Report

Alec Maughan Assignment 1 2021

Introduction

For this assignment I used the Kotlin programming language, which has its own 'Coroutines' multithreading system.

Below is an introduction to Coroutines if you are unfamiliar.

This system is higher level than Java's multithreading system, so there is not necessarily a 1-to-1 mapping to things like BlockingQueues or thread pools, and it is not as necessary to divide the code into multiple files as many complex operations take very little code.

Coroutines are operations that run in a Coroutine context, which is like a thread pool. There are three contexts out of the box, all managed under the hood (Don't need to be opened/closed).

- `Dispatchers.Default` - An arbitrary thread pool, used for CPU operations.
- `Dispatchers.Main` - The UI thread, for JavaFX this is `Platform.runLater`.
- `Dispatchers.IO` - A thread pool for IO operations.

This project mainly uses the Coroutines `Flow` system, which are conceptually similar to Java's `CompletableFuture`, but for a lazily evaluated sequence of data. Here is an overview of how flows work and what they can do.

```
// Create a flow that 'emits' values 1 through 10
(1..10).asFlow()

    // Filter out odd numbers
    .filter { it % 2 == 0 }

    // Multiply each value by 2
    .map { it * 2 }

    // Run the above operations in the CPU context (Thread pool)
    .flowOn(Dispatchers.Default)

    // Update a GUI
    .onEach { myUserInterface.showValue(it) }

    // Run the above operations in the UI context (Platform.runLater())
    .flowOn(Dispatchers.Main)

    // Separate the above and below code to run asynchronously, passing values
    // through a shared buffer
    .buffer()

    // Write the value to a file
    .onEach { File("numbers.txt").appendText("$it\n") }

    // Run the above code in the IO context (Thread pool)
    .flowOn(Dispatchers.IO)

    // Handle any thrown exceptions
    .catch { e -> println(e) }

    // Run through the flow, you could also choose to do something with the
    // values here
    .collect() // (The terminating blocking operation)
```

As you can see, lots of the low level stuff is handled behind the scenes.

Multithreading Design Explanation

Introduction

Because of how little code is needed to do multithreading, all the multithreading code was done from two flows in a single class, `FileSearcher`, except for the writing of `results.txt`, which was handled by the `ResultsWriter` class.

Because there are no mutexes or similar constructs needed, there is no risk of deadlock, as long as we don't stray far from how Coroutines are meant to be used.

`ResultsWriter`

This is a separate class that the `FileSearcher` holds an instance of. The class manages its own context that is backed by exactly one thread. When it receives an object to write to the file, it launches a new Coroutine to run on its context that just appends to the `results.txt` file.

Because it operates in exactly one thread per instance, there's no risk of race condition or deadlock when writing.

`FileSearcher`: `fileFlow`

The first flow, `fileFlow` walks through the directory tree on the IO context and emits all files it comes across, filtering out directories, non-text files, and empty files. This is separate because new instances are made of it many times.

`FileSearcher`: `fileSearchFlow`

The other flow, `fileSearchFlow`, does everything else, covered below.

This flow is collected (run) in its own coroutine when the UI starts a search.

Start

Starting off, `fileFlow` is collected to count the total number of files and calculate the total number of comparisons, for the progress bar.

Then, the `flatMatConcat` operation is run on a new instance of `fileFlow`, mapping each file emitted from `fileFlow` to a new 'inner' `fileFlow`, and collapsing and concatenating into one flow when the inner flow is done. This is how each file can be run against every other file.

Inner Flow

The inner flow is broken into three parts that operate in parallel and pass data through the flow buffer, like producers and consumers. These three parts operate on their own coroutines and concern IO, CPU, and UI operations.

Inner Flow Coroutine 1: IO

Flow value: `file1`, `file2`.

- In the CPU context filter out cases when the files are the same and when the path of `file1` is lexicographically less than that of `file2`, to ensure comparisons are only done once.
- In the IO context, the contents of the file are read into strings.

Inner Flow Coroutine 2: CPU

Flow value: `file1`, contents of `file1`, `file2`, contents of `file2`.

- In the CPU context, the read files have their similarity calculated and placed into a `ComparisonResult` object containing the filenames and similarity value.

Flow value: `result` (Object of `ComparisonResult`)

- The results are then sent to a `ResultsWriter` object, which asynchronously adds `result` to the `results.txt` file (Does not block the flow).

Inner Flow Coroutine 3: UI

Flow value: `result`.

- In the UI context, the progress bar is incremented.

This ends the inner flow operations, and the inner flow is concatenated onto the outer flow.

Outer Flow

- In the CPU context, low similarity results are filtered out.
- In the UI context, the remaining results are added to the table.

Completion

On the completion of the entire flow, cancellation is checked for and the ui is updated accordingly. The `ResultsWriter`'s context thread is also closed.

Scalability

Handling of Scale

In the app's current state, very large files cannot be compared, as the longest common subsequence algorithm exceeds the Java heap limit. This could be alleviated by using a computationally cheaper LCS algorithm that reads the file as it goes.

In terms of large numbers of files, the app does just fine in its current state. The flows work on dynamically sized buffers, and the coroutine contexts handle allocating as many threads as required to their thread pools.

Possible Improvements

A possible optimisation to be made would be to cache the files emitted by fileFlow after the first collection, so that the files don't have to be traversed multiple times.

The contents of the files could also be cached so that they would only be read in once, but this also comes with problems with how to store all of this data.

Use cases

This app could be used in a mass storage service to find duplicate or almost duplicate files to remove and save storage space.