TP4 Exercices d'optimisation

PENG Hanyuan & YAN Wenli

Exercice1

1.Changement de type de données

Problème et modification

Dans Exercice 1, on a trouvé qu'il mélange de différents types de données, `int` , `float` qui sont primitifs, et `Integer` et `Float` qui sont les classes de wrapper d'objet correspondante. Ceci provoque des occurrences fréquentes de ces phénomènes: `Boxing` et `Unboxing` lors de l'exécution de l'opération.

Donc pour éviter d'utiliser de différents types de données et augmenter la performance pendant le calcul, on a changé tous les `Integer` et `Float` aux types primitifs `ìnt` et `float` . Comme indiqué ci-dessous:

```
// my code
private static final float[][] MATRIX_B = {
        {1 / 42f, 1 / 42f, 2 / 42f, 2 / 42f, 2 / 42f, 1 / 42f, 1 / 42f},
        {1 / 42f, 2 / 42f, 3 / 42f, 4 / 42f, 3 / 42f, 2 / 42f, 1 / 42f},
        {2 / 42f, 3 / 42f, 4 / 42f, 5 / 42f, 4 / 42f, 3 / 42f, 2 / 42f},
        {2 / 42f, 4 / 42f, 5 / 42f, 8 / 42f, 5 / 42f, 4 / 42f, 2 / 42f},
        {2 / 42f, 3 / 42f, 4 / 42f, 5 / 42f, 4 / 42f, 3 / 42f, 2 / 42f},
        {1 / 42f, 2 / 42f, 3 / 42f, 4 / 42f, 3 / 42f, 2 / 42f, 1 / 42f},
        {1 / 42f, 1 / 42f, 2 / 42f, 2 / 42f, 2 / 42f, 1 / 42f, 1 / 42f}
};
```

```
public static float[][] my_multiply2(final int[][] matrix) {
    final float[][] result = new float[7][7];

    for (int i = 0; i < 7; i = i + 1)
        for (int j = 0; j < 7; j = j + 1) {
            float currentValue = 0F;

            for (int k = 0; k < 7; k = k + 1) {
                currentValue = currentValue + matrix[i][k] * MATRIX_B[k][j];
            }
            result[i][j] = currentValue;
        }
    return result;
}
```

Boxing, Unboxing

Le `Boxing` est la conversion automatique effectuée par le compilateur Java entre les types primitifs et les classes de wrapper d'objets correspondantes, par exemple, la conversion d'un `int` en un `Integer` , d'un `double` en un `Double` , etc.

Si la conversion va dans l'autre sens, cela s'appelle `Unboxing` .

Impact sur la performance

Quand le type `int` et `Integer` sont utilisés dans un même calcul (c'est-à-dire contient des opérations arithmétiques), en fait, ils comparent par la valeur, donc le processus déclenche le `Boxing` et `Unboxing` qui ajoute la complexité du processus et réduit les performances. Donc c'est mieux d'utiliser des mêmes types de données. Et en plus, pendant le calcul, c'est l'opération sur des valeurs, c'est mieux d'utiliser des types primitifs.

2. Déclaration de variable dans la boucle

Problème et modification

```
for (Integer i = 0; i < 7; i = i + 1) {
    for (Integer j = 0; j < 7; j = j + 1) {
        Float currentValue = 0F;
```

On a trouvé que la éclaration de variable `float currentValue` est faite réculierement dans les deux boucles for. On pense qu'ici on a pas besoin de faire la déclaration chaque fois car cette variable n'est pas une variable temporaire.

Donc on extrait la déclaration dehors les boucles for.

```
public static float[][] my_multiply2_out(final int[][] matrix) {
        final float[][] result = new float[7][7];
        float currentValue;
        for (int i = 0; i < 7; i = i + 1)
            for (int j = 0; j < 7; j = j + 1) {
                currentValue = 0F;

                for (int k = 0; k < 7; k = k + 1) {
                    currentValue = currentValue + matrix[i][k] * MATRIX_B[k][j];
                }

                result[i][j] = currentValue;
            }

        return result;
    }
```

Impact sur la performance

En fait, ça augmente pas beaucoup la performance, mais c'est toujours mieux.

Benchmark pour Ex1

```
Benchmark                          (param)  Mode  Cnt     Score     Error  Units
BenchMark.testFloat2float              N/A  avgt    6     1.802 ±   0.648  ms/op
BenchMark.testFloat2float_int          N/A  avgt    6     0.507 ±   0.107  ms/op
BenchMark.testFloat2float_out          N/A  avgt    6     0.534 ±   0.174  ms/op
BenchMark.testOrigin                   N/A  avgt    6     3.208 ±   1.522  ms/op
```

Exercice2 Fibonacci

Code original:

```
public static int fibonacci(final Integer i) {
    if (i < 0) {
        throw new IllegalArgumentException("Invalid input value");
    }
    return (i < 3) ? i : fibonacci(i - 1) + fibonacci(i - 2);
```

```
}
```

1. Changement de type de données

Problème et modification

```
public static int my_fibonacci(final int i) {
    if (i < 0) {
        throw new IllegalArgumentException("Invalid input value");
    }
    return (i < 3) ? i : fibonacci(i - 1) + fibonacci(i - 2);
}
```

Comme avant, il existe des problèmes sur les types de données. Dans cette méthode, le type de retour est `int` mais le type de paramètre est `Integer`. Cela provoque une augmentation du coût en temps des dépenses en `Boxing` et `Unboxing`.

2. Mot clé `final`

```
public static final int my_fibonacci_final(final int i) {
    if (i < 0) {
        throw new IllegalArgumentException("Invalid input value");
    }
    return (i < 3) ? i : fibonacci(i - 1) + fibonacci(i - 2);
}
```

Modification

On a ajouté le mot clé `final` sur la méthode.

Impact sur la performance

3. Utilisation de la méthode récursive ou boucle for

Problème et modification

La méthode récursive est une manière d'écrire succincte, mais ce type d'imbrication conduit à plusieurs appels de la même partie. Si on ne stocke pas les résultats précédents, cela entraînera beaucoup de doubles comptages.

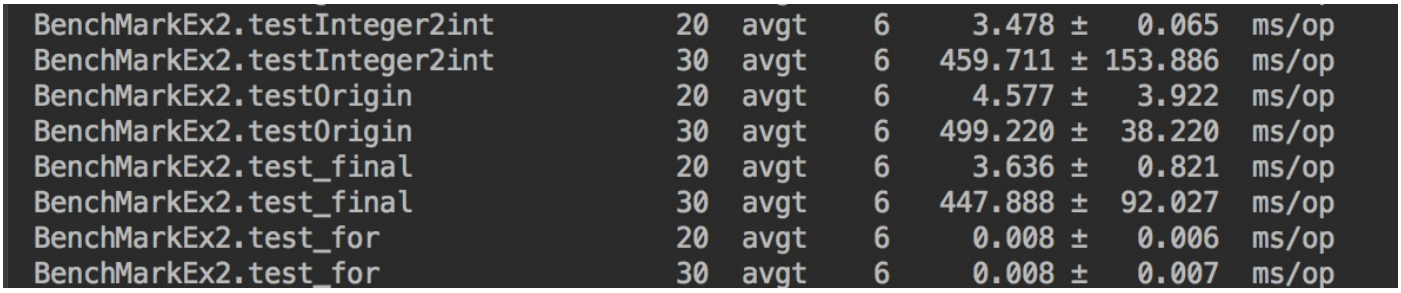
Donc on a modifié l'algorithme en utilisant la boucle `for` et en stockant les résultats précédents.

```
public static int my_fibonacci_for(final int i) {
    if (i < 0) {
        throw new IllegalArgumentException("Invalid input value");
    }
    int result = 1;
    int[] temp = new int[100];
    temp[0] = 1;
    temp[1] = 1;
    for (int j = 2; j <= i; j++) {
        temp[j] = temp[j - 2] + temp[j - 1];
    }
    return temp[i];
}
```

Impact sur la performance

Après la modification, la performance est significativement améliorée.

Benchmark pour Ex2



```
BenchMarkEx2.testInteger2int          20  avgt    6     3.478 ±   0.065  ms/op
BenchMarkEx2.testInteger2int          30  avgt    6   459.711 ± 153.886  ms/op
BenchMarkEx2.testOrigin               20  avgt    6     4.577 ±   3.922  ms/op
BenchMarkEx2.testOrigin               30  avgt    6   499.220 ±  38.220  ms/op
BenchMarkEx2.test_final               20  avgt    6     3.636 ±   0.821  ms/op
BenchMarkEx2.test_final               30  avgt    6   447.888 ±  92.027  ms/op
BenchMarkEx2.test_for                 20  avgt    6     0.008 ±   0.006  ms/op
BenchMarkEx2.test_for                 30  avgt    6     0.008 ±   0.007  ms/op
```

Exercice3a/3b

Exercice 3a et 3b sont similaires. On parle d'abord Exercice 3a.

1. Changement de type de données

```
//ex3a_int
public int exercice3a_int(final Integer nbThreads, final Integer nbIterationByThread)
```

```
//ex3a_int
for (int i = 0; i < nbThreads; i++) {
    final Future future = service.submit(() -> iterate(nbIterationByThread));
    futures.add(future);
}
```

Dans cettes méthode, on a modifié aussi (Comme ex1)le type de donnée de retour `count` et de l'itérateur `i` dans la boucle `for`. Ça augmente un peu la performance car `int` est le type primitif, mais `Integer` est la classe de wrapper d'objet correspondante de `int`.

2. Utilisation de mots-clés `synchronised`

```
//3a_synchronised_variable
private void incrementCounter() {
    synchronized (MUTEX) {
        count++;
    }
}
```

Dans cette méthode, on a utilisé qu'une fois le mot clé `synchronized`, sur le block intérieur. La partie synchronisation du programme Java affecte beaucoup l'efficacité de l'exécution. En général, une méthode crée d'abord des variables locales, puis effectue certaines opérations sur ces variables, telles que les calculations, l'affichage, etc. Et le plus de code couvert par la synchronisation, l'impact sur l'efficacité est plus grave. Donc, nous essayons généralement de réduire l'influence de la synchronisation en utilisant ce mot clé `synchronized` sur le block code qu'on veut synchroniser entre les threads.

Mais en fait dans ce cas là si on met le mot clé `synchronised` sur la méthode, la performance sera augmented aussi, car ici il y a qu'une ligne de code. L'impact est pareil.

3. Détermination de la fin d'exécution de tous les threads

```
//ex3a excute(), isTerminated()
public Integer exercice3a_excute(final Integer nbThreads, final Integer
nbIterationByThread) throws ExecutionException, InterruptedException {
    final ExecutorService service = Executors.newFixedThreadPool(nbThreads);

    for (Integer i = 0; i < nbThreads; i++) {
        service.execute(()->iterate(nbIterationByThread));
    }

    service.shutdown();
    while(true){
        if(service.isTerminated()){
            return count;
        }
    }
}
```

Dans le code source, pour assurer que le processus retourne la valeur après tous les threads sont terminés, il utilise le `futur.get()`. On a essayé de changer la façon à déterminer la fin d'exécution de tous les threads en utilisant `excute()` et `isTerminated()`. Cette ligne fait la même chose.

Ça n'augmente pas de performance car en fait il faut attendre et vérifier que tous les threads sont terminés en utilisant n'importe quelle méthode.

```
//ex3a nolist_futures
public Integer exercice3a_nolistfuture(final Integer nbThreads, final Integer
nbIterationByThread) throws ExecutionException, InterruptedException {
    final ExecutorService service = Executors.newFixedThreadPool(nbThreads);

    for (Integer i = 0; i < nbThreads; i++) {
        final Future future = service.submit(() -> iterate_1(nbIterationByThread));
        future.get();
    }

    // Wait for it...
//        for (final Future<Runnable> future : futures) {
//            future.get();
//        }
    service.shutdown();
    return count;
}
```

Cette façon augmente la performance de plus de 4 fois. En fait la méthode `service.shutdown()` peut faire terminer tous les threads à la fin des tâches, exécute des tâches précédemment soumise, mais n'accepte pas de nouvelles tâches. S'il est déjà fermé, l'appel n'a pas d'autre effet.

Benchmark pour Ex3a

```
Benchmark                   Mode  Cnt     Score      Error  Units
BenchMarkEx3a.testOrigin    avgt    4  4482.202 ±  953.966  ms/op
BenchMarkEx3a.test_1        avgt    4  2645.122 ± 2312.131  ms/op
BenchMarkEx3a.test_execute  avgt    4  4721.106 ± 3509.067  ms/op
BenchMarkEx3a.test_int      avgt    4   286.924 ±   16.803  ms/op
BenchMarkEx3a.test_nolist   avgt    4  1082.543 ±   21.237  ms/op
```

Exercice 3b

Exercice 3b est similaire à exercice 3a, on choisit le mot clé `synchronised` à modifier:

```
//3b_synchronised_variable

private void incrementCounter_synchro_variable(final Integer modulo) {
   synchronized (MUTEX) {
       count = (count + 1) % modulo;
   }
}
private void iterate_synchro_variable(final Integer nbIteration, final Integer
modulo) {
    for(Integer i = 0; i < nbIteration; i++) {
        this.incrementCounter_synchro_variable(modulo);
    }
}
public Integer exercice3b_synchro_varible(final Integer nbThreads, final Integer
nbIterationByThread, final Integer modulo) throws ExecutionException,
InterruptedException {
    final ExecutorService service = Executors.newFixedThreadPool(nbThreads);
    final List<Future<Runnable>> futures = new ArrayList<>();

    for (Integer i = 0; i < nbThreads; i++) {
        final Future future = service.submit(() ->
iterate_synchro_variable(nbIterationByThread, modulo));
        futures.add(future);
    }

    for (final Future<Runnable> future : futures) {
        future.get();
    }

    service.shutdown();
    return count;
}


//3b_synchronised_méthode
private synchronized void incrementCounter_synchro_methode(final Integer modulo) {
        count = (count + 1) % modulo;
}
private void iterate_synchro_methode(final Integer nbIteration, final Integer modulo)
{
    for(Integer i = 0; i < nbIteration; i++) {
        this.incrementCounter_synchro_methode(modulo);
    }
}
public Integer exercice3b_synchro_methode(final Integer nbThreads, final Integer
nbIterationByThread, final Integer modulo) throws ExecutionException,
InterruptedException {
    final ExecutorService service = Executors.newFixedThreadPool(nbThreads);
    final List<Future<Runnable>> futures = new ArrayList<>();

    for (Integer i = 0; i < nbThreads; i++) {
        final Future future = service.submit(() ->
iterate_synchro_methode(nbIterationByThread, modulo));
        futures.add(future);
    }

    for (final Future<Runnable> future : futures) {
        future.get();
    }
    service.shutdown();
    return count;
}
```

Benchmark pour Ex3b

```
Benchmark                                Mode  Cnt     Score      Error  Units
BenchMarkEx3b.test4b_synchro_methode     avgt    4  2159.382 ± 1318.426  ms/op
BenchMarkEx3b.test4b_synchro_variable    avgt    4  2250.277 ± 1249.285  ms/op
BenchMarkEx3b.testOrigin                 avgt    4  3684.409 ± 4180.025  ms/op
```

Selon le résultat de benchmark, la performance est augmentée si on utilise une seule fois le mot clé `synchronised`. Car ici on n'a pas besoin de le faire deux fois et il va prendre de plus de temps. Ici, la performance entre les deux cas: synchroniser la méthode ou synchroniser la variable est presque pareille car il y a qu'une ligne de code.

Exercice4

1. LinkedList to ArrayList

```
public static byte[] exercice4_array(final byte[]... bytes)
```

Dans cette méthode, nous avons utilisé le `ArrayList`. Pour la partie l'insertion dans tableau, il n'y pas grande différence entre ces deux type car nous insérons les élément à la fin de arrayList (linkedlist). Nous avons pensé que pour la partie lecture:

```
for(Integer i = 0; i < list.size(); i++) {
           result[i] = list.get(i);
       }
```

Comme c'est l'accès aléatoire. Donc `ArrayList` est plus rapide que `LinkedList`.

```
BenchMarkEx4.testOrigin    avgt    4  0.203 ± 0.169  ms/op
BenchMarkEx4.testOrigin_1  avgt    4  0.187 ± 0.092  ms/op
```

2. "No more List"

```
public static byte[] exercice4_nolist(final byte[]... bytes){
      int count = 0;
      for(final byte[] byteArray : bytes) {
           for(final Byte currentByte : byteArray) {
               count++;
           }
       }

       final byte[] result = new byte[count];

       ...
}
```

Dans cette méthode, nous avons enlevé la liste, et nous avons fait une autre boucle pour compter le nombre total de byte.

```
Benchmark                  Mode  Cnt  Score   Error  Units
BenchMarkEx4.testOrigin    avgt    4  0.185 ± 0.190  ms/op
BenchMarkEx4.testOrigin_2  avgt    4  0.099 ± 0.080  ms/op
```

3. Integer to int

```
public static byte[] exercice4_nolist_int
```

Dans cette méthode, nous avons changé le type `Integer` au type `int`. Il y a l'accès aléatoire donc il va faire `unboxing` quand il accède au mémoire. La version `int` marche plus rapide qu'avant.

```
Benchmark                    Mode  Cnt  Score   Error  Units
BenchMarkEx4.testOrigin      avgt    4  0.190 ± 0.216  ms/op
BenchMarkEx4.testOrigin_3    avgt    4  0.098 ± 0.078  ms/op
```

4. Byte to byte

```
public static byte[] exercice4_nolist_int_byte(final byte[]... bytes) {

        int count = 0;
        for(final byte[] byteArray : bytes) {
            for(int j = 0;j<byteArray.length;j++) {
                count++;
            }
        }

        final byte[] result = new byte[count];

        int i = 0;
        for(final byte[] byteArray : bytes) {
            for(final byte currentByte : byteArray) {
                result[i] = currentByte;
                i++;
            }
        }

        return result;
    }
```

Dans cette méthode, nous avons utilisé que le type `byte` pour qu'il ne fait plus de `unboxing` pendant l'exécution.

```
Benchmark                    Mode  Cnt  Score   Error  Units
BenchMarkEx4.testOrigin      avgt    4  0.189 ± 0.202  ms/op
BenchMarkEx4.testOrigin_4    avgt    4  0.009 ± 0.007  ms/op
```

Exercice5

1. Désactiver la vérification de permission

```
public static String getName_2(final Guy guy) throws NoSuchMethodException,
InvocationTargetException, IllegalAccessException {
    Method m = guy.getClass().getMethod("getName");
    m.setAccessible(true);
    return (String)m.invoke(guy);
}
```

Dans cette méthode `getName_2`, nous avons ajouté `m.setAccessible(true);` pour désactiver la vérification de permission du JVM. Mais après le benchmark nous avons trouvé que l'exécution est plus lent qu'avant.

```
Benchmark                (param)  Mode  Cnt  Score   Error  Units
BenchMarkEx5.testGet_2       bob  avgt   10  0.348 ± 0.022  ms/op
BenchMarkEx5.testGet_2      mark  avgt   10  0.344 ± 0.018  ms/op
BenchMarkEx5.testGet_2      peng  avgt   10  0.344 ± 0.015  ms/op
BenchMarkEx5.testOrigin      bob  avgt   10  0.346 ± 0.013  ms/op
BenchMarkEx5.testOrigin     mark  avgt   10  0.339 ± 0.014  ms/op
BenchMarkEx5.testOrigin     peng  avgt   10  0.345 ± 0.015  ms/op
```

2. L'appelle direct de la méthode `.getName`

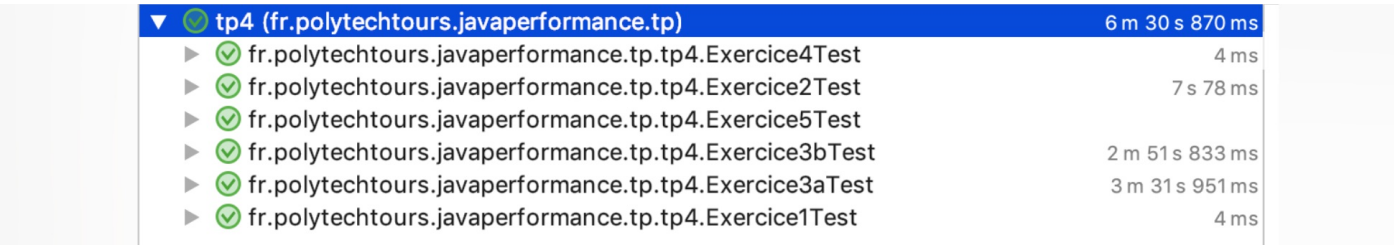
```
public static String getName_1(final Guy guy) {
        return guy.getName();
    }
```

Dans cette méthode, nous avons appelé directement la méthode `getName`. Elle exécute plus rapide qu'avant.

```
Benchmark                (param)  Mode  Cnt  Score    Error  Units
BenchMarkEx5.testGet_1       bob  avgt   10  0.003 ±  0.001  ms/op
BenchMarkEx5.testGet_1      mark  avgt   10  0.003 ±  0.001  ms/op
BenchMarkEx5.testGet_1      peng  avgt   10  0.003 ±  0.001  ms/op
BenchMarkEx5.testOrigin      bob  avgt   10  0.351 ±  0.023  ms/op
BenchMarkEx5.testOrigin     mark  avgt   10  0.343 ±  0.013  ms/op
BenchMarkEx5.testOrigin     peng  avgt   10  0.343 ±  0.013  ms/op
```

On pense que la méthode `getMethod` et `invoke` prennent beaucoup de temps à vérifier la permission, chercher la méthode etc.

Test unitaire



Toutes les méthodes ont passés correctement des tests unitaires.