Concurrent programming

Kristoffer Klokker

2022

Contents

1 Introduction **3**

2 Anonymous,- and lambdafunctions **3**

3 Streams **5**
3.1 Parrallel streams 6

4 The possible bugs of parallelizing **6**
4.1 Race conditions 7

5 Threading in Java **7**

1 Introduction

Concurrency is the act of having multiple execution done simultaneously which interact with each other.
This is done to utilise multiple CPU cores rather than rely on CPU speed.
Not only this but instead of having single powerfull computers, bigger networks of computers can be used.
The benefits comes at a cost of complexity, due to the all possible outcomes of different timed execution.
The risk includes race-conditions where variables are changed while used in another thread or liveness where threads are stuck forever waiting for a non realeased variable nad such.

2 Anonymous,- and lambdafunctions

For at simple class which is given in an argument, instead of creating a class and then parsing it, the class can be created in the argument field.
For instance a class which implements comparable, it can be programmed as such:

```
1  public interface StringExecute {
2      public void run(String content);
3  }
4
5  public static void doAndMeasure( StringExecutable
       runnable ) {
6      long t1 = System.currentTimeMillis();
7      runnable.run();
8      System.out.println( "Elapsed time: " + (System.
           currentTimeMillis() - t1) + "ms" );
9  }
10
11 public static void anonFunc() {
12     doAndMeasure(new StringExecute() {
13         public static void run(String content) {
14             System.out.println(content + " Wow!");
15         }
16     });
17 }
18
19 public static void lambdaFunc() {
20     doAndMeasure( (content) -> System.out.println(
```

```
        content + " Wow!"););
}

public static void lambdaFuncOpt() {
    doAndMeasure( (content) -> System.out::println;); //
        Only prints content and not + " Wow!"
}
```

Here lambda function is only possible due to the compiler knowing what type of object is created due to restrains from the function and the runnable interface only contains a single run function. Another use of lambda expression is when working with maps.

```
public static void main() {
    String text = "Hello world hope your having a good
        day!";
    Map<Charachter, Interger> occurrences = new HashMap
        <>();
    for(int i = 0; i < text.length(); i++){
        final char c = text.charAt(i);
        if(occurrences.containsKey(c))
            occurrences.put(c,occurrences.get(c)+1);
        else
            occurrences.put(c,1);
    }
    for(int i = 0; i < text.length(); i++){
        final char c = text.charAt(i);
        occurrences.merge(c,1,(currValue,value) ->
            currValue+value);
    }
```

Both for loops do the same, the second uses merge which takes a position (c) and a default value (1) and a bifunction which is a function with two inputs. The arguments will automaticly be assigned such currValue is the current hash value and value is the same as the default value 1.
Some of the most usefull built-in functional interfaces includes:

- Predicates - 1 argument returns boolean
- Functions - 1 argument returns result
- Suppliers - Like Functions but no arguments
- Consumers - 1 argument no return
- Comaprators - implements compareTo

3 Streams

Streams are like an foreach and like the name it is a stream of data. In more fine words it is a monad which is datastructure of a sequence of steps of operations.
There are different types of sterams, the *Stream* is an object stream, wheras primitives stream also exists $IntStream, LongStream, DoubleStream$ with poosibilites like $IntStream.range(1,4)$ which has a stream of 1,2,3 and $IntStream.range(1,4).sum()$
To tell the compiler if an object stream is transformed into a primitive type $.mapTo(Int, Double, ...)$ with a parser as argument.
The most common functions used by streams are:

```
40 List<String> words = Arrays.asList("watercan", "digital"
       , "citizen");
41 //Performs function with each element
42 words.stream().forEach(word -> System.out::println);
43 //Modifies the element
44 words.stream().map(word -> String::length);
45 //Only elements which fullfill the function will 'parse'
46 words.stream().filter(word -> word.startsWith("d"));
47 //Streams the sorted stream
48 words.stream().sorted((s1, s2) -> s1.compareTo(s2));
49 //Counts the number of elements
50 words.stream().count();
51 //replace element by stream of given function
52 words.stream().flatmap(word -> Stream.of(word.split("a")
       );
53 //Gather elements in stream, for analyzis or into an
       object ex list
54 words.stream().collect(Collectors.toList());
55 //Reduce takes a stream and makes a single element Ex.
       this takes the longest string
56 words.stream().reduce((s1,s2) -> s1.length() > s2.length
       () ? s1 : s2);
```

The functions can then be chaineed together, so methods which returns a stream can be worked upon.
When working with file stream it is done as following:

```
57 try( Stream< String > lines = Files.lines(Paths.get("
       text.txt"))) {
58     lines.forEach(System.out::println);
59 }
```

```
60 catch ( IOException e) {
61     e.printStackTrace();
62 }
```

By using the try and putting it into the parenthesis it will cause the stream to close when the stream is finished.
When a terminal operation (returns result rather than intermideate which returns stream) is done one the stream, it will close the stream. To counteract this a function which returns the stream may be used as such:

```
63 Supplier<Stream<String>> streamSupplier =  () -> Stream.
      of("d2", "a2").filter(s -> s.startsWith("a"));
64 //Returns true due to there is elements in the stream
65 streamSupplier.get().anyMatch(s -> true);  //
66 //Returns false due to elements existing in stream
67 streamSupplier.get().noneMatch(s -> true);
```

3.1 Parrallel streams

When working with parrallel stream the stream is outsourced to threads using a common ForJoinPool.
This makes stream elements being handled side by side.
Some function needs modifications such as reduce:

```
68 words.stream().parallelStream().reduce(0,(sum,s) -> sum
      + s.length(), (sum1, sum2) -> sum1+sum2);
```

Reduce is now broken up onto three arguments,

- indetifier - initial value
- accumulator - takes current result and element and operatoes
- Combiner - takes two partial accumulators and combines them

Like sot the *sort*() function will either wait for all parralle streams or use parralell sorting in large amounts.

4 The possible bugs of parallelizing

When working with parrallel threads it is important to encapsulate as much as possible. Threads may access data outside it thread, but this can lead to some of the following bugs.

4.1 Race conditions

This is a bug which accour when two threads are accessing the same global variable.
This is a bug based upon atomicity is not kept and an operation is not done before another access data from the operation.
The most simple form is two threads which are incrementing a variable.
Say that the first threads read the variable, at the same the second thread reads the variable.
This will conclude in both threads having the same value for the variable and when incrementing the variable will only increment once.
To counteract this lock can be used. This is where code snippets are ran syncronised and threads are not able to execute twice.

5 Threading in Java

Threading is done thorugh the *Thread* object which takes a runable object as an argument.
The thread is then executed by *start*() and *join*() is used as a wait till a thread is done.
If join is not used the main thread will stop its execution before child threads are done.

```
69 Thread t1 = new Thread( () -> {for(int i = 0; i < 1000;
      {i++; System.out.println(i);})});
70 t1.start();
71 try{
72     t1.join();
73 }
74 catch(InterruptedException e) {e.printStackTrace();}
```

The try is due to threads may be interrupted, which otherwise would case the program to halt, due to the thread never joining.
For when working with locks in java the following can be done:

```
75 private static class Counter {
76     private int c = 0;
77 }
78
79 public static void main() {
80     Thread t1 = new Thread( () -> {for(int i = 0; i <
          1000; i++) {
```

```
81          synchronized( counter ) {
82              counter.c++;
83          }
84      });
85      t1.start();
86      Thread t2 = new Thread( () -> {for(int i = 0; i <
            1000; i++) {
87          synchronized( counter ) {
88              counter.c++;
89          }
90      });
91      t2.start();
92      try{
93          t1.join();
94          t2.join();
95      }
96      catch(InterruptedException e) {e.printStackTrace();}
```

The synchronized will here lock the counter object before access from which the code is executed. This will ensure that c is equal 2000.
Another way is to making a whole method syncronised by:

```
97  private synchronized void increment() {
98       i++;
99  }
```