Virtual Threads

Java 21’s features

Post No.2

Threads

- Definitions:
  - A *thread* is the smallest unit of processing that can be scheduled.
  - Before Java 19 the java just supports platform threads or kernel threads.
  - Based on Official java definition for platform thread, “a platform thread is implemented as a thin wrapper around an operating system (OS) thread.”
  - In a simple words, platform thread in java has a one to one relationship with OS thread. So the number of available platform threads is limited to the number of OS threads.

Threads and Concurrent Programming

In the following slides we will try to answer the following questions:

- Why VIRTUAL THREADs have been added to JDK?
- What Performance do you except from them?
- How do they work?

Threads and Concurrent Programming

- Suppose that you want to call a webservice to get a user object from a remote server. You have the name of the user and you need to fetch the full user object.
- your code probably will be as follow:

1. Json request = buildUserRequest(name); → // Create JsonObject;
2. String userJson = UserServer.getUserByName(request); → // Sending request to network
3. User user = Json.deserialize(userJson); → // Unmarshaling response to user Object

* This code is simple to write, simple to read, simple to debug, simple to test and simple to maintain.

Threads and Concurrent Programming

- Now let's compute the CPU usage during running this code.
- The first step of code is very simple , it just is an im-memory computations, so it can be assumed that it will run in the order of nanosecond.
- The second Step is the sending request to the network and waiting for the response. This takes much longer probably in the order of 100 millisecond. And the CPU is not doing anything during this time. It just there and it's waiting for the response.
- The last step is unmarshaling Json and it takes in the order of nano second.

** During this operation the CPU is working the order of nano second and it's be idle in the order of 100 millisecond. **So the CPU usage during this operation will be : 0.0001%**

Threads and Concurrent Programming

- During the last operation and by knowing this fact that in a classical model one thread assigns to each request, the first solution you may thinking about this is load more than 1 request in parallel. Let's calculate the number of parallel request to make CPU busy 100%.

Number of Request	Number of working thread	CPU usage
1	1	0.0001%
10	10	0.001%
100	100	0.01%
1000	1000	0.1%
10000	10000	1%
100000	100000	10%
1000000	1000000	100%

Threads and Concurrent Programming

- Based On the last table for using 100% of the CPU usage we need to create about 1 million Threads in parallel.
- Based on Java documentation a java thread is implemented as a thin wrapper to OS thread so typically create each thread takes 2MB of memory and needs times in the order of millisecond.
- Creating 1 million threads takes 2TB RAM, and also needs 15 Second time.
- If you want to create about 1 million threads in a loop, based on your operating system, you probably hit a limit long before that, at about a few thousand threads.
- So if you want to solve the problem with this approach even if there is not any limitation of resource(2TB of Memory) you will quickly reach the limit and your CPU usage will still be very low.
- Ex. : Spring Web is a traditional web framework that is built on top of the Servlet API. It is designed to handle blocking I/O, where a thread is blocked until a response is received from a database or another service.

Threads and Asynchronous programming

- There is 2 way to solving the problem:
  - *Create e new type of threads that are less expensive than the current kernel thread*
  - **Lunch more than one request per platform thread.**
    - Launching more request per thread means that you can not write your code using imperative approach anymore. Imperative approach means: split the code into small atomic operations. Each operation gets inputs, performs one thing and produces a result.
    - For doing above approach, all this operation should be wrote as lambdas and then you should wire them together using an asynchronous frameworks.
    - The responsibility of this framework is that execute your operations in a right orders and distribute them among the threads it has, to keep your CPU busy.
    - i.e. one of the lambdas that will launch a request on the network should immediately return, to make sure that it dose not block the thread that is running it. *AVOID writing blocking lambdas *
    - Ex.: Spring Webflux, is a reactive web framework that is built on top of Reactive Streams. It is designed to handle non-blocking I/O, where a thread is not blocked while waiting for a response from a database or another service. Instead, the application can continue to process other requests while waiting for the response

Threads and Asynchronous programming

- Lunch more than one request per platform thread.
  - As a developer, you should change your way of writing the code for using in asynchronous frameworks.
  - One of the most popular asynchronous APIs is CompletableFuture that it is a part of JDK.
  - Let's consider the previous example for get user from a remote server by username.
    - Json request = buildUserRequest(name);
    - String userJson = UserServer.getUserByName(request);
    - User user = Json.deserialize(userJson);

    This code is imperative, simple to write, simple to debug, simple to test and simple to maintain.

Threads and Asynchronous programming

- The Code should be adapted with CompletableFuture API, as we want to run the code asynchronously.

```
CompletableFuture completableFuture =
    supplyAsync(()-> buildUserRequest(name))
    .thenApply(()-> UserServer.getUserByName(request))
    .thenApply(()-> Json.deserialize(userJson));
```

**note that this is a very simple example and all the operation perform by only 1 thread.*

- In this type of code how can check and debug the code?
- How to write the unit test for it?
- This type of code is very hard to understand
- How a piece of work actually work together.
- This code is not imperative; it is hard to write, hard to maintain, impossible to debug(running step by step) and impossible to write unit tests.

Virtual Threads and Concurrent Programming

- As you remember another way to solve CPU usage problem is :

"*Create e new type of threads that are less expensive than the current kernel thread*"

And this type of Threads named VIRTUAL THREADs.

- There is a question here, how much the virtual threads must be less expensive?
  - The answer is based on the aforementioned table :

Number of Request	Number of working thread	CPU usage
1	1	0.0001%
10	10	0.001%
100	100	0.01%
1000	1000	0.1%
10000	10000	1%
100000	100000	10%
1000000	1000000	100%

***we need to be able to launch in the order of 1,000,000 million threads.**

Virtual Threads and Concurrent Programming

Number of Request	Number of working thread	CPU usage
1	1	0.0001%
10	10	0.001%
100	100	0.01%
1000	1000	0.1%
10000	10000	1%
100000	100000	10%
1000000	1000000	100%

- As it mentioned before, the platform thread can be launched in the order of 1,000, as the threads need 100 millisecond to create.
- So we need new types of threads(Virtual Threads) that a thousand times less expensive than the current platform threads.
- With this approach, your code writes in a imperative and a blocking way but becomes compatible with the number of requests per second that you need. And you will not need to write asynchronous code anymore.
- * Virtual threads from JDK 21 are much lighter than platform threads, by a factor of more than a thousand.

Virtual Threads and Concurrent Programming

- As a Result of last topic:
  - By using Virtual Threads we can block a thread for each request but this thread is a virtual thread and is much lighter than platform thread.
  - We can launch in the order of 1,000,000 virtual threads and make CPU busy 100%.
- How to create and use them?
  - To create a new virtual thread in Java, you use the new Thread.ofVirtual() factory method, passing an implementation of the Runnable interface.

```
Runnable runnable = () -> {
    for(int i=0; i<10; i++) {
        System.out.println("Index: " + i);
    }
};

Thread vThread = Thread.ofVirtual().start(runnable);
```

  - we can use and join the virtual thread just like a platform thread


The End.