Concurrent programming

Kristoffer Klokker

2022

Contents

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.

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:

```java
public interface StringExecute {
    public void run(String content);
}

public static void doAndMeasure( StringExecutable
   runnable ) {
    long t1 = System.currentTimeMillis();
    runnable.run();
    System.out.println( "Elapsed time: " + (System.
       currentTimeMillis() - t1) + "ms" );
}

public static void anonFunc() {
    doAndMeasure(new StringExecute() {
        public static void run(String content) {
            System.out.println(content + " Wow!");
        }
    });
}

public static void lambdaFunc() {
    doAndMeasure( (content) -> System.out.println(
       content + " Wow!"););
}

```

```java
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.

```java
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:

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

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

```java
catch ( IOException e) {
    e.printStackTrace();
}
```

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:

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

```java
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.