链滴

Netty 01 - 基础

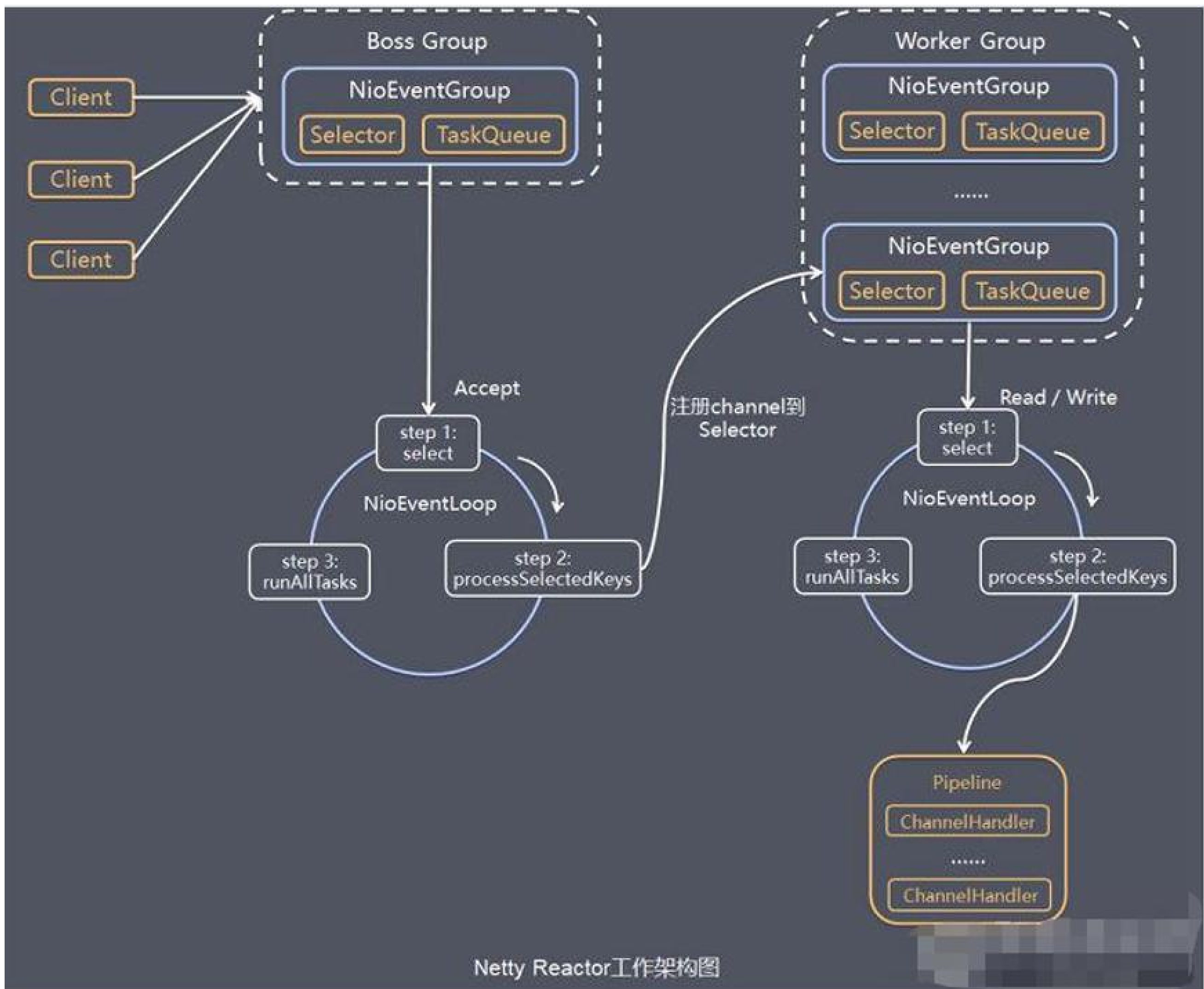
作者：ximenxiaolanggou

原文链接：https://ld246.com/article/1594890124788

来源网站：链滴

许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

Netty模型



工作原理

1. Netty抽象出两组线程池BossGroup 专门负责接收客户端的连接, WorkerGroup 专门负责网络的写

2. BossGroup 和 WorkerGroup 类型都是 NioEventLoopGroup

3. NioEventLoopGroup 相当于一个事件循环组, 这个组中含有多个事件循环，每一个事件循环是 NioventLoop

4. NioEventLoop 表示一个不断循环的执行处理任务的线程，每个NioEventLoop 都有一个selector, 用于监听绑定在其上的socket的网络通讯

5. NioEventLoopGroup 可以有多个线程, 即可以含有多个NioEventLoop

6. 每个Boss NioEventLoop 循环执行的步骤有3步

   1. 轮询accept事件

   2. 处理accept事件，与client建立连接，生成NioScocketChannel，并将其注册到某个worker NIOventLoop 上的 selector

   3. 处理任务队列的任务， 即 runAllTasks

7. 每个 WorkerNIOEventLoop 循环执行的步骤有3步

    1. 轮询read,write 事件
    2. 处理i/o事件， 即read, write 事件，在对应NioScocketChannel 处理
    3. 处理任务队列的任务， 即 runAllTasks

8. 每个WorkerNIOEventLoop 处理业务时，会使用pipeline(管道), pipeline 中包含了 channel, 即过pipeline可以获取到对应通道, 管道中维护了很多的处理器

代码

服务端

Server

```
public class NettyServer {
    public static void main(String[] args) throws Exception {


        //创建BossGroup 和 WorkerGroup
        //说明
        //1. 创建两个线程组 bossGroup 和 workerGroup
        //2. bossGroup 只是处理连接请求 , 真正的和客户端业务处理，会交给 workerGroup完成
        //3. 两个都是无限循环
        //4. bossGroup 和 workerGroup 含有的子线程(NioEventLoop)的个数
        //   默认实际 cpu核数 * 2
        EventLoopGroup bossGroup = new NioEventLoopGroup(1);
        EventLoopGroup workerGroup = new NioEventLoopGroup(); //8



        try {
            //创建服务器端的启动对象，配置参数
            ServerBootstrap bootstrap = new ServerBootstrap();

            //使用链式编程来进行设置
            bootstrap.group(bossGroup, workerGroup) //设置两个线程组
                    .channel(NioServerSocketChannel.class) //使用NioSocketChannel 作为服务器的通
实现
                    .option(ChannelOption.SO_BACKLOG, 128) // 设置线程队列得到连接个数
                    .childOption(ChannelOption.SO_KEEPALIVE, true) //设置保持活动连接状态
//                    .handler(null) // 该 handler对应 bossGroup , childHandler 对应 workerGroup
                    .childHandler(new ChannelInitializer<SocketChannel>() {//创建一个通道初始化对象
匿名对象)
                        //给pipeline 设置处理器
                        @Override
                        protected void initChannel(SocketChannel ch) throws Exception {
                            System.out.println("客户socketchannel hashcode=" + ch.hashCode()); //可
使用一个集合管理 SocketChannel， 再推送消息时，可以将业务加入到各个channel 对应的 NIOEve
tLoop 的 taskQueue 或者 scheduleTaskQueue
                            ch.pipeline().addLast(new NettyServerHandler());
```

```
                }
            }); // 给我们的workerGroup 的 EventLoop 对应的管道设置处理器

        System.out.println(".....服务器 is ready...");

        //绑定一个端口并且同步, 生成了一个 ChannelFuture 对象
        //启动服务器(并绑定端口)
        ChannelFuture cf = bootstrap.bind(6668).sync();

        //给cf 注册监听器，监控我们关心的事件

        cf.addListener(new ChannelFutureListener() {
            @Override
            public void operationComplete(ChannelFuture future) throws Exception {
                if (cf.isSuccess()) {
                    System.out.println("监听端口 6668 成功");
                } else {
                    System.out.println("监听端口 6668 失败");
                }
            }
        });


        //对关闭通道进行监听
        cf.channel().closeFuture().sync();
    }finally {
        bossGroup.shutdownGracefully();
        workerGroup.shutdownGracefully();
    }

}

}
```

ServerHandler

```
/*
说明
1. 我们自定义一个Handler 需要继续netty 规定好的某个HandlerAdapter(规范)
2. 这时我们自定义一个Handler , 才能称为一个handler
 */
public class NettyServerHandler extends ChannelInboundHandlerAdapter {

    //读取数据实际(这里我们可以读取客户端发送的消息)
    /*
    1. ChannelHandlerContext ctx:上下文对象, 含有 管道pipeline , 通道channel, 地址
    2. Object msg: 就是客户端发送的数据 默认Object
     */
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {

/*

        //比如这里我们有一个非常耗时长的业务-> 异步执行 -> 提交该channel 对应的
```

```
//NIOEventLoop 的 taskQueue中,

//解决方案1 用户程序自定义的普通任务

ctx.channel().eventLoop().execute(new Runnable() {
    @Override
    public void run() {

        try {
            Thread.sleep(5 * 1000);
            ctx.writeAndFlush(Unpooled.copiedBuffer("hello, 客户端~(>^ω^<)喵2", CharsetUt
il.UTF_8));
            System.out.println("channel code=" + ctx.channel().hashCode());
        } catch (Exception ex) {
            System.out.println("发生异常" + ex.getMessage());
        }
    }
});

ctx.channel().eventLoop().execute(new Runnable() {
    @Override
    public void run() {

        try {
            Thread.sleep(5 * 1000);
            ctx.writeAndFlush(Unpooled.copiedBuffer("hello, 客户端~(>^ω^<)喵3", CharsetUt
il.UTF_8));
            System.out.println("channel code=" + ctx.channel().hashCode());
        } catch (Exception ex) {
            System.out.println("发生异常" + ex.getMessage());
        }
    }
});

//解决方案2 : 用户自定义定时任务 -》 该任务是提交到 scheduleTaskQueue中

ctx.channel().eventLoop().schedule(new Runnable() {
    @Override
    public void run() {

        try {
            Thread.sleep(5 * 1000);
            ctx.writeAndFlush(Unpooled.copiedBuffer("hello, 客户端~(>^ω^<)喵4", CharsetUt
il.UTF_8));
            System.out.println("channel code=" + ctx.channel().hashCode());
        } catch (Exception ex) {
            System.out.println("发生异常" + ex.getMessage());
        }
    }
}, 5, TimeUnit.SECONDS);



System.out.println("go on ...");*/
```

```
        System.out.println("服务器读取线程 " + Thread.currentThread().getName() + " channle ="
+ ctx.channel());
        System.out.println("server ctx =" + ctx);
        System.out.println("看看channel 和 pipeline的关系");
        Channel channel = ctx.channel();
        ChannelPipeline pipeline = ctx.pipeline(); //本质是一个双向链接, 出站入站


        //将 msg 转成一个 ByteBuf
        //ByteBuf 是 Netty 提供的，不是 NIO 的 ByteBuffer.
        ByteBuf buf = (ByteBuf) msg;
        System.out.println("客户端发送消息是:" + buf.toString(CharsetUtil.UTF_8));
        System.out.println("客户端地址:" + channel.remoteAddress());
    }

    //数据读取完毕
    @Override
    public void channelReadComplete(ChannelHandlerContext ctx) throws Exception {

        //writeAndFlush 是 write + flush
        //将数据写入到缓存，并刷新
        //一般讲，我们对这个发送的数据进行编码
        ctx.writeAndFlush(Unpooled.copiedBuffer("hello, 客户端~(>^ω^<)喵1", CharsetUtil.UTF_
));
    }

    //处理异常, 一般是需要关闭通道

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

客户端

Client

```
public class NettyClient {
    public static void main(String[] args) throws Exception {

        //客户端需要一个事件循环组
        EventLoopGroup group = new NioEventLoopGroup();


        try {
            //创建客户端启动对象
            //注意客户端使用的不是 ServerBootstrap 而是 Bootstrap
            Bootstrap bootstrap = new Bootstrap();
```

```
            //设置相关参数
            bootstrap.group(group) //设置线程组
                    .channel(NioSocketChannel.class) // 设置客户端通道的实现类(反射)
                    .handler(new ChannelInitializer<SocketChannel>() {
                        @Override
                        protected void initChannel(SocketChannel ch) throws Exception {
                            ch.pipeline().addLast(new NettyClientHandler()); //加入自己的处理器
                        }
                    });

            System.out.println("客户端 ok..");

            //启动客户端去连接服务器端
            //关于 ChannelFuture 要分析，涉及到netty的异步模型
            ChannelFuture channelFuture = bootstrap.connect("127.0.0.1", 6668).sync();
            //给关闭通道进行监听
            channelFuture.channel().closeFuture().sync();
        }finally {

            group.shutdownGracefully();

        }
    }
}
```

ClientHandler

```
public class NettyClientHandler extends ChannelInboundHandlerAdapter {

    //当通道就绪就会触发该方法
    @Override
    public void channelActive(ChannelHandlerContext ctx) throws Exception {
        System.out.println("client " + ctx);
        ctx.writeAndFlush(Unpooled.copiedBuffer("hello, server: (>^ω^<)喵", CharsetUtil.UTF_8))

    }

    //当通道有读取事件时，会触发
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {

        ByteBuf buf = (ByteBuf) msg;
        System.out.println("服务器回复的消息:" + buf.toString(CharsetUtil.UTF_8));
        System.out.println("服务器的地址： "+ ctx.channel().remoteAddress());
    }

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

工作原理再说明

1. Netty 抽象出两组线程池，BossGroup 专门负责接收客户端连接，WorkerGroup 专门负责网络写操作。

2. NioEventLoop 表示一个不断循环执行处理任务的线程，每个NioEventLoop 都有一个 selector，于监听绑定在其上的socket网络通道。

3. NioEventLoop 内部采用串行化设计，从消息的读取->解码->处理->编码->发送，始终由IO 线程NoEventLoop 负责

- NioEventLoopGroup 下包含多个NioEventLoop
- 每个 NioEventLoop 中包含有一个Selector，一个 taskQueue
- 每个 NioEventLoop 的 Selector上可以注册监听多个NioChannel
- 每个 NioChannel 只会绑定在唯一的NioEventLoop 上
- 每个 NioChannel 都绑定有一个自己的ChannelPipeline

异步模型

基本介绍

1. 异步的概念和同步相对。当一个异步过程调用发出后，调用者不能立刻得到结果。实际处理这个调的组件在完成后，通过状态、通知和回调来通知调用者。

2. Netty 中的 I/O 操作是异步的，包括Bind、Write、Connect等操作会简单的返回一个ChannelFutue。

3. 调用者并不能立刻获得结果，而是通过Future-Listener机制，用户可以方便的主动获取或者通过知机制获得IO 操作结果

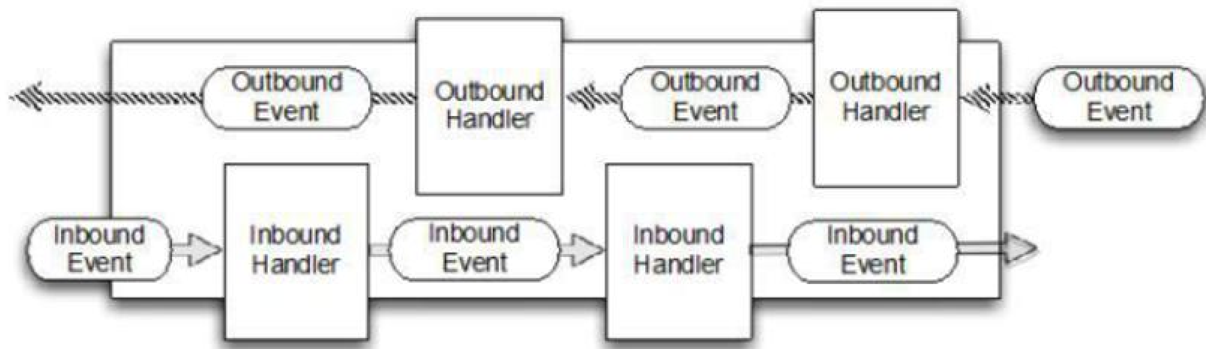
4. Netty 的异步模型是建立在future和 callback的之上的。callback就是回调。重点说Future，它的心思想是：假设一个方法fun，计算过程可能非常耗时，等待fun返回显然不合适。那么可以在调用fun的时候，立马返回一个Future，后续可以通过Future去监控方法 fun 的处理过程(即： Future-Listenr机制)

Future说明

1. 表示异步的执行结果, 可以通过它提供的方法来检测执行是否完成，比如检索计算等等.

2. ChannelFuture 是一个接口： publicinterface ChannelFuture extendsFuture <Void>我们可添加监听器，当监听的事件发生时，就会通知到监听器.

工作原理示意图



1. 在使用 Netty 进行编程时，拦截操作和转换出入站数据只需要您提供callback或利用future即可。使得链式操作简单、高效, 并有利于编写可重用的、通用的代码。

2. Netty 框架的目标就是让你的业务逻辑从网络基础应用编码中分离出来、解脱出来

Future_Listener机制

1. 当 Future对象刚刚创建时，处于非完成状态，调用者可以通过返回的ChannelFuture 来获取操作行的状态，注册监听函数来执行完成后的操作。

2. 常见有如下操作

- 通过 isDone 方法来判断当前操作是否完成;
- 通过 isSuccess 方法来判断已完成的当前操作是否成功;
- 通过 getCause 方法来获取已完成的当前操作失败的原因;
- 通过 isCancelled 方法来判断已完成的当前操作是否被取消
- 通过 addListener 方法来注册监听器，当操作已完成(isDone 方法返回完成)，将会通知指定的听器；如果Future对象已完成，则通知指定的监听器

举例

```
serverBootstrap.bind(port).addListener(future -> {
      if(future.isSuccess()) {
          System.out.println(newDate() + ": 端口["+ port + "]绑定成功!");
      } else{
          System.err.println("端口["+ port + "]绑定失败!");
      }
  });
```