NIO 网络编程框架 Netty

课程讲义

主讲：Reythor 雷

2019

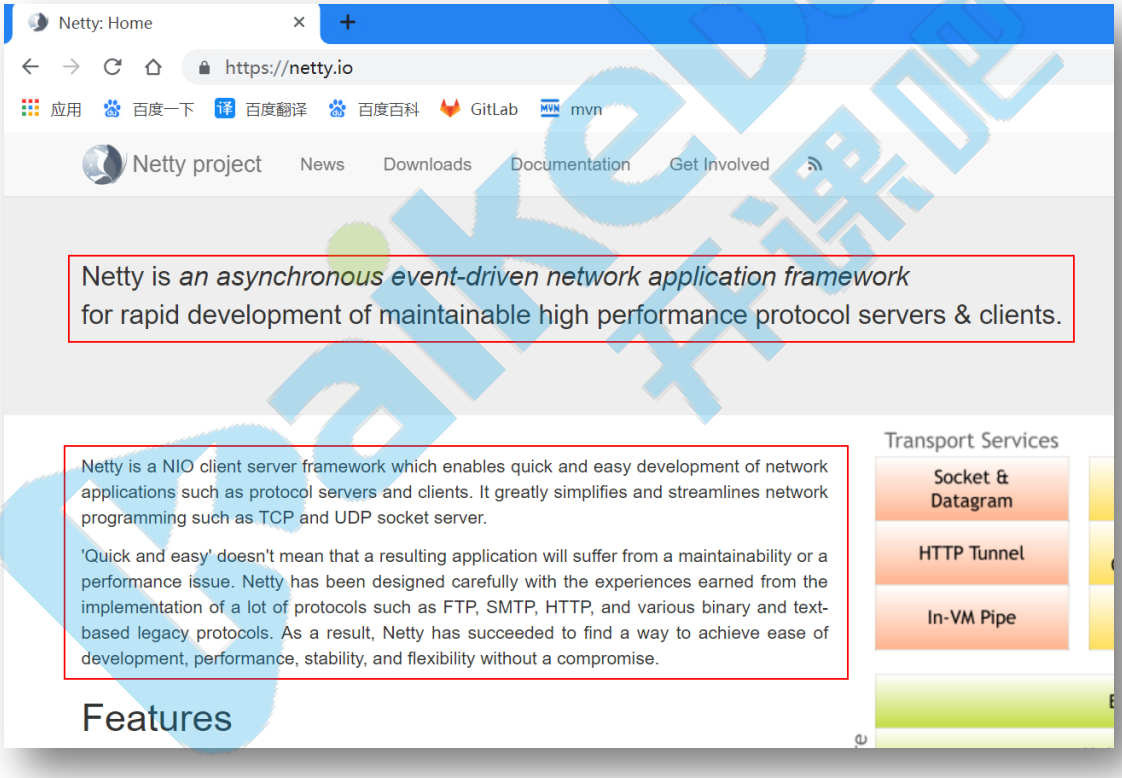
专题： NIO 网络编程框架 Netty

第1章 Netty 入门

1.1 Netty 概述

1.1.1 Netty 简介

Netty 官网上可以看到最权威的介绍：

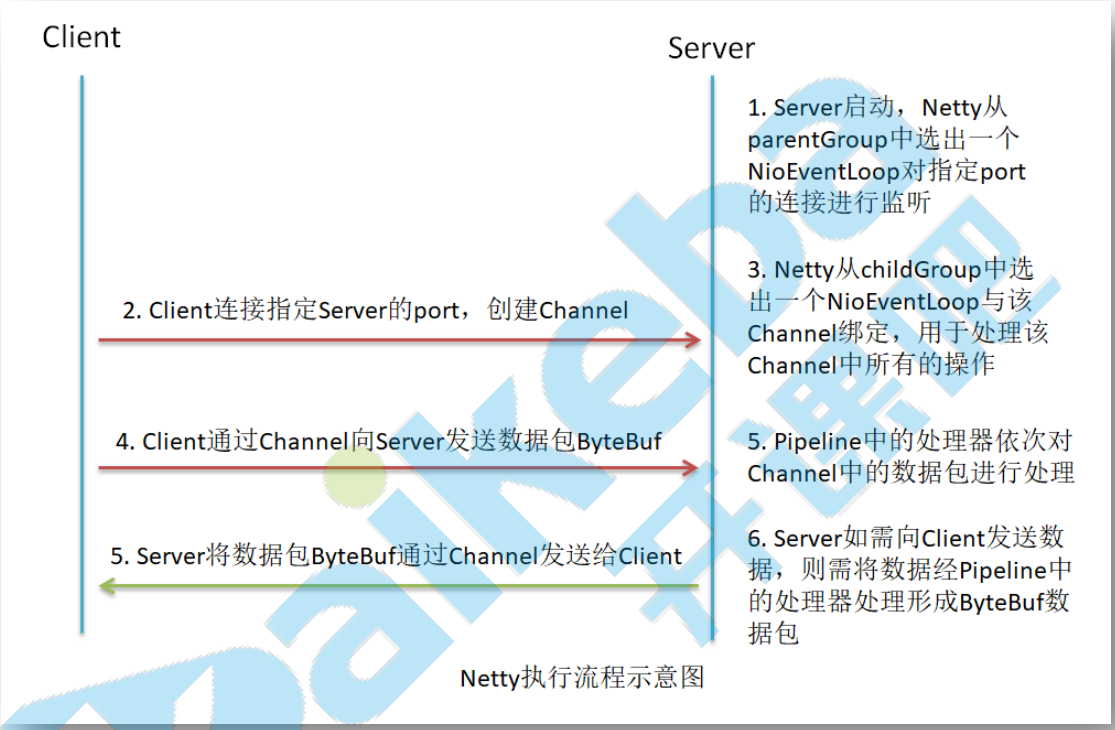


- Netty 是一个异步事件驱动的网络应用程序框架，用于快速开发可维护的高性能服务器和客户端。
- Netty 是一个 NIO 客户机-服务器框架，它支持快速、简单地开发网络应用程序，如服务器和客户机。它大大简化了网络编程，如 TCP 和 UDP 套接字服务器。
- “快速和简单”并不意味着生成的应用程序将受到可维护性或性能问题的影响。Netty 经过精心设计，并积累了许多协议（如 ftp、smtp、http）的实施经验，以及各种二进制和基于文本的遗留协议。因此，Netty 成功地找到了一种方法，在不妥协的情况下实现了易于开发、性能、稳定性和灵活性。

1.1.2 谁在使用 Netty

Dubbo、zk、RocketMQ、ElasticSearch、Spring5(对 HTTP 协议的实现)、GRpc、Spark 等大型开源项目都在使用 Netty 作为底层通讯框架。

1.1.3 Netty 执行流程



Netty执行流程示意图

1.1.4 Netty 中的核心概念

（1） Channel

管道，其是对 Socket 的封装，其包含了一组 API，大大简化了直接与 Socket 进行操作的复杂性。

（2） EventLoopGroup

EventLoopGroup 是一个 EventLoop 线程池，包含很多的 EventLoop。
Netty 为每个 Channel 分配了一个 EventLoop，用于处理用户连接请求、对用户请求的处

理等所有事件。EventLoop 本身只是一个线程驱动，在其生命周期内只会绑定一个线程，让该线程处理一个 Channel 的所有 IO 事件。

一个 Channel 一旦与一个 EventLoop 相绑定，则在 Channel 的整个生命周期内是不会也不能发生变化。但一个 EventLoop 可以与多个 Channel 相绑定。

（3） ServerBootStrap

用于配置整个 Netty 代码，将各个组件关联起来。服务端使用的是 ServerBootStrap，而客户端使用的是则 BootStrap。

（4） ChannelHandler 与 ChannelPipeline

ChannelHandler 是对 Channel 中数据的处理器，这些处理器可以是系统本身定义好的编解码器，也可以是用户自定义的。这些处理器会被统一添加到一个 ChannelPipeline 的对象中，然后按照添加的顺序对 Channel 中的数据进行依次处理。

（5） ChannelFuture

Netty 中所有的 I/O 操作都是异步的，即操作不会立即得到返回结果，所以 Netty 中定义了一个 ChannelFuture 对象作为这个异步操作的“代言人”，表示异步操作本身。如果想获取到该异步操作的返回值，可以通过该异步操作对象的 addListener()方法为该异步操作添加监听器，为其注册回调：当结果出来后马上调用执行。

Netty 的异步编程模型都是建立在 Future 与回调概念之上的。

1.2 牛刀小试 01-primary

通过该程序达到的目的是，对 Netty 编程的基本结构及流程有所了解。

该程序是通过 Netty 实现 HTTP 请求的处理，即接收 HTTP 请求，返回 HTTP 响应。

这个代码相当于“SpringMVC + Tomcat”

1.2.1 创建工程

创建一个普通的 Maven 的 Java 工程。

1.2.2 导入依赖

仅导入一个 netty-all 依赖即可。

```
<dependencies>
    <!-- netty-all 依赖 -->
```

```
    <dependency>
        <groupId>io.netty</groupId>
        <artifactId>netty-all</artifactId>
        <version>4.1.36.Final</version>
    </dependency>
    <!--Lombok 依赖-->
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.6</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

1.2.3 定义服务器启动类

该服务器就是用于创建并初始化服务器启动对象 ServerBootStrap。

```
public class HttpServer {
    public static void main(String[] args)
                                throws InterruptedException {
        EventLoopGroup parentGroup = new NioEventLoopGroup();
        EventLoopGroup childGroup = new NioEventLoopGroup();

        try {
            ServerBootstrap bootstrap = new ServerBootstrap();

            bootstrap.group(parentGroup, childGroup)
                    .channel(NioServerSocketChannel.class)
                    .childHandler(new HttpChannelInitializer());
```

```
            ChannelFuture future = bootstrap.bind(8888).sync();
            future.channel().closeFuture().sync();

        } finally {
            parentGroup.shutdownGracefully();
            childGroup.shutdownGracefully();
        }
    }
}
```

1.2.4 定义管道初始化器

```
public class HttpChannelInitializer
                    extends ChannelInitializer<SocketChannel> {
    @Override
    protected void initChannel(SocketChannel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        pipeline.addLast("httpServerCodec", new HttpServerCodec());
        pipeline.addLast("httpServerHandler", new HttpServerHandler());
    }
}
```

1.2.5 定义服务端处理器

```
public class HttpServerHandler extends ChannelInboundHandlerAdapter {

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {

        if(msg instanceof HttpRequest) {
            HttpRequest request = (HttpRequest) msg;
            System.out.println("请求方式：" + request.method().name());
            System.out.println("请求URI：" + request.uri());

            if("/favicon.ico".equals(request.uri())) {
                System.out.println("不处理/favicon.ioc请求");
                return;
            }
```

```
            ByteBuf content =
                    Unpooled.copiedBuffer("Hello Netty World", CharsetUtil.UTF_8);
            DefaultFullHttpResponse response = new DefaultFullHttpResponse(
                            HttpVersion.HTTP_1_1, HttpResponseStatus.OK, content);

            HttpHeaders headers = response.headers();
            headers.set(HttpHeaderNames.CONTENT_TYPE, "text/plain");
            headers.set(HttpHeaderNames.CONTENT_LENGTH, content.readableBytes());

            ctx.writeAndFlush(response)
               .addListener(ChannelFutureListener.CLOSE);
        }
    }
```

```
    @Override
    public void exceptionCaught(ChannelHandlerContext ctx,
                                Throwable cause) throws Exception {
        cause.printStackTrace();
        ctx.close();
    }
}
```

1.3 Socket 编程 02-socket

在 Netty 中若要学习 TCP 的拆包与粘包，则首先要清楚基于 TCP 协议的 Socket 编程。

1.3.1 创建工程

复制 02-tomcat 工程，在此基础上进行修改。

本例要实现的功能是：客户端连接上服务端后，服务端会向客户端发送一个数据。客户端每收到服务端的一个数据后，便会再向服务端发送一个数据。而服务端每收到客户端的一个数据后，便会再向客户端发送一个数据。如此反复，无穷匮也。

1.3.2 定义服务端

（1） 定义服务端启动类

```
public class SomeServer {
    public static void main(String[] args) throws InterruptedException {
        EventLoopGroup parentGroup = new NioEventLoopGroup();
        EventLoopGroup childGroup = new NioEventLoopGroup();
        try {
            ServerBootstrap bootstrap = new ServerBootstrap();
            bootstrap.group(parentGroup, childGroup)
                    .channel(NioServerSocketChannel.class)
                    // .handler(new LoggingHandler(LogLevel.INFO))
                    .childHandler(new ChannelInitializer<SocketChannel>() {
                        @Override
                        protected void initChannel(SocketChannel ch) throws Exception {
                            ChannelPipeline pipeline = ch.pipeline();
                            pipeline.addLast(new StringDecoder(CharsetUtil.UTF_8));
                            pipeline.addLast(new StringEncoder(CharsetUtil.UTF_8));
                            pipeline.addLast(new SomeSocketServerHandler());
                        }
                    });
```

```
            ChannelFuture future = bootstrap.bind(8888).sync();
            System.out.println("服务器已启动。。。");
            future.channel().closeFuture().sync();
        } finally {
            parentGroup.shutdownGracefully();
            childGroup.shutdownGracefully();
        }
    }
}
```

（2） 定义服务端处理器

```
public class SomeSocketServerHandler
                    extends ChannelInboundHandlerAdapter {
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg)
            throws Exception {
        System.out.println(ctx.channel().remoteAddress() + ", " + msg);
        ctx.channel().writeAndFlush("from server:" + UUID.randomUUID());
        TimeUnit.MILLISECONDS.sleep(500);
    }
    @Override
    public void exceptionCaught(ChannelHandlerContext ctx,
                                Throwable cause) throws Exception {
        cause.printStackTrace();
        ctx.close();
    }
}
```

1.3.3 定义客户端

（1） 定义客户端启动类

```
public class SomeClient {
    public static void main(String[] args) throws InterruptedException {
        NioEventLoopGroup eventLoopGroup = new NioEventLoopGroup();
        try {
            Bootstrap bootstrap = new Bootstrap();
            bootstrap.group(eventLoopGroup)
                    .channel(NioSocketChannel.class)
                    .handler(new ChannelInitializer<SocketChannel>() {
                        @Override
                        protected void initChannel(SocketChannel ch) throws Exception {
                            ChannelPipeline pipeline = ch.pipeline();
                            pipeline.addLast(new StringDecoder(CharsetUtil.UTF_8));
                            pipeline.addLast(new StringEncoder(CharsetUtil.UTF_8));
                            pipeline.addLast(new SomeSocketClientHandler());
                        }
                    });
```

```
            ChannelFuture future = bootstrap.connect("localhost", 8888).sync();
            future.channel().closeFuture().sync();
        } finally {
            if(eventLoopGroup != null) {
                eventLoopGroup.shutdownGracefully();
            }
        }
    }
}
```

（2） 定义客户端处理器

```
public class SomeSocketClientHandler
                extends SimpleChannelInboundHandler<String> {
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, String msg)
                        throws Exception {
        System.out.println(ctx.channel().remoteAddress() + ", " + msg);
        ctx.channel().writeAndFlush("from client: " + LocalDateTime.now());
        TimeUnit.MILLISECONDS.sleep(500);
    }

    @Override
    public void channelActive(ChannelHandlerContext ctx)
            throws Exception {
        ctx.channel().writeAndFlush("from client: begin talking");
    }
```

```
    @Override
    public void exceptionCaught(ChannelHandlerContext ctx,
                                Throwable cause) throws Exception {
        cause.printStackTrace();
        ctx.close();
    }
}
```

第2章 TCP 的拆包与粘包

第3章 Netty 高级应用

3.1 热身运动-手写 Tomcat

3.2 网络聊天 06-webchat

3.3 心跳机制

3.4 WebSocket 长连接

3.5 手写 RPC 框架

3.6 手写 Dubbo 框架

第4章 Netty 源码解析

4.1Netty 服务端启动

4.2NioEventLoop

4.3Pipeline

4.4Channel 的 inBound 与 outBound 处理器

4.5异常的传递与处理