Performance Java Streams vs For (plus Go)

Test Einführung

Das Java Stream API bietet seit Java Version 8 mächtige Möglichkeiten zur Durchführung von Operationen auf Arrays und Listen.Wenn früher Arrays und Listen mittels der klassichen for-Schleife verarbeitet wurden erfolgt dies in der Regel heute via Java Streams.Denn Java Stream haben einige Vorteile:Streams sind deklarativer und eleganterStarkes Method ChainingStreams sind kurz und bündigStreams sind funktional (Lambda)Streams bieten die parallele AusführungStreams fördern die lose Kopplung...Es gibt aber auch gewichtige Nachteile:Leistungintensiver und damit schlechtere PerformanceWeniger vertraut als die klassische SchleifeTeilweise schwer verständlich -> Code schwerer lesbar da eher kompaktSchwieriger zu debuggen...Das A und O einer Anwendung ist deren Performance. Wir betrachten in diesem Blog die Performance von Java Streams gegenüber dem klassischen For Loop.

Test Context

Die Performance Analyse basiert auf einem einfachen Fall. Wir verwenden ein Array mit pro Testfall unterschiedlichen Längen, welche über Command Line Parameter variert werden kann. Das Array vom Typ int wird mit aufsteigenden Zahlen 0...N initialisiert. Diese Initialisierung ist nicht Teil der Performacnce Analyse. Alsdann bestimmen wir die Summe aller geraden Zahlen. Diese Operation wiederholen wir für den Test 1000000 mal und messen die Zeit in Millisekunden. Diesen Algorithmus lösen wir mit Java Streams wie folgt über die Methode mapStreamIncrementDemo:public static void mapStreamIncrementDemo(int cycles, int size) {
  int[] numbers = new int[size];
  Arrays.setAll(numbers, j -> j);
  long start = System.currentTimeMillis();
  for (int i = 0; i < cycles; i++) {
 Arrays.stream(numbers).filter(v -> v % 2 == 0).reduce(0, (a, b) -> a + b);
  }
  Integer evenSum = Arrays.stream(numbers).filter(v -> v % 2 == 0).reduce(0, (a, b) -> a + b);
 long end = System.currentTimeMillis();
  System.out.println("map stream, array length = " + numbers.length + ", evenSum = " + evenSum + ", time = " + (end - start) + "ms");
}Den gleichen Algorithmus programmieren wir mit dem klassischen for-Loop über die Methode mapForIncrementDemo:public static void mapForIncrementDemo(int cycles, int size) {
  int[] numbers = new int[size];
  Arrays.setAll(numbers, j -> j);
 long start = System.currentTimeMillis();
  for (int i = 0; i < cycles; i++) {
  int evenSum = 0;
  for (int j = 0; j < numbers.length; j++) {
  if (numbers[j] % 2 == 0) {
 evenSum += numbers[j];
  }
  }
  }
  long end = System.currentTimeMillis();
  int evenSum = 0;
  for (int j = 0; j < numbers.length; j++) {
  numbers[j] = j;
  if (numbers[j] % 2 == 0) {
  evenSum += numbers[j];
 }
  }
  System.out.println("map increment, array length = " + numbers.length + ", evenSum = " + evenSum + ", time = " + (end - start) + "ms");
}Die Zeit messen wir über die Methode System.currentTimeMillis() jeweils am Anfang und am Ende. Default arbeiten wir mit einem Array der Länge 100. Das folgenden Listing zeigt die Methode main:public static void main(String[] args) {
  int cycles = 1000000;
  int size = 100;
  if (args.length > 0) {
  size = Integer.parseInt(args[0]);
  }
 mapStreamIncrementDemo(cycles, size);
  mapForIncrementDemo(cycles, size);
}Der gesamte Testprogramm Java Code findet man unter dem Link JavaArrayStreamVsFor.java.Das ausführbare Testprogramm als Java Archive Version 13 findet man unter dem Link JavaArrayStreamVsFor.jar.

Test Analyse

Wir analysieren die beiden Varianten mit verschiedenen Bereichen der Array Länge [1-100], [1-1000] und [1-10000]. Unser Windows System arbeitet mit einem i7 Prozessor mit 16GB RAM, 4 CPU und der Java Version 13.

Test Analyse [1-100]

Hier analysieren wir beide Varianten im Bereich 1-100 in 5 Schritten.mit dem entsprechenden Chart Diagramm:Klar schneller ist die Variante mit dem klassischen for Loop.

Test Analyse [1-1000]

Hier analysieren wir beide Varianten im Bereich 1-1000 in 7 Schritten.mit dem entsprechenden Chart Diagramm:Der klassische for Loop ist noch marginal schneller und beide praktisch gleichauf.

Test Analyse [1-10000]

Hier analysieren wir beide Varianten im Bereich 1-10000 in 9 Schritten.mit dem entsprechenden Chart Diagramm:Die Variante Stream ist leicht schneller.

Fazit

Java Streams wirken sich bei Aggregaten ab Länge 1000 vorteilhaft aus. Bei Längen im Bereich < 1000 sind der klassische for Loop vorzuziehen.Zu beachten ist auch, dass Java Stream nur einmalig anwendbar sind. Die mehrfache Filterung von Daten bedingt pro Filter einen Stream. In diesem Fall sind klassiche for Loops sicher noch effizienter.Java Stream sind etwas moderner, aber es gibt keinen Grund, dass man nun alles mit Streams anstelle dem alten for Loop programmieren muss.Beide Konzepte haben ihre Berechtigung und sollten nicht gegeneinander antreten sondern sich ergänzen.

Vergleich mit Go

Zu guter letzt vergleichen wir beide Java Lösungen mit der go for Loop Variante.package main
import (
  "fmt"
  "os"
  "strconv"
  "time"
)
func main() {
  cycles := 1000000
  size := 10
  if len(os.Args) > 1 {
    size, _ = strconv.Atoi(os.Args[1])
  }
  numbers := make([]int, size)
  for i := 0; i < size; i++ {
    numbers[i] = i
  }
  start := makeTimestamp()
    for i := 0; i < cycles; i++ {
      evenSum := 0
      for j := 0; j < size; j++ {
        if numbers[j]%2 == 0 {
          evenSum += numbers[j]
        }
      }
    }
    evenSum := 0
    for j := 0; j < size; j++ {
    if numbers[j]%2 == 0 {
      evenSum += numbers[j]
    }
  }
  end := makeTimestamp()
  fmt.Printf("\nsize = %d, evenSum= %d, time = %d ms\n", size, evenSum, (end - start))
}

func makeTimestamp() int64 {
  return time.Now().UnixNano() / int64(time.Millisecond)
}Die gemessenen Zeiten sprechen für go:mit dem entsprechenden Chart Diagramm:go ist damit praktisch doppelt so schnell wie die gleiche Lösung in Java.

Feedback

War dieser Blog für Sie wertvoll. Wir danken für jede Anregung und Feedback

Kontakt

Simtech AG
Finkenweg 23
3110 Münsingen
Schweiz

Impressum

Das Copyright für sämtliche Inhalte dieser Website liegt bei Simtech AG, Schweiz.
Beachten Sie auch unsere Hinweise zum Urheberrecht, Datenschutz und Haftungsauschluss.
Jeder Hinweis auf Fehler nehmen wir gerne entgegen.

Copyright

2024 Simtech AG, All rights reserved, Powered by stack.ch written in Golang by Daniel Schmutz

https://www.simtech-ag.ch/unixnano