UNCATEGORIZED

Spring Boot: Tuning your Undertow application for throughput

April 26, 2016 by Jakub Narloch

It's been some time since the previous blog post, but finally I though that it's a good time to make a post about very useful and practical aspect. How to prepare your Spring Boot application for production and how to guarantee that it will be able to handle a couple of millions of views each day.

If you think that you have already made all the needed steps by making your application stateless, scaling it out or running it on the high end machine, think twice because it's quite likely that there are some bottlenecks inside your application that if not treated with proper attention would most likely degradate the performance and the application overall throughput.

Tuning for latency vs tunning for throughput.

Interesting enough in the past, being aware of the Little's Law I have thought that tuning your application throughput requires nothing more then reducing your application latency as much as possible. It was just after reading the book Java Performance (https://www.amazon.com/Java-Performance-Charlie-Hunt/dp/0137142528/ref=sr_1_2?ie=UTF8&qid=1466019141&sr=8-2&keywords=Java+Performance) the after I realized that might not be true in all of the cases.

Generally you can improve the latency first by improving your application algorithmic performance, after that you should take a look on access patterns in your application introducing a caching layer or redesign the way your application is accessing the data can have huge impact on the overall performance. If your application is heavely I/O bound performing operations in the parallel can be a way to improve things a bit.

Also a good idea for improving you application latency is to configure asynchronous logging whether you using Logback or Log4J2, but of them provide proper functionality.

Thread pools

Undertow

Undertow uses XNIO as the default connector. XNIO has some interesting characteristics apart from the default configuration which by default is I/O threads initialized to the number of your logical threads and the worker thread equal to 8 * CPU cores. So on typical 4 cores Intel CPU with hypert-hreading you will end up with 8 I/O threads and 64 working threads. Is this enough? Well, it depends. Considerably the Tomcat's and Jetty defaults are 100 and 1000 threads respectively. If you need to be able to handle more request per second this is the first thing that need to consider to increase.

Hystrix

The Hystrix documentation states that:

> Most of the time the default value of 10 threads will be fine (often it could be made smaller).

After working with couple of the projects, I found it hardly to believe that this could be a true statement. The defaults for Hystrix is 10 threads per pool, which quickly might turn out to become a bottleneck. In fact the same documentation also states that in other to establish the correct size of hysterix thread pool you should use the fallowing formula:

> *requests per second at peak when healthy × 99th percentile latency in seconds + some breathing room*

So let's assume that you have a system that has to handle let's say 24 000 rps, divided by the number of instances, for instance 8, you can establish the appropriate pool size for single instance. This will vary greatly on the latency of your system.

RxJava

Memory usage

All of this is not given without a price. Each of the newly allocated threads consumes memory. Through Java you can configure this property through **-Xss** property with the default for 64 bit VM being 1 MB. So if you let's say configure your Undertow thread pool with 512 working threads, be ready that your memory consumption (only for allocating the thread stacks) will be increased to that number.

Connection pools

HTTP

Do you use for instance RestTemplate, or maybe RestEasy JAX-RS client. In fact there is a well known issue reported in RestEasy that uses exactly ONE connection for all of your calls. The good advice is to align that value with the number of working threads of your application server, otherwise when

performing the HTTP calls the threads will be waiting for acquiring the underlying HTTP connection from the pool, which will cause unnecessary and probably unintended delay.

Cache

The same basic principal applies to any other kind of service that is being communicated over TCP connection. For instance Memcached clients like XMemcache has nice capabilities of using a multiplexed TCP connection with binary protocol on top of it, giving a throughput of roughly 50 requests per connection, though still if you need to be able to handle greater throughput you need to configure your client to maintain a entire pool of connections.

Garbage collection

If you opt for low latency, probably you should consider optimizing the Garbage Collector as the last kind of resort. As much as garbage collection could be optimized through different settings this does not handles the true problem, if you can address those issue first you should be able to be just find and tune the garbage collector afterwards for the best overall performance.

Final thoughts

Equipped with this practical knowledge how you will be able to tell if your application is your application faces any of those problems, first of all equipped with proper tools. Stress test are one of them, you can either decide to treat the application as a black box and use for instance Gatling to measure the throughput of your application, if you need need more fined grained tools the jmh project that could used for running benchmarks of the individual Java methods. Finally use profiler to understand where your application is spending the most time, is it for instance a RestTemplate call, or maybe your cache access time sky rockets whenever you? A good advice on how to measure the characteristics of the application is to use the doubling approach, run your benchmark with for instance 64 RPS – monitor the results, and repeat the experiment with double the request number. Continue as long as you haven't reached the desired throughput level.

With all of this being said, the true is that this in fact describes the hard way, there is also a simple and fast path to solve your heavy load problems especially for HTTP:

Use a caching reverse proxy.

Either if it's Nginx or Varnish, both of them should take the load out of your backing services and if you can decrease your load you do not need spend so much time on the optimizations.

One comment

Orlando · June 30, 2016
Hi,

I believe (and I might be wrong since it has been a few months since I last worked with Hystrix) that Hystrix's threadpool is shared by a Hystrix Group. A Hystrix Command Group would normally includes multiple Hystrix Commands and that 10-threads (default value) applies to each Group, other Hystrix Groups would use a different thread pool and it could be configured individually, and yes, Netflix claims (at the time I read about this) that their biggest Hystrix Group thread pool is about 30 threads.
Even if command.execute() is called (blocking call), it would run in a thread from the group's thread pool, via command.queue().get() which blocks until the Future completes.
My guess is that at this point, it would be more important properly group Hystrix commands and tweak each Hystrix Group threadpool, queue size, timeout, … than container threadpool.
Anyway, good blog, keep it up.

Reply