Benchmarking Java Streams

Processamento de Dados com Streams de JAVA

Afonso Silva Octávio Maia

9 de Janeiro de 2018

Conteúdo

Parte I.

Testes

1. Cálculo dos valores de transações registadas

Observações

Métodos a testar

```
public double sumArray() {
    double[] values = new double[this.transactions.size()];
    int i = 0;

    for (TransCaixa transaction : this.transactions) {
        values[i++] = transaction.getValor();
    }

    double sum = 0.0;

    for (i = 0; i < values.length; i++) {
        sum += values[i];
    }

    return sum;
}
```

Listing 1.1: Cálculo da soma dos valores das transações atraveś de um array do tipo double

```
public double sumDoubleStream() {
    DoubleStream values = this.transactions.stream()
        .mapToDouble(TransCaixa::getValor);
    return values.sum();
}

public double sumDoubleStreamP() {
    DoubleStream values = this.transactions.parallelStream()
        .mapToDouble(TransCaixa::getValor);
    return values.sum();
}
```

Listing 1.2: Cálculo da soma dos valores das transações através de uma DoubleStream

```
public double sumStream() {
    Stream<Double> values = this.transactions.stream()
        .map(TransCaixa::getValor);
    return values.reduce(0.0, (a,b) -> a + b);
}

public double sumStreamP() {
    Stream<Double> values = this.transactions.parallelStream()
        .map(TransCaixa::getValor);
    return values.reduce(0.0, (a,b) -> a + b);
}
```

Listing 1.3: Cálculo da soma dos valores das transações através de Stream<Double>

Resultados

Input	(1) sumArray	(2) sumDoubleStream	(3) sumDoubleStreamP	(4) sumStream	(5) sumStreamP
1000000 transactions	0,014806	0,011167	0,009183	0,028905	0,021365
2000000 transactions	0,021788	0,021629	0,016812	0,061124	0,039403

Análise e conclusões

2. Extração dos primeiros e últimos 20% de transações realizadas

Observações

Métodos a testar

```
public SimpleEntry<List<TransCaixa>, List<TransCaixa>> byDateList() {
    int nelems = 20 * this.transactions.size() / 100;

    List<TransCaixa> sorted = new ArrayList<>(this.transactions);
    sorted.sort(Comparator.comparing(TransCaixa::getData));

    List<TransCaixa> first = sorted.subList(0, nelems);
    List<TransCaixa> last = sorted.subList(
            sorted.size() - 1 - nelems, sorted.size() - 1);

    return new SimpleEntry<>(first, last);
}
```

```
public SimpleEntry<List<TransCaixa>, List<TransCaixa>> byDateSet() {
    int nelems = 20 * this.transactions.size() / 100;

    TreeSet<TransCaixa> sorted = new TreeSet<>(
        // Com este comparador garante-se que a lista fica ordenada
        // e nao se removem os elementos iguais
        (t1, t2) -> t1.getData().isBefore(t2.getData()) ? -1 : 1
    );
    sorted.addAll(this.transactions);

    List<TransCaixa> first = new ArrayList<>(sorted)
            .subList(0, nelems);
    List<TransCaixa> last = new ArrayList<>(sorted.descendingSet())
            .subList(0, nelems);

    return new SimpleEntry<>(first, last);
}
```

```
public SimpleEntry<List<TransCaixa>, List<TransCaixa>> byDateStream() {
    int nelems = 20 * this.transactions.size() / 100;

    List<TransCaixa> first = this.transactions.stream()
```

```
                .sorted(Comparator.comparing(TransCaixa::getData))
                .limit(nelems)
                .collect(Collectors.toList());
        List<TransCaixa> last = this.transactions.stream()
                .sorted((t1, t2) -> t2.getData().compareTo(t1.getData()))
                .limit(nelems)
                .collect(Collectors.toList());

        return new SimpleEntry<>(first, last);
    }
```

```
    public SimpleEntry<List<TransCaixa>, List<TransCaixa>> byDateStreamP() {
        int nelems = 20 * this.transactions.size() / 100;

        List<TransCaixa> first = this.transactions.stream()
                .sorted(Comparator.comparing(TransCaixa::getData))
                .limit(nelems)
                .collect(Collectors.toList());
        List<TransCaixa> last = this.transactions.stream()
                .sorted((t1, t2) -> t2.getData().compareTo(t1.getData()))
                .limit(nelems)
                .collect(Collectors.toList());

        return new SimpleEntry<>(first, last);
    }
```

Resultados

Input	(1) byDateList	(2) byDateSet	(3) byDateStream	(4) byDateStreamP
1000000 transactions	1,409970	1,404477	2,628257	2,468100
2000000 transactions	2,840850	3,395177	5,932004	5,683127

Análise e conclusões

3. Esforço de eliminação de duplicados

Observações

Métodos a testar

```
public Integer[] uniqueArray() {
    Set<Integer> nodups = new TreeSet<>();
    for (int value : this.values) {
        nodups.add(value);
    }

    return nodups.toArray(new Integer[nodups.size()]);
}
```

Listing 3.1: Eliminação dos duplicados através de um array de inteiros

```
public Integer[] uniqueList() {
    List<Integer> aux = new ArrayList<>();
    for (int value : this.values) {
        aux.add(value);
    }

    List<Integer> nodups = new ArrayList<>(new HashSet<>(aux));
    return nodups.toArray(new Integer[nodups.size()]);
}
```

Listing 3.2: Eliminação dos duplicados através de uma lista de inteiros

```
public int[] uniqueIntStream() {
    IntStream values = new Random().ints(this.values.length, 0, 9999);
    return values.distinct().toArray();
}
```

Listing 3.3: Eliminação dos duplicados através de uma stream de inteiros

Resultados

Input	(1) uniqueArray	(2) uniqueList	(3) uniqueIntStream
1000000 random numbers	0,130102	0,027098	0,025731
2000000 random numbers	0,253295	0,056908	0,048782
4000000 random numbers	0,483225	0,094416	0,087328
8000000 random numbers	1,033746	0,223069	0,203687

Análise e conclusões

4. Comparação entre a aplicação de método estático, BiFunction e Lambda

Observações

Métodos a testar

```
public static int div(int x, int y) {
    return x / y;
}

public int[] divSMethodStream() {
    return Arrays.stream(this.values)
        .map(x -> div(x, 2)).toArray();
}

public int[] divSMethodStreamP() {
    return Arrays.stream(this.values).parallel()
        .map(x -> div(x, 2)).toArray();
}
```

Listing 4.1: Divisão de todos os números por 2 através de um método estático

```
public int[] divBiFunStream() {
    BiFunction<Integer, Integer, Integer> f = (x, y) -> x / y;
    return Arrays.stream(this.values)
        .map(x -> f.apply(x, 2)).toArray();
}

public int[] divBiFunStreamP() {
    BiFunction<Integer, Integer, Integer> f = (x, y) -> x / y;
    return Arrays.stream(this.values).parallel()
        .map(x -> f.apply(x, 2)).toArray();
}
```

Listing 4.2: Divisão de todos os números por 2 através de uma BiFunction

```
public int[] divLambdaStream() {
    return Arrays.stream(this.values).map(x -> x / 2).toArray();
}

public int[] divLambdaStreamP() {
```

```
    return Arrays.stream(this.values).parallel().map(x -> x / 2).toArray();
}
```

Listing 4.3: Divisão de todos os números por 2 através de um Lambda

Resultados

Input	(1) divSMethodStream	(2) divSMethodStreamP	(3) divBiFunStream	(4) divBiFunStreamP	(5) divLambdaStream	(6) divLambdaStreamP
1000000 random numbers	0,009272	0,003703	0,014785	0,006303	0,009477	0,005227
2000000 random numbers	0,012993	0,006200	0,022588	0,012378	0,015442	0,007290
4000000 random numbers	0,026100	0,013964	0,045603	0,021953	0,024147	0,012784
8000000 random numbers	0,048509	0,025890	0,083032	0,047038	0,047284	0,024083

Análise e conclusões

5. Comparação da ordenação através de um TreeSet e do método `sorted`

Observações

Métodos a testar

```
public TreeSet<TransCaixa> sortTreeSet() {
    Comparator<TransCaixa> byDate =
        // Com este comparador garante-se que a lista fica ordenada
        // e nao se removem os elementos iguais
        (t1, t2) -> t1.getData().isBefore(t2.getData()) ? -1 : 1;

    return this.transactions.stream()
        .collect(Collectors.toCollection(
                () -> new TreeSet<>(byDate)));
}
```

Listing 5.1: Ordenação através de um TreeSet

```
public List<TransCaixa> sortList() {
    Comparator<TransCaixa> byDate =
        Comparator.comparing(TransCaixa::getData);

    return this.transactions.stream()
        .sorted(byDate).collect(Collectors.toList());
}
```

Listing 5.2: Ordenação através do método sorted

Resultados

Input	(1) sortTreeSet	(2) sortList
1000000 transactions	1,475999	0,953704
2000000 transactions	2,838944	2,010944

Análise e conclusões

6. Determinação da maior transação entre uma determinada hora

Observações

Métodos a testar

```
String biggestTransaction7() {
    List<TransCaixa> transactions = new ArrayList<>(this.transactions);

    transactions.sort(new Comparator<TransCaixa>() {
        @Override
        public int compare(TransCaixa t1, TransCaixa t2) {
            return Double.compare(t1.getValor(), t2.getValor());
        }
    });

    for (TransCaixa transaction : transactions) {
        int hour = transaction.getData().getHour();

        if (hour >= 16 && hour <= 20) {
            return transaction.getTrans();
        }
    }

    return null;
}
```

Listing 6.1: Determinação da maior transação entre uma determinada hora apenas com funcionalidades do JAVA7

```
Optional<String> biggestTransaction8() {
    t.getData().getHour() >= 16 && t.getData().getHour() <= 20;

    return this.transactions.stream()
        .filter(timeInRange)
        .max(Comparator.comparing(TransCaixa::getValor))
        .map(TransCaixa::getTrans);
}
```

Listing 6.2: Determinação da maior transação entre uma determinada hora com auxílio de Streams

Resultados

Input	(1) biggestTransaction7	(2) biggestTransaction8
1000000 transactions	0,300453	0,019556
2000000 transactions	0,629778	0,038287

Análise e conclusões