` om Mandelbrot fractals te berekenen.
- MandelView.java: Beeldt de `Mandelbrot` fractals af; dit is de klasse die je moet uitvoeren om de fractal af te beelden en performantiedata te verzamelen.

Compileer alle gegeven Java bestanden en voer daarna `MandelView` uit. Bekijk vervolgens de Java code van de klasses `ComplexNumber` en `Mandelbrot`. Het is niet noodzakelijk om de werking van het MandelBrot algoritme te verstaan. Let echter wel op de structuur van het programma. Ga bij jezelf na welke optimalisaties je zou toevoegen aan het programma, maar implementeer ze niet.

Voeg nu de opdracht `System.exit(0);` toe aan het einde van de `paint()`-methode in `MandelView.java` zodat het programma afgesloten wordt nadat de Mandelbrotfractaal voor de eerste maal op het venster getekend wordt. Hercompileer het programma en gebruik de IntelliJ Ultimate profiler.

Probeer uit te vissen op welke plaats(en) van het programma het grootste gedeelte van de tijd gebruikt wordt. Tracht vervolgens het probleem te lokaliseren in de broncode. Hierbij kun je ook tijdsmetingen gebruiken. Laat het gedeelte van de code dat je wenst te optimaliseren verschillende malen uitvoeren in een lus, zodat je een betrouwbaar gemiddelde krijgt. Noteer deze tijdsmetingen en tracht nu dit gedeelte van de code te optimaliseren.

Telkens je een optimalisatie doorvoert, voer je je tijdsmetingen opnieuw uit en kijk je of ze het gewenste effect hebben.

Opmerking: Deze oefening is gebaseerd op een oefening uit "Performance, Profiling, and Tuning", MIT 2003.