Concurrency in modern programming languages: Java

Deepu K Sasidharan Follow | 30 Apr 2021 | 8 mins read |

This is part of my "concurrency in modern programming languages" series

1. Concurrency in modern programming languages: Introduction
2. Concurrency in modern programming languages: Rust
3. Concurrency in modern programming languages: Golang
4. Concurrency in modern programming languages: JavaScript on NodeJS
5. Concurrency in modern programming languages: TypeScript on Deno
6. **Concurrency in modern programming languages: Java**
7. Concurrency in modern programming languages: Rust vs Go vs Java vs Node.js vs Deno vs .NET 6

This is a multi-part series where I'll be talking about concurrency in modern programming languages and will be building and benchmarking a concurrent web server, inspired by the example from the Rust book, in popular languages like Rust, Go, JavaScript (NodeJS), TypeScript (Deno), Kotlin and Java to compare concurrency and its performance between these languages/platforms. The chapters of this series are as below.

1. Introduction
2. Concurrent web server in Rust

Concurrency in Java

> *The Java programming language and the Java virtual machine (JVM) have been designed to support concurrent programming, and all execution takes place in the context of threads*
>
> - *Wikipedia*

Java had support for concurrent programming from its early days. Prior to Java 1.1 it even had support for green threads (virtual threads). Spoiler Alert! It's coming back again with Project Loom.

Concurrent programming has always been at the core of Java as it was aimed at multi-threaded and multi-core CPUs. While not as simple as to use, it was powerful and flexible for almost any use case. While powerful, it's also quite complex especially when you have to access data between threads since the default mechanism in Java, due to its OOP roots, is to use shared state concurrency by synchronizing the threads.`goroutines`

Threads are at the core of concurrent & asynchronous programming in Java. From JDK 1.1 onwards these threads would map 1:1 to OS threads. Due to its early inception, the ecosystem has really mature libraries as well, from HTTP servers to concurrent message processors and so on. Asynchronous programming caught up a bit late in Java, the building blocks were there but it was practically useable only from Java 8, but it has matured as well and now has a great ecosystem with support for reactive programming and asynchronous concurrency.

Java 8 bought a lot of improvements and simplifications to make it easier to do concurrency. For example, standard Java APIs like the Stream API even provides a way to do parallel processing easily by just invoking a method call on complex and CPU intensive pipelines.

With Java, it's possible to do multi-threaded concurrency or parallel programming as well as asynchronous programming. This means as we saw in the first chapter, we can mix and match these models to get the best possible performance for any use case.

Multi-threading

Java provides building blocks to create and manage OS threads as part of the standard library and it also provides implementations required for shared-state concurrency using locks and synchronization. Message-passing concurrency is not provided by default but can be done using external libraries like Akka or using an Actor model implementation. However, due to the memory model, it's up to the developer to ensure there are no data races or memory leaks in the concurrent program.

In order to make multi-threading even more efficient, Java provides ways to create thread pools and reuse those threads to increase throughput. This will become even better once Project loom is released, hopefully with Java 17 or 18. Technically Java has one of the most mature ecosystems when it comes to multi-threading and most Java frameworks that you would end up using will be making use of it internally for performance improvements.

Asynchronous processing

Technically asynchronous programming is not part of concurrency but in practice, it goes hand in hand for many use cases and improves performance, and makes resource usage more efficient. In Java asynchronous programming is achieved using the same building blocks as concurrent/parallel programming. a.k.a, Threads. This wasn't very popular in Java before Java 8 due to complexity and, let's be honest, the lack of things like lambdas, functional programming support, CompletableFuture, and so on.

The latest versions of Java provide the building blocks required for asynchronous programming with standard interfaces and implementations. But do keep in mind that using an asynchronous programming model increases the overall complexity and the ecosystem is still evolving. There are also many popular libraries and frameworks like Spring and RxJava that support asynchronous/reactive programming.

Java still doesn't have any syntax sugar for async/await though but there are alternatives like the EA Async library that's close enough.

Benchmarking

Now that we have some basic understanding of concurrency features in Java, let us build a simple concurrent web server in Java. Since Java offers multiple ways to achieve this we'll be building two sample applications and comparing them. The Java version used is the latest (16.0.1) at the time of writing.

Multi-threaded concurrent webserver

This example is closer to the Rust multi-threaded example we built in the rust chapter, I have omitted import statements for brevity. You can find the full example on GitHub here. We use for this. We are not using any external dependency in this case.`java.net.ServerSocket`

```java
public class JavaHTTPServer {
    public static void main(String[] args) {
        var count = 0; // count used to introduce delays
        // bind listener
        try (var serverSocket = new ServerSocket(8080, 100)) {
            System.out.println("Server is listening on port 8080");
            while (true) {
                count++;
                // listen to all incoming requests and spawn each connection in a new thread
                new ServerThread(serverSocket.accept(), count).start();
            }
        } catch (IOException ex) {
            System.out.println("Server exception: " + ex.getMessage());
        }
    }
}

class ServerThread extends Thread {

    private final Socket socket;
    private final int count;
    public ServerThread(Socket socket, int count) {
        this.socket = socket;
        this.count = count;
    }

    @Override
    public void run() {
        var file = new File("hello.html");
        try (
                // get the input stream
                var in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
                // get character output stream to client (for headers)
                var out = new PrintWriter(socket.getOutputStream());
                // get binary output stream to client (for requested data)
                var dataOut = new BufferedOutputStream(socket.getOutputStream());
                var fileIn = new FileInputStream(file)
        ) {
            // add 2 second delay to every 10th request
            if (count % 10 == 0) {
                System.out.println("Adding delay. Count: " + count);
                Thread.sleep(2000);
            }

            // read the request first to avoid connection reset errors
            while (true) {
                String requestLine = in.readLine();
                if (requestLine == null || requestLine.length() == 0) {
                    break;
                }
            }

            // read the HTML file
            var fileLength = (int) file.length();
            var fileData = new byte[fileLength];
            fileIn.read(fileData);

            var contentMimeType = "text/html";
            // send HTTP Headers
            out.println("HTTP/1.1 200 OK");
            out.println("Content-type: " + contentMimeType);
            out.println("Content-length: " + fileLength);
            out.println("Connection: keep-alive");

            out.println(); // blank line between headers and content, very important!
            out.flush(); // flush character output stream buffer

            dataOut.write(fileData, 0, fileLength); // write the file data to output stream
            dataOut.flush();
        } catch (Exception ex) {
            System.err.println("Error with exception : " + ex);
        }
    }
}
```

As you can see we bind a TCP listener using to port 8080 and listen to all incoming requests. Each request is processed in a new thread.`ServerSocket`

Let us run a benchmark using ApacheBench. We will make 10000 requests with 100 concurrent requests.

```
❯ ab -c 100 -n 10000 http://127.0.0.1:8080/
This is ApacheBench, Version 2.3 <$Revision: 1879490 $>
...

Document Path:          /
Document Length:        176 bytes

Concurrency Level:      100
Time taken for tests:   20.326 seconds
Complete requests:      10000
Failed requests:        0
Total transferred:      2600000 bytes
HTML transferred:       1760000 bytes
Requests per second:    491.98 [#/sec] (mean)
Time per request:       203.262 [ms] (mean)
Time per request:       2.033 [ms] (mean, across all concurrent requests)
Transfer rate:          124.92 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    1   1.5      0      13
Processing:     0  201 600.0      1    2023
Waiting:        0  201 600.0      0    2023
Total:          0  202 600.0      1    2025

Percentage of the requests served within a certain time (ms)
  50%      1
  66%      2
  75%      4
  80%      6
  90%   2000
  95%   2001
  98%   2003
  99%   2006
 100%   2025 (longest request)
```

As you can see the request handler thread sleeps for 2 seconds for every 10th request. In a real-world scenario, the thread pool itself could become the bottleneck and you may not be able to set so many threads as the OS may not be able to provide so many thus creating increased resource usage and bottleneck. In this simple use case, since each thread spawns and processes the request really fast we won't encounter an issue.

So let's see if we can have another solution without such a bottleneck.

Asynchronous concurrent webserver

This example is closer to the asynchronous example from the rust chapter, I have omitted import statements for brevity. You can find the full example on GitHub here. Notice that we are using here and no external dependencies.`java.nio.channels.AsynchronousServerSocketChannel`

```java
public class JavaAsyncHTTPServer {
    public static void main(String[] args) throws Exception {
        new JavaAsyncHTTPServer().start();
        Thread.currentThread().join(); // Wait forever
    }

    private void start() throws IOException {
        // we shouldn't use try with resource here as it will kill the stream
        var server = AsynchronousServerSocketChannel.open();
        server.bind(new InetSocketAddress("127.0.0.1", 8080), 100); // bind listener
        server.setOption(StandardSocketOptions.SO_REUSEADDR, true);
        System.out.println("Server is listening on port 8080");

        final int[] count = {0}; // count used to introduce delays

        // listen to all incoming requests
        server.accept(null, new CompletionHandler<>() {
            @Override
            public void completed(final AsynchronousSocketChannel result, final Object attachment) {
                if (server.isOpen()) {
                    server.accept(null, this);
                }
                count[0]++;
                handleAcceptConnection(result, count[0]);
            }

            @Override
            public void failed(final Throwable exc, final Object attachment) {
                if (server.isOpen()) {
                    server.accept(null, this);
                    System.out.println("Connection handler error: " + exc);
                }
            }
        });
    }

    private void handleAcceptConnection(final AsynchronousSocketChannel ch, final int count) {
        var file = new File("hello.html");
        try (var fileIn = new FileInputStream(file)) {
            // add 2 second delay to every 10th request
            if (count % 10 == 0) {
                System.out.println("Adding delay. Count: " + count);
                Thread.sleep(2000);
            }
            if (ch != null && ch.isOpen()) {
                // Read the first 1024 bytes of data from the stream
                final ByteBuffer buffer = ByteBuffer.allocate(1024);
                // read the request fully to avoid connection reset errors
                ch.read(buffer).get();

                // read the HTML file
                var fileLength = (int) file.length();
                var fileData = new byte[fileLength];
                fileIn.read(fileData);

                // send HTTP Headers
                var message = ("HTTP/1.1 200 OK\n" +
                        "Connection: keep-alive\n" +
                        "Content-length: " + fileLength + "\n" +
                        "Content-Type: text/html; charset=utf-8\r\n\r\n" +
                        new String(fileData, StandardCharsets.UTF_8)
                ).getBytes();

                // write the to output stream
                ch.write(ByteBuffer.wrap(message)).get();

                buffer.clear();
                ch.close();
            }
        } catch (IOException | InterruptedException | ExecutionException e) {
            System.out.println("Connection handler error: " + e);
        }
    }
}
```

As you can see we bind an asynchronous listener to port 8080 and listen to all incoming requests. Each request is processed in a new task provided by . We are not using any thread pools here and all the incoming requests are processed asynchronously and hence we don't have a bottleneck for maximum connections. But one thing you may immediately notice is that the code is much more complex now.`AsynchronousServerSocketChannel`

Let us run a benchmark using ApacheBench. We will make 10000 requests with 100 concurrent requests.

```
ab -c 100 -n 10000 http://127.0.0.1:8080/

This is ApacheBench, Version 2.3 <$Revision: 1879490 $>
...

Document Path:          /
Document Length:        176 bytes

Concurrency Level:      100
Time taken for tests:   20.243 seconds
Complete requests:      10000
Failed requests:        0
Total transferred:      2770000 bytes
HTML transferred:       1760000 bytes
Requests per second:    494.00 [#/sec] (mean)
Time per request:       202.431 [ms] (mean)
Time per request:       2.024 [ms] (mean, across all concurrent requests)
Transfer rate:          133.63 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.6      0       5
Processing:     0  201 600.0      0    2026
Waiting:        0  201 600.0      0    2026
Total:          0  202 600.0      0    2026

Percentage of the requests served within a certain time (ms)
  50%      0
  66%      1
  75%      3
  80%      4
  90%   2000
  95%   2001
  98%   2002
  99%   2003
 100%   2026 (longest request)
```

We got almost identical results here, this one is even faster by 100ms. Hence this version seems much more efficient than the multi-threaded version for this particular use case. However at the cost of added complexity.

Conclusion

As I explained in the first part of this serious, this simple benchmarking is not an accurate representation for all concurrency use cases. It's a simple test for a very particular use case, a simple concurrent web server that just serves a file. The idea is to see the differences in solutions and to understand how concurrency works in Java. And for this particular use case, asynchronous solutions do seem to be the best choice.

So stay tuned for the next post where we will look at concurrency in Kotlin and build the same use case in Kotlin.

References

- blogs.oracle.com
- dzone.com
- www.vogella.com
- dzone.com
- www.baeldung.com
- dzone.com
- www.baeldung.com

If you like this article, please leave a like or a comment.

You can follow me on Twitter and LinkedIn.

Cover image credit: Photo by Evgeniya Litovchenko on Unsplash

Post **6** of **7** in series **"concurrency in modern programming languages"**.

« Prev post in series **Next post in series »**

#concurrency #java #jvm #languages

« The state of Linux as a daily use OS in 2021 **My second impression of Rust and why I think it's a great general-purpose language! »**

Share

9 Comments

What's your reaction?

🔥 2 😍 0 😮 0 😢 0 😂 0 😡 0

9 Comments 1 ONLINE Sort By Best

Write your comment (Markdown is supported) LOGIN SIGNUP

Manuel Soto 1 year ago
This is a very bad idea to create an OS thread per request, you should use an executor
Reply Share 👍 0 👎 0

Deepu K Sasidharan 1 year ago
Its a comparison, in real world you should always be using mature libraries like actixhweb, Undertow, netty, gin etc
Reply Share 👍 0 👎 0

Y **Yewo Mhango** 2 years ago
I haven't started reading this particular article yet, but it seems your code snippets use a sans-serif font, instead of a monospace font. Just wanted to let you know in case you haven't noticed already :)
Reply Share 👍 0 👎 0

D **Deepu K Sasidharan** 2 years ago
Hey thanks for letting me know. May I know which OS you are on? I have set to use system default font for most so may be I should force a monospace font instead
Reply Share 👍 0 👎 0

Y **Yewo Mhango** 2 years ago
Okay, I'm using Android, with Opera browser (not Opera Mini). And yeah, you could just do that since reading code in a non-monospace font looks kinda weird.
Reply Share 👍 0 👎 0

D **Deepu K Sasidharan** 2 years ago
Interesting. I'm on Android but with Chrome and it renders monospace. So I guess forcing the font would be more reliable
Reply Share 👍 0 👎 0

Y **Yewo Mhango** 2 years ago
Okay, I see. So I guess you'll just have to do that then
Reply Share 👍 0 👎 0

D **Deepu K Sasidharan** 2 years ago
done. Let me know if it looks better now
Reply Share 👍 0 👎 0

Y **Yewo Mhango** 2 years ago
Yeah, thanks. It's monospaced now :)
Reply Share 👍 0 👎 0

by Hyvor Talk

Never miss a **story** from us, subscribe to our newsletter

Email address

Subscribe

Explore →

languages (15) programming (15) javascript (13) java (11) rust (9) beginners (8)
concurrency (7) go (7) linux (6) microservices (5) typescript (5) computerscience (4)
development (4) fedora (4) functional (4) kubernetes (4) thepragmaticprogrammer (4)
async (3) jhipster (3) azure (2) career (2) codequality (2) deno (2) devops (2) discuss (2)
docker (2) garbagecollection (2) gnome (2) golang (2) kde (2) node (2) nodejs (2)
openjdk (2) react (2) showdev (2) terminal (2) webassembly (2) webdev (2) architecture (1)
bash (1) blogging (1) books (1) clojure (1) codenewbie (1) desktop (1)
developerexperience (1) devrel (1) devsecops (1) distributedsystems (1) engineering (1)
gaming (1) ide (1) interview (1) istio (1) jdk (1) jekyll (1) js (1) jvm (1) kotlin (1)