Ratpack Executions: Async + Serial, not Parallel

May 1, 2016

Developers familiar with Ratpack know that it is a non-blocking and asynchronous framework that's built on top of Netty. It uses a small pool of "compute" threads (by default `2 * <# of CPUs>`) to do all of the non-blocking processing of thousands of requests a second.

The documentation (and blog posts and Dan Woods' excellent *Learning Ratpack*) all discuss another benefit of Ratpack: **serialized execution of asynchronous code**.

Even though I'd read about Ratpack's serial execution model, I had not fully internalized the consequences of that feature of Ratpack until I dug in for myself. My previous async programming had been NodeJS and Scala-based and I was using that as my mental model for how Ratpack would behave.

In those other systems, "async" and "parallel" were mostly interchangeable. If you need to make 3 async GET requests, you `map` over the urls and fire off the async requests. All of the requests are likely sent before any of them have responded.

Ratpack doesn't work this way. If you make 3 async GET requests, Ratpack will wait for the first one to be completed before sending off the second one.

To understand why this is, we need to discuss some of the details of Ratpack's architecture.

In Ratpack, Work is Done on One of Two Thread Pools

1. "Compute" Thread Pool

This thread pool is where all requests are handled, and are where all async, non-blocking code in your app executes. Under the covers, it is a Netty epoll EventLoopGroup, so it is very fast as long as you don't run any blocking operations on it (on non-linux boxes it uses NIO instead of epoll).

The compute thread pool size by default is `2 * # of CPUs`. Though you can easily change it with a config value:

```
ratpack {
    serverConfig {
        threads 8
        …
```

2. "Blocking" Thread Pool

The Blocking thread pool is unbounded in size (till you run out of memory). It uses a cached thread pool so that it can re-use previously threads when they are available again.

No work is done on a blocking thread unless you explicitly ask. Any blocking operations (like ones using traditional database drivers or file IO), should wrap their call in a `Blocking` method.

`Blocking` methods create a Java 8 `CompletableFuture` that is registered to notify a Ratpack Promise running on the current compute thread when the Future is completed.

It can be useful when testing async code to print out the `Thread.currentThread().name` to understand which thread your code is running on.

This simple Ratpack app uses the compute and blocking thread pools (full app):

```
  handlers {
    all { Context context ->

      println "A. Original compute thread: ${Thread.currentThread()

      Blocking.exec { ->
        context.render "hello from blocking" // pretend blocking wo
        println "B. Blocking thread : ${Thread.currentThread().name
      }

      println "C. Original compute thread: ${Thread.currentThread()
    }
  }
```

Which prints

```
A. Original compute thread: ratpack-compute-1-2
C. Original compute thread: ratpack-compute-1-2
B. Blocking thread : ratpack-blocking-3-1
```

This Blocking code is executed *after* the original thread, Ratpack detects that no response has yet been rendered in the original thread and that work has been scheduled on a thread in the blocking pool that it needs to register a callback for.

This code will always print `A … C … B`, serial behavior is guaranteed by Ratpack.

Requests are Processed in a Pipeline of Async Execution Segments

When a Ratpack app starts, it creates an ExecController which is in charge of running all the Execution segments during request processing.

If you do not have any asynchronous calls, each request will run in a single execution segment that runs on a compute thread.

If you *do* have asynchronous calls (including blocking calls which become asynchronous via `Blocking`), the request is processed in multiple execution segments, each of which is encapsulated in a Ratpack Promise. (full app)

```
handlers {
  all { Context context ->

    println "A. Original compute thread: ${Thread.currentThread()

    Promise.async { downstream ->
      println "B. Promise thread : ${Thread.currentThread().name}
      downstream.success("hello from async promise")
    }.then { result ->
      context.render result
    }

    println "C. Original compute thread: ${Thread.currentThread()
  }
}
```

The output shows that the async Promise runs after the original handler code, but it's execution stays on the same compute thread:

```
A. Original compute thread: ratpack-compute-1-2
C. Original compute thread: ratpack-compute-1-2
B. Promise thread : ratpack-compute-1-2
```

Registering an `ExecInterceptor` Lets You See the Segments of Execution

Ratpack allows you to register an `ExecInterceptor` to view the segments of execution (and create metrics).

If we create this `ExecInterceptor` that captures time at the execution and segment level:

```
public class LoggingExecInterceptor implements ExecInterceptor {
  @Override
  void intercept(Execution execution, ExecInterceptor.ExecType execT
    ExecutionTimer timer = ExecutionTimer.startExecutionSegment(exec
    try {
      executionSegment.execute()
    } finally {
      println "${Thread.currentThread().name} - $timer - ${execType}
    }
  }
}
```

and register it in this app:

```
  bindings {
    bindInstance(new LoggingExecInterceptor())
  }
  handlers {
    all { Context context ->
      final String executionId = context.get(ExecutionTimer).id.toSt

      println "${Thread.currentThread().name} - $executionId - A. Or
      context.render "hello from compute"
    }
  }
```

You'll see interceptor output with one `COMPUTE` thread `println` because we did not have any async or blocking work in our app, all of the work is done in a single execute segment:

```
ratpack-compute-1-4 - 7a265c2b-82b7-4c23-9f0d-92130fff5c26 - A. Ori
ratpack-compute-1-4 - 7a265c2b-82b7-4c23-9f0d-92130fff5c26 - segmen
```

Adding a blocking call to our app (full app):

```
    handlers {
     all { Context context ->
       final String executionId = context.get(ExecutionTimer).id.toSt

       println "${Thread.currentThread().name} - $executionId - A. Or

       Blocking.exec { ->
         context.render "hello from blocking" // pretend blocking wo
         println "${Thread.currentThread().name} - $executionId - B.
       }

       println "${Thread.currentThread().name} - $executionId - C. Or
     }
   }
```

gives us this output:

```
ratpack-compute-1-6 - f04d95cd-2043-47ae-8fc7-0600085eb399 - A. Ori
ratpack-compute-1-6 - f04d95cd-2043-47ae-8fc7-0600085eb399 - C. Ori
ratpack-compute-1-6 - f04d95cd-2043-47ae-8fc7-0600085eb399 - segment
ratpack-blocking-3-1 - f04d95cd-2043-47ae-8fc7-0600085eb399 - B. Bl
ratpack-blocking-3-1 - f04d95cd-2043-47ae-8fc7-0600085eb399 - segmen
ratpack-compute-1-6 - f04d95cd-2043-47ae-8fc7-0600085eb399 - segment
```

Notice that it adds an extra trailing `COMPUTE` execution after the `BLOCKING` one? Ratpack registered our `Blocking` call to notify an execution segment `Promise` on our original thread (`ratpack-compute-1-6`) when it was complete.

That kind of monitoring is how Ratpack knows when an execution is finished. If you spawn your own threads outside of a `Promise`, Ratpack has no idea that your work exists and you won't get the behavior that you're probably looking for.

Parallelized Code *Must* Notify the Original Compute Thread

Normally, running your non-blocking async work in serial fashion on the same compute thread is fast enough.

If you really want something to run in parallel, you can ask for that work to be scheduled on a different compute thread, but you have to notify the original thread that the work is complete (full app):

```
handlers {
  all { Context context ->
    final String executionId = context.get(ExecutionTimer).id.toS

    println "${Thread.currentThread().name} - $executionId - A. O

    Promise.async({ Downstream downstream ->
      println "${Thread.currentThread().name} - $executionId - B1

      // ask for an execution to be scheduled on another compute
      Execution.fork().start({ forkedExec ->
        println "${Thread.currentThread().name} - $executionId -
        downstream.success("hello from fork")
      })

      println "${Thread.currentThread().name} - $executionId - B2

    }).then { result ->
      println "${Thread.currentThread().name} - $executionId - D.
      context.render result
    }
  }
}
```

The output shows that the original compute thread has an execution segment that runs last. It is notified of the work that was done on that other thread by the call to `downstream.success`:

```
ratpack-compute-1-6 - edd2b6d3-54f2-43cf-87af-41fc6752cde5 - A. Ori
ratpack-compute-1-6 - edd2b6d3-54f2-43cf-87af-41fc6752cde5 - B1. Ins
ratpack-compute-1-6 - edd2b6d3-54f2-43cf-87af-41fc6752cde5 - B2. Aft
ratpack-compute-1-6 - edd2b6d3-54f2-43cf-87af-41fc6752cde5 - segmen
ratpack-compute-1-7 - edd2b6d3-54f2-43cf-87af-41fc6752cde5 - C. For
ratpack-compute-1-7 - 83505b78-455a-47c1-8012-486b163d587f - segmen
ratpack-compute-1-6 - edd2b6d3-54f2-43cf-87af-41fc6752cde5 - D. `the
ratpack-compute-1-6 - edd2b6d3-54f2-43cf-87af-41fc6752cde5 - segmen
```

Parallelizing Promise Streams

There isn't much built-in syntax sugar for working with parallelism using `Promise`s, partially because many apps don't need it. As of this blog post, there's an open issue on github to make this better in future versions of Ratpack.

If you need to parallelize your request handling right now, your best option is to use the RxJava integration. This makes RxJava `Observable`s work on top of Ratpack's Execution model.

RxJava/Promise Streams are Processed in Serial by Default

All work that ratpack does within an execution is on the same thread, and the work is fully serial. This has implications if you're trying to do something like make a microservice that makes HTTP requests to multiple back-end services for each request it receives.

Your requests will all be made one after the other, even though you are using fully non-blocking http APIs.

Demonstrating this takes a bit of setup. Here is stub embedded ratpack app that our application under test will use as a back-end service.

Each GET request to `http://localhost:<port>/:sleepFor>` will `sleep` and then return back to the caller. We `sleep` on a `Blocking` thread so we don't hold up our compute threads as `sleep` is blocking!

```
EmbeddedApp stubApp = GroovyEmbeddedApp.of {
  handlers {
    get(":sleepFor") {
      Integer sleepFor = context.pathTokens['sleepFor'].toInteger()
      Blocking.exec { ->
        println "Stub Sleep App GET Request, sleep for: $sleepFor se
        sleep(sleepFor * 1000)
        context.render sleepFor.toString()
      }
    }
  }
}
```

Our application under test will have an `Observable` stream of 3 `URI`s that will each do a non-blocking, async call to our stub sleep application above.

It will then collect the results from each request and render out a response to the original caller to the app (full app):

```
handlers {
  all { Context context ->
    HttpClient httpClient = context.get(HttpClient)
    final String executionId = context.get(ExecutionTimer).id.toSt

    // create a List of URIs to the stub app above that will ask 
    // for N seconds before returning the number of seconds it was
    final List<URI> REQUEST_SLEEP_URIS = [3, 2, 1].collect {
      URI.create("http://${stubApp.address.host}:${stubApp.address
    }

    println "${Thread.currentThread().name} - $executionId - A. Or

    // Iterate over all uris, make async http request for each and
    Observable.from(REQUEST_SLEEP_URIS) // stream of URIs
      .flatMap { uri ->
        println "${Thread.currentThread().name} - $executionId - 
        RxRatpack.observe(httpClient.get(uri))  // async http requ
      }
      .map { it.body.text } // get the body text for each http res
      .toList()             // collect into a single list and then
      .subscribe({ List<String> responses ->
        println "${Thread.currentThread().name} - $executionId - 
        context.render responses.join(", ")
      })
  }
}
```

We're asking for the requests in `REQUEST_SLEEP_URIS` to each sleep `3`, `2`, and `1` seconds before returning results. We can see from the output that it took slightly over 6 seconds (`3+2+1`) for our request to be fulfilled, and that stub app did not get the 2nd request till the execution segment for the first request had been completed.

```
ratpack-compute-1-4 - 64949f0f-6010-4eb0-abd7-b655769809e7 - A. Ori
ratpack-compute-1-4 - 64949f0f-6010-4eb0-abd7-b655769809e7 - B. GET
ratpack-compute-1-4 - 64949f0f-6010-4eb0-abd7-b655769809e7 - B. GET
ratpack-compute-1-4 - 64949f0f-6010-4eb0-abd7-b655769809e7 - B. GET
ratpack-compute-1-4 - 64949f0f-6010-4eb0-abd7-b655769809e7 - segment
Stub Sleep App GET Request, sleep for: 3 seconds
```

```
ratpack-compute-1-4 - 64949f0f-6010-4eb0-abd7-b655769809e7 - segmen
Stub Sleep App GET Request, sleep for: 2 seconds
ratpack-compute-1-4 - 64949f0f-6010-4eb0-abd7-b655769809e7 - segmen
Stub Sleep App GET Request, sleep for: 1 seconds
ratpack-compute-1-4 - 64949f0f-6010-4eb0-abd7-b655769809e7 - C. Sub
ratpack-compute-1-4 - 64949f0f-6010-4eb0-abd7-b655769809e7 - segmen
```

Also notice that all work in the app under test was done on the same `COMPUTE` thread: `ratpack-compute-1-4`.

This kind of behavior is a good default for Ratpack to have as it makes things very predictable and easy to reason about. There are cases though where you might really need additional performance for a single request.

Parallelism Must be Explicitly Requested

If you want your reactive stream to be processed in parallel, but the work is still async non-blocking work, you can add the `forkEach` and `bindExec` methods into your stream.

`forkEach` will schedule each observable value to be run on the next available compute thread.

`bindExec` works like a thread "join" operation. It converts the stream into a Ratpack `Promise` and then back into an observable. This brings processing of that value back to the original thread. If you don't include an explicit `bindExec`, Ratpack will take care of bringing the execution back to the main thread for the subscriber automatically.

If we add `forkEach` and `bindExec` into our stream from above (full app):

```
Observable.from(REQUEST_SLEEP_URIS)
  .forkEach()            // <-- run in parallel on different compute
  .flatMap { uri ->
    println "${Thread.currentThread().name} - $executionId - B. GET
    RxRatpack.observe(httpClient.get(uri))
  }
  .map { it.body.text }
  .bindExec()            // <-- bind forked thread results to origin
  .toList()
  .subscribe({ List<String> responses ->
    println "${Thread.currentThread().name} - $executionId - C. Sub
    context.render responses.join(", ")
  })
```

You'll see that our request time reduces to slightly over 3 seconds, the longest sleep time that we were using:

```
ratpack-compute-1-7 - 6537dfd0-732a-4599-b82c-7f48bf1c5a42 - A. Ori
ratpack-compute-1-7 - 6537dfd0-732a-4599-b82c-7f48bf1c5a42 - segmen
ratpack-compute-1-8 - 6537dfd0-732a-4599-b82c-7f48bf1c5a42 - B. GET
ratpack-compute-1-9 - 6537dfd0-732a-4599-b82c-7f48bf1c5a42 - B. GET
ratpack-compute-1-10 - 6537dfd0-732a-4599-b82c-7f48bf1c5a42 - B. GE
ratpack-compute-1-9 - 41bb05d8-82b8-45bc-8f2b-b3f9330ee61a - segmen
ratpack-compute-1-8 - 9d0a9729-641a-416b-bd1b-cf04e1aa16b1 - segmen
ratpack-compute-1-10 - cd2b08e4-625c-48a3-8141-227c6b496ae4 - segme
Stub Sleep App GET Request, sleep for: 3 seconds
Stub Sleep App GET Request, sleep for: 2 seconds
Stub Sleep App GET Request, sleep for: 1 seconds
ratpack-compute-1-10 - cd2b08e4-625c-48a3-8141-227c6b496ae4 - segme
ratpack-compute-1-9 - 41bb05d8-82b8-45bc-8f2b-b3f9330ee61a - segmen
ratpack-compute-1-7 - 6537dfd0-732a-4599-b82c-7f48bf1c5a42 - C. Sub
ratpack-compute-1-8 - 9d0a9729-641a-416b-bd1b-cf04e1aa16b1 - segmen
ratpack-compute-1-7 - 6537dfd0-732a-4599-b82c-7f48bf1c5a42 - segmen
```

We've gone from 4 execution segments in the original (serial) execution to 8 execution segments (3 more for forking each URI onto the new compute threads, and one for collecting the returned results).

Asking for parallel execution of your streams means that a single request could be handled more quickly, but you are likely reducing the number of transactions per second that your app can handle.

You shouldn't parallelize your code without first running performance and load tests to determine that you get an actual boost.

You are also giving up some of the ordering guarantees that Ratpack gives you as a default, it can make your code harder to reason about, but only within the forked part of the stream.

Other Notes About RxJava/Ratpack

If you're familiar with RxJava, you might have seen information on using a Scheduler along with the `scheduleOn` and `observeOn` methods. We don't have direct access to a scheduler of ratpack's compute/blocking thread pools so we can't use these methods to get our work done in parallel. Currently, `forkEach`/`bindExec` is the best way to get your `Observable` code to run in parallel.

Understanding Ratpack Executions for Yourself

If you're new to async/non-blocking programming, there will be a bit of a learning curve. Even if you've worked with previous frameworks, every framework has it's own behavior that you need to learn, and Ratpack is no different. I've linked to full running groovy scripts for each of the sample applications above. I think the best way to internalize how Ratpack works is to dive in and play around with some examples for yourself.

Hopefully this post has helped given you some tools and places to start exploring for yourself.

I'd also highly recommend joining the Ratpack Slack Channel, I've gotten a huge amount of help from Ratpack team members as well as others in the community. Simply lurking there has been extremely valuable, and I've always gotten a great response to getting my questions answered.

tednaleid
tednaleid