SpaceFox

Zeste de savoir

Java : presque 9000 requêtes par seconde avec 8 Mo de RAM

1er juin 2022

Table des matières

Introduction

Vous l'avez peut-être remarqué: mon avatar est aléatoire.

L'implémentation actuelle est faite avec trois lignes de PHP, ce qui m'ennuie un peu parce que c'est le seul outil qui a encore besoin de PHP sur mon serveur. Je me suis donc demandé: est-ce que je pourrais réimplémenter ça en Java? Après tout, la partie dynamique est complètement triviale: c'est une URL qui réponds en HTTP et, quoi qu'on lui demande, renvoie un code retour `HTTP 302` vers l'une des 21 URLs d'images de renard disponibles.

On colle le serveur dynamique derrière un Nginx pour faire proxy, HTTPS et serveur d'images, et ça roule.

1. Un peu de code

```java
package fr.spacefox.avatar;

import com.sun.net.httpserver.HttpServer;
import java.io.IOException;
import java.net.InetSocketAddress;
import java.util.Random;

public class AvatarHttpServer {

    private final Random random = new Random();
    private final int port;
    private final int imgCount;

    public AvatarHttpServer(final int port, final int imgCount) {
        this.port = port;
        this.imgCount = imgCount;
```

```
17      }
18
19      public void run() throws IOException {
20          var server = HttpServer.create(new
                InetSocketAddress(port), 0);
21          server.createContext("/", exchange -> {
22              exchange.getResponseHeaders().add(
23                      "Location",
24                      "https://avatar.spacefox.fr/Renard-" +
                            (random.nextInt(imgCount) + 1) + ".png");
25              exchange.sendResponseHeaders(302, -1);
26          });
27          server.start();
28      }
29
30      public static void main(String[] args) throws IOException {
31          new AvatarHttpServer(Integer.parseInt(args[0]),
                Integer.parseInt(args[1])).run();
32      }
33  }
```

C'est du Java pur, sans aucune dépendance à aucune bibliothèque externe

33 lignes avec tout le *boilerplate*, c'est dix fois plus que l'implémentation actuelle. J'aurais pu gratter un peu en ne permettant pas la configuration du port ou du nombre d'images (la version actuelle ne le fait pas) mais ça ne change pas l'ordre de grandeur.

D'un côté, ça fait pas mal de blabla pour un fonctionnement aussi simple; de l'autre, ça reste très raisonnable par rapport à ce qu'a pu être Java: pas besoin d'aller jouer manuellement avec des *socket* ou d'autres opérations acrobatiques.

Et surtout: c'est du Java pur, sans la moindre dépendance, qui va tourner directement sur n'importe quelle JVM (17 ou supérieure) – y compris les dérivées d'IBM J9. Le fichier `.jar` généré et exécutable (via `java -jar`) fait 2.02 ko.

2. Les performances

La question que je me suis posé ensuite, c'est: puisque le truc est tout petit, ça doit consommer presque rien en mémoire. D'accord, mais «presque rien» avec Java, c'est quoi? Pour quelles performances? Essayons donc.

Je définis le tas à 8 Mo avec `-Xmx8m` et limite les piles à 256 ko avec `-Xss256k` (par défaut c'est 1 Mo, clairement inutiles ici). Et je teste:

Débit en sortie

```
spacefox@shub-niggurath:~$ java -Xmx8m -Xss256k -jar AvatarServer-1.0-SNAPSHOT.jar 6666 21 &
[1] 654741
spacefox@shub-niggurath:~$ ab -n 100000 -c 10 http://localhost:6666/
This is ApacheBench, Version 2.3 <$Revision: 1879490 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient)
Completed 10000 requests
Completed 20000 requests
Completed 30000 requests
Completed 40000 requests
Completed 50000 requests
Completed 60000 requests
Completed 70000 requests
Completed 80000 requests
Completed 90000 requests
Completed 100000 requests
Finished 100000 requests


Server Software:
Server Hostname:        localhost
Server Port:            6666

Document Path:          /
Document Length:        0 bytes

Concurrency Level:      10
Time taken for tests:   11.355 seconds
Complete requests:      100000
Failed requests:        0
Non-2xx responses:      100000
Total transferred:      16157218 bytes
HTML transferred:       0 bytes
Requests per second:    8806.68 [#/sec] (mean)
Time per request:       1.136 [ms] (mean)
Time per request:       0.114 [ms] (mean, across all concurrent requests)
Transfer rate:          1389.56 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.2      0      11
```

```
Processing:     0    1   0.8      1      66
Waiting:        0    1   0.8      1      65
Total:          0    1   0.8      1      66

Percentage of the requests served within a certain time (ms)
  50%      1
  66%      1
  75%      1
  80%      1
  90%      2
  95%      2
  98%      4
  99%      4
 100%     66 (longest request)
```

8806.68 requêtes par seconde en moyenne en local. C'est pas mal, surtout que le code complètement mono-thread et que le serveur sur lequel je lance ça tourne sur des Intel® Xeon® CPU E5–2690 v3 @ 2.60GHz, donc des processeurs assez lents – je suis plutôt à 17000 sur mon ordinateur.

Je dis *«c'est pas mal»* mais je devrais plutôt écrire *«c'est énorme»*: le même test, sur le même serveur, sur un fichier statique hébergé sur cette même machine, via nginx et HTTPS, plafonne plutôt vers 450 requêtes par seconde. Ceci implique que le facteur limitant en conditions réelles sera le proxy (et tout ce qu'il doit gérer) plus que ce programme.

Mais surtout, les performances sont limitées par le CPU *et* par le passage du *garbage collector*: le simple fait d'assigner 16 Mo de mémoire au tas fait monter les performances à près de 10000 requêtes par seconde – au-delà, l'amélioration n'est plus significative.

Le choix du *garbage collector* est important: ici le test est fait avec G1GC, celui par défaut dans l'OpenJDK 17. Mais si je passe à ZGC, un *garbage collector* alternatif qui a ses avantages mais n'est clairement pas conçu pour ce genre de charge, les performance sont diminuées de moitié.

La vérité sur la consommation mémoire

Vérifier la consommation mémoire réelle d'une application n'est pas quelque chose de trivial. Pour commencer, beaucoup d'outils affichent la mémoire *demandée* et pas celle *réellement utilisée*, ce qui fausse les chiffres à la hausse.

Or, le tas susmentionné (et généralement la première valeur que l'on modifie dans les programmes Java avec la directive `-Xmx`) n'est pas la seule zone mémoire utilisée par la JVM, il y en a d'autres.

Avec les outils d'introspection Java (dont je ne vous colle pas la sortie, de toutes façon illisible), j'estime la consommation réelle de cet outil à entre 30 et 40 Mo de mémoire – qui ne bouge pas, même après avoir servi des dizaines de millions de requêtes.

3. Alors, Java c'est lourd et lent et verbeux ?

Ben… oui et non.

Le simple fait d'utiliser une JVM impose de se réserver quelques dizaines de Mo et plusieurs *threads* invisibles (notamment à cause du Garbage Collector). Si ça pouvait être énorme à l'époque où les ordinateurs avaient quelques centaines de Mo de RAM au total, ça n'est plus un vrai problème aujourd'hui, surtout que le code métier prends très vite beaucoup plus de mémoire que cette consommation de base. De plus, beaucoup de programmes Java dans la nature consomment trop par rapport à ce qu'ils pourraient consommer pour deux raisons principales: d'une part la confusion entre *garbage collector* et *mémoire magique* qui crée beaucoup de fuites de mémoire, d'autre part à cause de réglages désastreux par défaut. Par exemple, j'ai encore croisé cette semaine un outil qui conseille de paramétrer 2 Go de tas là où 250 Mo suffisent largement…

Les performances sur ce genre d'exercice sont suffisantes pour que le processeur ne soit jamais un problème. En fait, cette assertion est généralisable: les performances des compilateurs *just in time* intégrés aux JVM (OpenJDK et dérivées, J9 et dérivées) sont excellentes, tant qu'on ne tape pas dans les programmes extrêmement calculatoires. Les goulets d'étranglement d'un programme Java correctement réalisé sont rarement au niveau du CPU (dans mon test c'est le cas, mais le test n'est pas représentatif d'un usage réel: le proxy ou la connexion satureront avant le CPU en usage réel).

Enfin, Java a fait d'énormes efforts en verbosité, même s'il reste des axes d'amélioration (j'aurais pu éviter le constructeur au prix d'un import – qui sont toujours masqués – et d'une annotation avec Lombok). Kotlin fait mieux de ce côté, mais impose de distribuer les classes de *runtime*, ce qui n'est clairement pas intéressant dans ce cas.

De plus, tout ça c'est avec un programme écrit en 10 minutes qui ne nécessite aucune dépendance (autre qu'une JVM).

Quoi? Vous saviez déjà tout ça? Pourtant, c'est encore des critiques qu'on entends très régulièrement.

4. Et dans votre langage préféré ?

Je serais curieux de savoir ce que peut donner l'exercice dans votre langage préféré. N'hésitez pas à le réaliser et à partager vos résultats!