written by Jamie Perkins on 17 February 2016 in engineering, performance

java socket programming with netty

It's (hopefully) quite infrequent that one needs to work with network sockets directly to chuck bytes around. Normally in an application you'll use an existing application–level protocol like REST over HTTP to pass data around. The reasons for this include, but are not limited to; convenience, reliability, interoperability and sanity.

That said, should you find yourself in a position where you need better performance or more flexibility than an existing protocol, it's useful to know where to start.

For example, I used it recently in an IoT project where it would have been time consuming and inefficient to deal with HTTP clients in embedded C++ code.

netty

Netty is an NIO (non–blocking input/output) client–server framework for Java. It simplifies the process of writing servers and clients that talk to each other under the hood using your typical `DatagramSocket`, `ServerSocket` and `Socket` classes. In this example I'll show you how to write a very simple server that will accept connections over a TCP port, read and decode JSON and do something with it.

In real life you're probably more likely to use something binary like Thrift, Protocol Buffers or Smile, instead of JSON.

getting started

I am assuming you have imported Netty using the dependency manager of your choice and are ready to start typing code.

First off we need instances of `NioEventLoopGroup`. This class implements a multi-threaded Event Loop, that is, something that constantly and frequently polls IO abstractions for stuff to do like read data or start a new connection. There is also the `EpollEventLoopGroup` available if you're on Linux, which makes use of the more performant Epoll.

We need two of them, one to accept new connections and one to handle existing connections. If you've worked with an HTTP server you'll know it typically uses the same thing.

configuring the server

Next we must configure the server proper. Lets walk through.

`ServerBootstrap` is a helper of sorts that lets you avoid configuring every single aspect of the highly complex `ServerChannel` implementations. Basically does what it says on the tin, it bootstraps a server for us.

It needs setting up with a few things, first we give it the event loops we created earlier which allows our server to accept and handle connections.

Next is a call to `.channel()` with a class. Netty will creates instances of this class and uses them to accept new connections. In this case that's `NioServerSocketChannel` which is an implementation of `ServerChannel`.

Then a call to `.childHandler()` with an instance of `ChannelHandler`. This is where interesting things will happen, it sets up the pipeline that accepted connections are handled through. Here I'm using a class called `MySocketInitialiser`, my own creation, we'll come back to this.

Calls to `.option()` let us set server–related TCP options. In this case `SO_BACKLOG` tells the server to refuse connections if it already has 5 queued up.

Finally calls to `.childOption()` let us set client–related TCP options. `SO_KEEPALIVE` tells clients to keep their connections open with keepalive packets.

We then start the server by telling it to bind to a port at the local address and call `.sync()` to wait for the server to shutdown.

setting up a pipeline

Back to `MySocketInitialiser` to see where the magic happens.

The `initChannel()` method of this class is called by Netty whenever it receives a new connection. A `SocketChannel` is simply the channel abstraction over a TCP/IP socket.

Each `SocketChannel` has a pipeline associated with it. You can think of think of the pipeline as an ordered list of handlers with each feeding its output as the input to the next one. There are caveats to this but we can ignore them for now.

In the example pipeline above we have in order the following:

1. A `LineBasedFrameDecoder`, this delimits messages by detecting newlines bytes (i.e. \n or \r\n)
2. A `StringDecoder`, this decodes bytes into UTF-8 `String`s or any other encoding of your choice
3. A `JsonDecoder`, this decodes `String`s using Gson into objects of type `Person` or any other type of your choice
4. An anonymous class that simply prints the name of our decoded `Person` to standard output

`JsonDecoder` is not a part of Netty, its implementation is as follows:

That's everything we need for our example Netty server to do stuff.

seeing it in action

First we start up the server. How you do this will depend on the way your project is structured:

We can then use `telnet` to open a socket to our server:

And then in standard out we'll see:

wrapping up

There is a huge amount of detail I've glossed over for the sake of making this a very easy introduction and to get you off the ground quickly. The Netty user guide goes into more depth and is a good place to start when learning more. If you want to read about NIO in general the Oracle docs are also helpful.

If you enjoyed the read, drop us a comment below or share the article, follow us on Twitter or subscribe to our #MetaBeers newsletter. Before you go, grab a PDF of the article, and let us know if it's time we worked together.