03 引导器作用：客户端和服务端启动都要做些什么?

你好，我是若地。上节课我们介绍了 Netty 中核心组件的作用以及组件协作的方式方法。从这节课开始，我们将对 Netty 的每个核心组件依次进行深入剖析解读。我会结合相应的代码示例讲解，帮助你快速上手 Netty。

我们在使用 Netty 编写网络应用程序的时候，一定会从**引导器 Bootstrap**开始入手。Bootstrap 作为整个 Netty 客户端和服务端的**程序入口**，可以把 Netty 的核心组件像搭积木一样组装在一起。本节课我会从 Netty 的引导器**Bootstrap**出发，带你学习如何使用 Netty 进行最基本的程序开发。

从一个简单的 HTTP 服务器开始

HTTP 服务器是我们平时最常用的工具之一。同传统 Web 容器 Tomcat、Jetty 一样，Netty 也可以方便地开发一个 HTTP 服务器。我从一个简单的 HTTP 服务器开始，通过程序示例为你展现 Netty 程序如何配置启动，以及引导器如何与核心组件产生联系。

完整地实现一个高性能、功能完备、健壮性强的 HTTP 服务器非常复杂，本文仅为了方便理解 Netty 网络应用开发的基本过程，所以只实现最基本的**请求-响应**的流程：

1. 搭建 HTTP 服务器，配置相关参数并启动。
2. 从浏览器或者终端发起 HTTP 请求。
3. 成功得到服务端的响应结果。

Netty 的模块化设计非常优雅，客户端或者服务端的启动方式基本是固定的。作为开发者来说，只要照葫芦画瓢即可轻松上手。大多数场景下，你只需要实现与业务逻辑相关的一系列 ChannelHandler，再加上 Netty 已经预置了 HTTP 相关的编解码器就可以快速完成服务端框架的搭建。所以，我们只需要两个类就可以完成一个最简单的 HTTP 服务器，它们分别为**服务器启动类**和**业务逻辑处理类**，结合完整的代码实现我将对它们分别进行讲解。

服务端启动类

所有 Netty 服务端的启动类都可以采用如下代码结构进行开发。简单梳理一下流程：首先创建引导器；然后配置线程模型，通过引导器绑定业务逻辑处理器，并配置一些网络参数；最后绑定端口，就可以完成服务器的启动了。

```
public class HttpServer {

    public void start(int port) throws Exception {

        EventLoopGroup bossGroup = new NioEventLoopGroup();

        EventLoopGroup workerGroup = new NioEventLoopGroup();

        try {

            ServerBootstrap b = new ServerBootstrap();

            b.group(bossGroup, workerGroup)

                    .channel(NioServerSocketChannel.class)

                    .localAddress(new InetSocketAddress(port))

                    .childHandler(new ChannelInitializer<SocketChannel>() {

                        @Override

                        public void initChannel(SocketChannel ch) {

                            ch.pipeline()

                                    .addLast("codec", new HttpServerCodec())

                                    .addLast("compressor", new HttpContentCompresso

                                    .addLast("aggregator", new HttpObjectAggregator

                                    .addLast("handler", new HttpServerHandler());

                        }

                    })

                    .childOption(ChannelOption.SO_KEEPALIVE, true);

            ChannelFuture f = b.bind().sync();

            System.out.println("Http Server started,  Listening on " + port);

            f.channel().closeFuture().sync();

        } finally {

            workerGroup.shutdownGracefully();
```

```
            bossGroup.shutdownGracefully();
        }
    }
    public static void main(String[] args) throws Exception {
        new HttpServer().start(8088);
    }
}
```

服务端业务逻辑处理类

如下代码所示，HttpServerHandler 是业务自定义的逻辑处理类。它是入站 ChannelInboundHandler 类型的处理器，负责接收解码后的 HTTP 请求数据，并将请求处理结果写回客户端。

```
public class HttpServerHandler extends SimpleChannelInboundHandler<FullHttpRequest>
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, FullHttpRequest msg) {
        String content = String.format("Receive http request, uri: %s, method: %s,
        FullHttpResponse response = new DefaultFullHttpResponse(
                HttpVersion.HTTP_1_1,
                HttpResponseStatus.OK,
                Unpooled.wrappedBuffer(content.getBytes()));
        ctx.writeAndFlush(response).addListener(ChannelFutureListener.CLOSE);
    }
}
```

通过上面两个类，我们可以完成 HTTP 服务器最基本的请求-响应流程，测试步骤如下：

1. 启动 HttpServer 的 main 函数。
2. 终端或浏览器发起 HTTP 请求。

测试结果输出如下：

```
$ curl http://localhost:8088/abc

$ Receive http request, uri: /abc, method: GET, content:
```

当然，你也可以使用 Netty 自行实现 HTTP Client，客户端和服务端的启动类代码十分相似，我在附录部分提供了一份 HTTPClient 的实现代码仅供大家参考。

通过上述一个简单的 HTTP 服务示例，我们基本熟悉了 Netty 的编程模式。下面我将结合这个例子对 Netty 的引导器展开详细的介绍。

引导器实践指南

Netty 服务端的启动过程大致分为三个步骤：

1. **配置线程池；**
2. **Channel 初始化；**
3. **端口绑定。**

下面，我将逐一为大家介绍每一步具体需要做哪些工作。

配置线程池

Netty 是采用 Reactor 模型进行开发的，可以非常容易切换三种 Reactor 模式：**单线程模式、多线程模式、主从多线程模式。**

单线程模式

Reactor 单线程模型所有 I/O 操作都由一个线程完成，所以只需要启动一个 EventLoopGroup 即可。

```
EventLoopGroup group = new NioEventLoopGroup(1);

ServerBootstrap b = new ServerBootstrap();

b.group(group)
```

多线程模式

Reactor 单线程模型有非常严重的性能瓶颈，因此 Reactor 多线程模型出现了。在 Netty 中使用 Reactor 多线程模型与单线程模型非常相似，区别是 NioEventLoopGroup 可以不需要

任何参数，它默认会启动 2 倍 CPU 核数的线程。当然，你也可以自己手动设置固定的线程数。

```
EventLoopGroup group = new NioEventLoopGroup();

ServerBootstrap b = new ServerBootstrap();

b.group(group)
```

主从多线程模式

在大多数场景下，我们采用的都是**主从多线程 Reactor 模型**。Boss 是主 Reactor，Worker 是从 Reactor。它们分别使用不同的 NioEventLoopGroup，主 Reactor 负责处理 Accept，然后把 Channel 注册到从 Reactor 上，从 Reactor 主要负责 Channel 生命周期内的所有 I/O 事件。

```
EventLoopGroup bossGroup = new NioEventLoopGroup();

EventLoopGroup workerGroup = new NioEventLoopGroup();

ServerBootstrap b = new ServerBootstrap();

b.group(bossGroup, workerGroup)
```

从上述三种 Reactor 线程模型的配置方法可以看出：Netty 线程模型的可定制化程度很高。它只需要简单配置不同的参数，便可启用不同的 Reactor 线程模型，而且无需变更其他的代码，很大程度上降低了用户开发和调试的成本。

Channel 初始化

设置 Channel 类型

NIO 模型是 Netty 中最成熟且被广泛使用的模型。因此，推荐 Netty 服务端采用 NioServerSocketChannel 作为 Channel 的类型，客户端采用 NioSocketChannel。设置方式如下：

```
  b.channel(NioServerSocketChannel.class);
```

当然，Netty 提供了多种类型的 Channel 实现类，你可以按需切换，例如 OioServerSocketChannel、EpollServerSocketChannel 等。

注册 ChannelHandler

在 Netty 中可以通过 ChannelPipeline 去注册多个 ChannelHandler，每个 ChannelHandler 各司其职，这样就可以实现最大化的代码复用，充分体现了 Netty 设计的优雅之处。那么如何通过引导器添加多个 ChannelHandler 呢？其实很简单，我们看下 HTTP 服务器代码示例：

```
b.childHandler(new ChannelInitializer<SocketChannel>() {
    @Override
    public void initChannel(SocketChannel ch) {
        ch.pipeline()
                .addLast("codec", new HttpServerCodec())
                .addLast("compressor", new HttpContentCompressor())
                .addLast("aggregator", new HttpObjectAggregator(65536))
                .addLast("handler", new HttpServerHandler());
    }
})
```

ServerBootstrap 的 childHandler() 方法需要注册一个 ChannelHandler。**ChannelInitializer**是实现了 ChannelHandler**接口的匿名类**，通过实例化 ChannelInitializer 作为 ServerBootstrap 的参数。

Channel 初始化时都会绑定一个 Pipeline，它主要用于服务编排。Pipeline 管理了多个 ChannelHandler。I/O 事件依次在 ChannelHandler 中传播，ChannelHandler 负责业务逻辑处理。上述 HTTP 服务器示例中使用链式的方式加载了多个 ChannelHandler，包含 **HTTP 编解码处理器、HTTPContent 压缩处理器、HTTP 消息聚合处理器、自定义业务逻辑处理器。**

在以前的章节中，我们介绍了 ChannelPipeline 中**入站 ChannelInboundHandler**和**出站 ChannelOutboundHandler**的概念，在这里结合 HTTP 请求-响应的场景，分析下数据在 ChannelPipeline 中的流向。当服务端收到 HTTP 请求后，会依次经过 HTTP 编解码处理器、HTTPContent 压缩处理器、HTTP 消息聚合处理器、自定义业务逻辑处理器分别处理后，再将最终结果通过 HTTPContent 压缩处理器、HTTP 编解码处理器写回客户端。

设置 Channel 参数

Netty 提供了十分便捷的方法，用于设置 Channel 参数。关于 Channel 的参数数量非常多，如果每个参数都需要自己设置，那会非常繁琐。幸运的是 Netty 提供了默认参数设置，实际场景下默认参数已经满足我们的需求，我们仅需要修改自己关系的参数即可。

```
b.option(ChannelOption.SO_KEEPALIVE, true);
```

ServerBootstrap 设置 Channel 属性有**option**和**childOption**两个方法，option 主要负责设置 Boss 线程组，而 childOption 对应的是 Worker 线程组。

这里我列举了经常使用的参数含义，你可以结合业务场景，按需设置。

参数	**含义**
SO_KEEPALIVE	设置为 true 代表启用了 TCP SO_KEEPALIVE 属性，TCP 会主动探测连接状态，即连接保活
SO_BACKLOG	已完成三次握手的请求队列最大长度，同一时刻服务端可能会处理多个连接，在高并发海量连接的场景下，该参数应适当调大
TCP_NODELAY	Netty 默认是 true，表示立即发送数据。如果设置为 false 表示启用 Nagle 算法，该算法会将 TCP 网络数据包累积到一定量才会发送，虽然可以减少报文发送的数量，但是会造成一定的数据延迟。Netty 为了最小化数据传输的延迟，默认禁用了 Nagle 算法
SO_SNDBUF	TCP 数据发送缓冲区大小
SO_RCVBUF	TCP数据接收缓冲区大小，TCP数据接收缓冲区大小
SO_LINGER	设置延迟关闭的时间，等待缓冲区中的数据发送完成
CONNECT_TIMEOUT_MILLIS	建立连接的超时时间

端口绑定

在完成上述 Netty 的配置之后，bind() 方法会真正触发启动，sync() 方法则会阻塞，直至整个启动过程完成，具体使用方式如下：

```
ChannelFuture f = b.bind().sync();
```

bind() 方法涉及的细节比较多，我们将在《源码篇：从 Linux 出发深入剖析服务端启动流程》课程中做详细地解析，在这里就先不做展开了。

关于如何使用引导器开发一个 Netty 网络应用我们就介绍完了，服务端的启动过程一定离不开配置线程池、Channel 初始化、端口绑定三个步骤，在 Channel 初始化的过程中最重要的就是绑定用户实现的自定义业务逻辑。是不是特别简单？你可以参考本节课的示例，自己尝试开发一个简单的程序练练手。

总结

本节课我们围绕 Netty 的引导器，学习了如何开发最基本的网络应用程序。引导器串接了 Netty 的所有核心组件，通过引导器作为学习 Netty 的切入点有助于我们快速上手。Netty 的引导器作为一个非常方便的工具，避免我们再去手动完成繁琐的 Channel 的创建和配置等过程，其中有很多知识点可以深挖，在后续源码章节中我们再一起探索它的实现原理。

附录

HTTP 客户端类

```
public class HttpClient {

    public void connect(String host, int port) throws Exception {

        EventLoopGroup group = new NioEventLoopGroup();

        try {

            Bootstrap b = new Bootstrap();

            b.group(group);

            b.channel(NioSocketChannel.class);

            b.option(ChannelOption.SO_KEEPALIVE, true);

            b.handler(new ChannelInitializer<SocketChannel>() {

                @Override
```

```
                public void initChannel(SocketChannel ch) {
                    ch.pipeline().addLast(new HttpResponseDecoder());
                    ch.pipeline().addLast(new HttpRequestEncoder());
                    ch.pipeline().addLast(new HttpClientHandler());
                }
            });
            ChannelFuture f = b.connect(host, port).sync();
            URI uri = new URI("http://127.0.0.1:8088");
            String content = "hello world";
            DefaultFullHttpRequest request = new DefaultFullHttpRequest(HttpVersion
                    uri.toASCIIString(), Unpooled.wrappedBuffer(content.getBytes(St
            request.headers().set(HttpHeaderNames.HOST, host);
            request.headers().set(HttpHeaderNames.CONNECTION, HttpHeaderValues.KEEP
            request.headers().set(HttpHeaderNames.CONTENT_LENGTH, request.content()
            f.channel().write(request);
            f.channel().flush();
            f.channel().closeFuture().sync();
        } finally {
            group.shutdownGracefully();
        }
    }
    public static void main(String[] args) throws Exception {
        HttpClient client = new HttpClient();
        client.connect("127.0.0.1", 8088);
    }
}
```

客户端业务处理类

```
public class HttpClientHandler extends ChannelInboundHandlerAdapter {

    @Override

    public void channelRead(ChannelHandlerContext ctx, Object msg) {

        if (msg instanceof HttpContent) {

            HttpContent content = (HttpContent) msg;

            ByteBuf buf = content.content();

            System.out.println(buf.toString(io.netty.util.CharsetUtil.UTF_8));

            buf.release();

        }

    }

}
```

上一页 下一页