crazyhzm 265

Dubbo源码解析（十六）远程通信——Netty3

java dubbo netty 阅读约 44 分钟

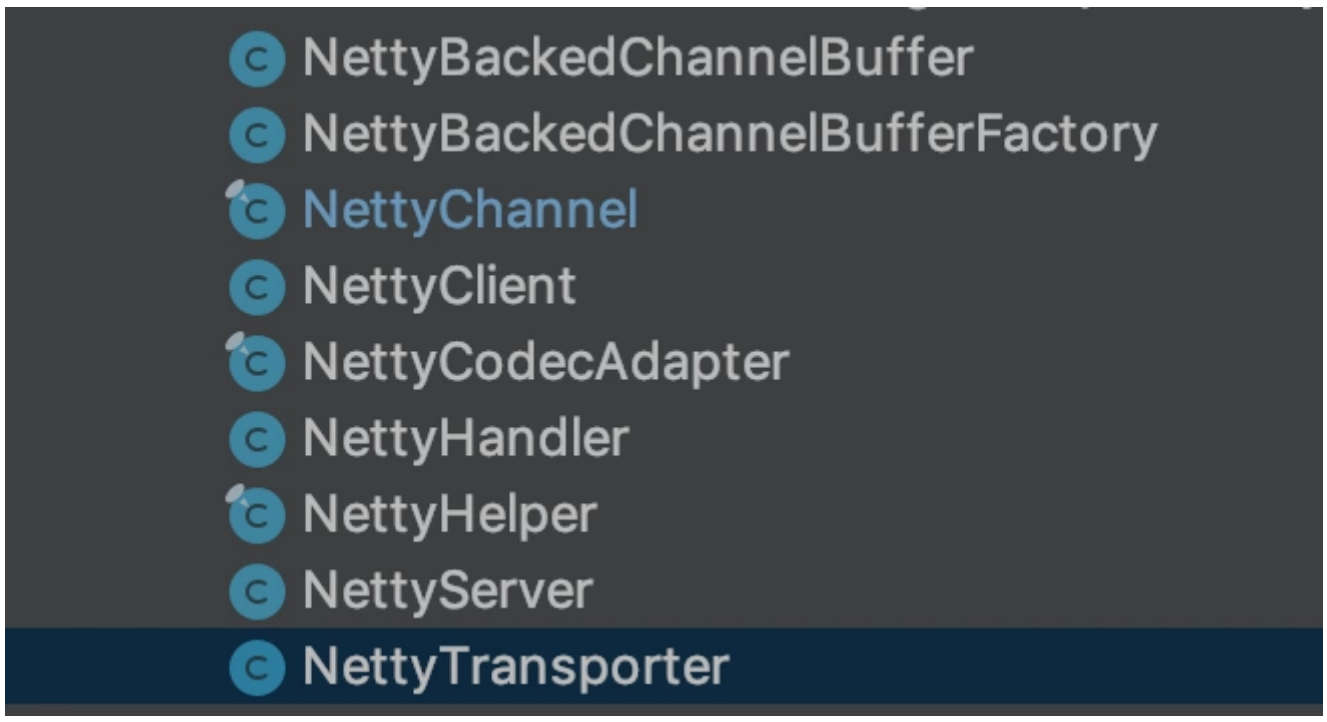
远程通讯——Netty3

> 目标：介绍基于netty3的来实现的远程通信、介绍dubbo-remoting-netty内的源码解析。

前言

现在dubbo默认的网络传输Transport接口默认实现的还是基于netty3实现的网络传输，不过马上后面默认实现就要改为netty4了。由于netty4对netty3对兼容性不是很好，所以保留了两个版本的实现。

下面是包结构：



源码分析

(一) NettyChannel

该类继承了AbstractChannel类，是基于netty3实现的通道。

1.属性

```
/**
 * 通道集合
 */
private static final ConcurrentMap<org.jboss.netty.channel.Channel, NettyChannel> channelMap = new
ConcurrentHashMap<org.jboss.netty.channel.Channel, NettyChannel>();

/**
 * 通道
 */
private final org.jboss.netty.channel.Channel channel;

/**
 * 属性集合
 */
private final Map<String, Object> attributes = new ConcurrentHashMap<String, Object>();
```

2.getOrAddChannel

```
static NettyChannel getOrAddChannel(org.jboss.netty.channel.Channel ch, URL url, ChannelHandler handler) {
    if (ch == null) {
        return null;
    }
    // 首先从集合中取通道
    NettyChannel ret = channelMap.get(ch);
    // 如果为空，则新建
    if (ret == null) {
        NettyChannel nc = new NettyChannel(ch, url, handler);
        // 如果通道连接着
        if (ch.isConnected()) {
            // 加入集合
            ret = channelMap.putIfAbsent(ch, nc);
        }
        if (ret == null) {
            ret = nc;
        }
    }
    return ret;
}
```

该方法是获得通道，当通道在集合中没有的时候，新建一个通道。

3.removeChannelIfDisconnected

```
static void removeChannelIfDisconnected(org.jboss.netty.channel.Channel ch) {
    if (ch != null && !ch.isConnected()) {
        channelMap.remove(ch);
    }
}
```

该方法是当通道没有连接的时候，从集合中移除它。

4.send

```
@Override
public void send(Object message, boolean sent) throws RemotingException {
    super.send(message, sent);

    boolean success = true;
    int timeout = 0;
    try {
        // 写入数据，发送消息
        ChannelFuture future = channel.write(message);
        // 如果已经发送过
        if (sent) {
            // 获得超时时间
            timeout = getUrl().getPositiveParameter(Constants.TIMEOUT_KEY, Constants.DEFAULT_TIMEOUT);
            // 等待timeout的连接时间后查看是否发送成功
            success = future.await(timeout);
        }
        // 看是否有异常
        Throwable cause = future.getCause();
        if (cause != null) {
            throw cause;
        }
    } catch (Throwable e) {
        throw new RemotingException(this, "Failed to send message " + message + " to " + getRemoteAddress() + ",
cause: " + e.getMessage(), e);
    }
```

该方法是发送消息，其中用到了channe.write方法传输消息，并且通过返回的future来判断是否发送成功。

5.close

```
@Override
public void close() {
    try {
        super.close();
    } catch (Exception e) {
        logger.warn(e.getMessage(), e);
    }
    try {
        // 如果通道断开，则移除该通道
        removeChannelIfDisconnected(channel);
    } catch (Exception e) {
        logger.warn(e.getMessage(), e);
    }
    try {
        // 清空属性
        attributes.clear();
    } catch (Exception e) {
        logger.warn(e.getMessage(), e);
    }
    try {
        if (logger.isInfoEnabled()) {
            logger.info("Close netty channel " + channel);
        }
        // 关闭通道
        channel.close();
    } catch (Exception e) {
```

该方法是关闭通道，做了三个操作，分别是从集合中移除、清除属性、关闭通道。

其他实现方法比较简单，我就讲解了。

（二）NettyHandler

该类继承了SimpleChannelHandler类，是基于netty3的通道处理器，而该类被加上了@Sharable注解，也就是说该处理器可以从属于多个ChannelPipeline

1.属性

```
/**
 * 通道集合，key是主机地址 ip:port
 */
private final Map<String, Channel> channels = new ConcurrentHashMap<String, Channel>(); // <ip:port, channel>

/**
 * url对象
 */
private final URL url;

/**
 * 通道
 */
private final ChannelHandler handler;
```

该类的属性比较简单，并且该类中实现的方法都是调用了属性handler的方法，我举一个例子来讲，其他的可以自己查看源码，比较简单。

```
@Override
public void channelConnected(ChannelHandlerContext ctx, ChannelStateEvent e) throws Exception {
    // 获得通道实例
    NettyChannel channel = NettyChannel.getOrAddChannel(ctx.getChannel(), url, handler);
    try {
        if (channel != null) {
            // 保存该通道，加入到集合中
            channels.put(NetUtils.toAddressString((InetSocketAddress) ctx.getChannel().getRemoteAddress()), channel);
        }
        // 连接
        handler.connected(channel);
    } finally {
        NettyChannel.removeChannelIfDisconnected(ctx.getChannel());
    }
}
```

该方法是通道连接的方法，其中先获取了通道实例，然后吧该实例加入到集合中，最好带哦用handler.connected来进行连接。

（三）NettyClient

该类继承了AbstractClient，是基于netty3实现的客户端类。

1.属性

```
private static final Logger logger = LoggerFactory.getLogger(NettyClient.class);

// ChannelFactory's closure has a DirectMemory leak, using static to avoid
// https://issues.jboss.org/browse/NETTY-424
/**
 * 通道工厂，用static来避免直接缓存区的一个OOM问题
 */
private static final ChannelFactory channelFactory = new
NioClientSocketChannelFactory(Executors.newCachedThreadPool(new NamedThreadFactory("NettyClientBoss", true)),
        Executors.newCachedThreadPool(new NamedThreadFactory("NettyClientWorker", true)),
        Constants.DEFAULT_IO_THREADS);
/**
 * 客户端引导对象
 */
private ClientBootstrap bootstrap;

/**
 * 通道
 */
private volatile Channel channel; // volatile, please copy reference to use
```

上述属性中ChannelFactory用了static修饰，为了避免netty3中会有直接缓冲内存泄漏的现象，具体的讨论可以访问注释中的讨论。

2.doOpen

```
@Override
protected void doOpen() throws Throwable {
    // 设置日志工厂
    NettyHelper.setNettyLoggerFactory();
    // 实例化客户端引导类
    bootstrap = new ClientBootstrap(channelFactory);
    // config
    // @see org.jboss.netty.channel.socket.SocketChannelConfig
    // 配置选择项
    bootstrap.setOption("keepAlive", true);
    bootstrap.setOption("tcpNoDelay", true);
    bootstrap.setOption("connectTimeoutMillis", getConnectTimeout());
    // 创建通道处理器
    final NettyHandler nettyHandler = new NettyHandler(getUrl(), this);
    // 设置责任链路
    bootstrap.setPipelineFactory(new ChannelPipelineFactory() {
        /**
         * 获得通道
         * @return
         */
        @Override
        public ChannelPipeline getPipeline() {
            // 新建编解码
            NettyCodecAdapter adapter = new NettyCodecAdapter(getCodec(), getUrl(), NettyClient.this);
            // 获得管道
            ChannelPipeline pipeline = Channels.pipeline();
```

该方法是创建客户端，并且打开，其中的逻辑就是用netty3的客户端引导类来创建一个客户端，如果对netty不熟悉的朋友可以先补补netty知识。

3.doConnect

```
@Override
protected void doConnect() throws Throwable {
    long start = System.currentTimeMillis();
    // 用引导类连接
    ChannelFuture future = bootstrap.connect(getConnectAddress());
    try {
        // 在超时时间内是否连接完成
        boolean ret = future.awaitUninterruptibly(getConnectTimeout(), TimeUnit.MILLISECONDS);

        if (ret && future.isSuccess()) {
            // 获得通道
            Channel newChannel = future.getChannel();
            // 异步修改此通道
            newChannel.setInterestOps(Channel.OP_READ_WRITE);
            try {
                // Close old channel 关闭旧的通道
                Channel oldChannel = NettyClient.this.channel; // copy reference
                if (oldChannel != null) {
                    try {
                        if (logger.isInfoEnabled()) {
                            logger.info("Close old netty channel " + oldChannel + " on create new netty channel "
+ newChannel);
                        }
                        // 关闭
                        oldChannel.close();
                    } finally {
```

该方法是客户端连接服务器的方法。其中调用了bootstrap.connect。后面的逻辑是用来检测是否连接，最后如果未连接，则会取消该连接任务。

4.doClose

```
@Override
protected void doClose() throws Throwable {
    /*try {
        bootstrap.releaseExternalResources();
    } catch (Throwable t) {
        logger.warn(t.getMessage());
    }*/
}
```

在这里不能关闭是因为channelFactory 是静态属性，被多个 NettyClient 共用。所以不能释放资源。

（四）NettyServer

该类继承了AbstractServer，实现了Server，是基于netty3实现的服务器类。

1.属性

```
/**
 * 连接该服务器的通道集合
 */
private Map<String, Channel> channels; // <ip:port, channel>

/**
 * 服务器引导类对象
 */
private ServerBootstrap bootstrap;

/**
 * 通道
 */
private org.jboss.netty.channel.Channel channel;
```

2.doOpen

```
@Override
protected void doOpen() throws Throwable {
    // 设置日志工厂
    NettyHelper.setNettyLoggerFactory();
    // 创建线程池
    ExecutorService boss = Executors.newCachedThreadPool(new NamedThreadFactory("NettyServerBoss", true));
    ExecutorService worker = Executors.newCachedThreadPool(new NamedThreadFactory("NettyServerWorker", true));
    // 新建通道工厂
    ChannelFactory channelFactory = new NioServerSocketChannelFactory(boss, worker,
getUrl().getPositiveParameter(Constants.IO_THREADS_KEY, Constants.DEFAULT_IO_THREADS));
    // 新建服务引导类对象
    bootstrap = new ServerBootstrap(channelFactory);

    // 新建通道处理器
    final NettyHandler nettyHandler = new NettyHandler(getUrl(), this);
    // 获得通道集合
    channels = nettyHandler.getChannels();
    // https://issues.jboss.org/browse/NETTY-365
    // https://issues.jboss.org/browse/NETTY-379
    // final Timer timer = new HashedWheelTimer(new NamedThreadFactory("NettyIdleTimer", true));
    // 禁用nagle算法，将数据立即发送出去。纳格算法是以减少封包传送量来增进TCP/IP网络的效能
    bootstrap.setOption("child.tcpNoDelay", true);
    // 设置管道工厂
    bootstrap.setPipelineFactory(new ChannelPipelineFactory() {
        /**
         * 获得通道
```

该方法是创建服务器，并且打开服务器。同样创建服务器的方式跟正常的用netty创建服务器方式一样，只是新加了编码器和解码器。还有一个注意点就是这里ServerBootstrap 的可选项。

3.doClose

```
@Override
protected void doClose() throws Throwable {
    try {
        if (channel != null) {
            // unbind.关闭通道
            channel.close();
        }
    } catch (Throwable e) {
        logger.warn(e.getMessage(), e);
    }
    try {
        // 获得所有连接该服务器的通道集合
        Collection<com.alibaba.dubbo.remoting.Channel> channels = getChannels();
        if (channels != null && !channels.isEmpty()) {
            // 遍历通道集合
            for (com.alibaba.dubbo.remoting.Channel channel : channels) {
                try {
                    // 关闭通道连接
                    channel.close();
                } catch (Throwable e) {
                    logger.warn(e.getMessage(), e);
                }
            }
        }
    } catch (Throwable e) {
        logger.warn(e.getMessage(), e);
```

该方法是关闭服务器，一系列的操作很清晰，我就不多说了。

4.getChannels

```
@Override
public Collection<Channel> getChannels() {
    Collection<Channel> chs = new HashSet<Channel>();
    for (Channel channel : this.channels.values()) {
        // 如果通道连接，则加入集合，返回
        if (channel.isConnected()) {
            chs.add(channel);
        } else {
            channels.remove(NetUtils.toAddressString(channel.getRemoteAddress()));
        }
    }
    return chs;
}
```

该方法是返回连接该服务器的通道集合，并且用了HashSet保存，不会重复。

(五) NettyTransporter

```
public class NettyTransporter implements Transporter {

    public static final String NAME = "netty";

    @Override
    public Server bind(URL url, ChannelHandler listener) throws RemotingException {
        // 创建一个NettyServer
        return new NettyServer(url, listener);
    }

    @Override
    public Client connect(URL url, ChannelHandler listener) throws RemotingException {
        // 创建一个NettyClient
        return new NettyClient(url, listener);
    }

}
```

该类就是基于netty3的Transporter实现类，同样两个方法也是分别创建了NettyServer和NettyClient。

（六）NettyHelper

该类是设置日志的工具类，其中基于netty3的InternalLoggerFactory实现类一个DubboLoggerFactory。这个我就不讲解了，比较好理解，不理解也无伤大雅。

（七）NettyCodecAdapter

该类是基于netty3实现的编解码类。

1.属性

```
/**
 * 编码者
 */
private final ChannelHandler encoder = new InternalEncoder();

/**
 * 解码者
 */
private final ChannelHandler decoder = new InternalDecoder();

/**
 * 编解码器
 */
private final Codec2 codec;

/**
 * url对象
 */
private final URL url;

/**
 * 缓冲区大小
 */
private final int bufferSize;

/**
```

InternalEncoder和InternalDecoder属性是该类的内部类，分别掌管着编码和解码

2.构造方法

```
public NettyCodecAdapter(Codec2 codec, URL url, com.alibaba.dubbo.remoting.ChannelHandler handler) {
    this.codec = codec;
    this.url = url;
    this.handler = handler;
    int b = url.getPositiveParameter(Constants.BUFFER_KEY, Constants.DEFAULT_BUFFER_SIZE);
    // 如果缓存区大小在16字节以内，则设置配置大小，如果不是，则设置8字节的缓冲区大小
    this.bufferSize = b >= Constants.MIN_BUFFER_SIZE && b <= Constants.MAX_BUFFER_SIZE ? b :
Constants.DEFAULT_BUFFER_SIZE;
}
```

你会发现对于缓存区大小的规则都是一样的。

3.InternalEncoder

```
@Sharable
private class InternalEncoder extends OneToOneEncoder {

    @Override
    protected Object encode(ChannelHandlerContext ctx, Channel ch, Object msg) throws Exception {
        // 动态分配一个1k的缓冲区
        com.alibaba.dubbo.remoting.buffer.ChannelBuffer buffer =
                com.alibaba.dubbo.remoting.buffer.ChannelBuffers.dynamicBuffer(1024);
        // 获得通道对象
        NettyChannel channel = NettyChannel.getOrAddChannel(ch, url, handler);
        try {
            // 编码
            codec.encode(channel, buffer, msg);
        } finally {
            NettyChannel.removeChannelIfDisconnected(ch);
        }
        // 基于buteBuffer创建一个缓冲区，并且写入数据
        return ChannelBuffers.wrappedBuffer(buffer.toByteBuffer());
    }
}
```

该内部类实现类编码的逻辑，主要调用了codec.encode。

4.InternalDecoder

```
private class InternalDecoder extends SimpleChannelUpstreamHandler {

    private com.alibaba.dubbo.remoting.buffer.ChannelBuffer buffer =
            com.alibaba.dubbo.remoting.buffer.ChannelBuffers.EMPTY_BUFFER;

    @Override
    public void messageReceived(ChannelHandlerContext ctx, MessageEvent event) throws Exception {
        Object o = event.getMessage();
        // 如果消息不是一个ChannelBuffer类型
        if (!(o instanceof ChannelBuffer)) {
            // 转发事件到与此上下文关联的处理程序最近的上游
            ctx.sendUpstream(event);
            return;
        }

        ChannelBuffer input = (ChannelBuffer) o;
        // 如果可读数据不大于0，直接返回
        int readable = input.readableBytes();
        if (readable <= 0) {
            return;
        }

        com.alibaba.dubbo.remoting.buffer.ChannelBuffer message;
        if (buffer.readable()) {
            // 判断buffer是否是动态分配的缓冲区
            if (buffer instanceof DynamicChannelBuffer) {
```

该内部类实现了解码的逻辑，其中大部分逻辑都在对数据做读写，关键的解码调用了codec.decode。

（八）NettyBackedChannelBufferFactory

该类是创建缓冲区的工厂类。它实现了ChannelBufferFactory接口，也就是实现类它的三种获得缓冲区的方法。

```java
public class NettyBackedChannelBufferFactory implements ChannelBufferFactory {

    /**
     * 单例
     */
    private static final NettyBackedChannelBufferFactory INSTANCE = new NettyBackedChannelBufferFactory();

    public static ChannelBufferFactory getInstance() {
        return INSTANCE;
    }


    @Override
    public ChannelBuffer getBuffer(int capacity) {
        return new NettyBackedChannelBuffer(ChannelBuffers.dynamicBuffer(capacity));
    }


    @Override
    public ChannelBuffer getBuffer(byte[] array, int offset, int length) {
        org.jboss.netty.buffer.ChannelBuffer buffer = ChannelBuffers.dynamicBuffer(length);
        buffer.writeBytes(array, offset, length);
        return new NettyBackedChannelBuffer(buffer);
    }


```

可以看到，都是创建了一个NettyBackedChannelBuffer，下面讲解NettyBackedChannelBuffer。

（九）NettyBackedChannelBuffer

该类是基于netty3的buffer重新实现的缓冲区，它实现了ChannelBuffer接口，并且有一个属性：

```java
private org.jboss.netty.buffer.ChannelBuffer buffer;
```

那么其中的几乎所有方法都是调用了这个buffer的方法，因为我在dubbo源码解析（十一）远程通信——Buffer中写到ChannelBuffer接口方法定义跟netty中的缓冲区定义几乎一样，连注释都几乎一样。所有知识单纯的调用了buffer的方法。具体的代码可以查看我的GitHub

后记

> 该部分相关的源码解析地址：https://github.com/CrazyHZM/i...

该文章讲解了基于netty3的来实现的远程通信、介绍dubbo-remoting-netty内的源码解析，关键需要对netty有所了解。下一篇我会讲解基于netty4实现远程通信部分。

crazyhzm

265

关注作者

0 条评论

得票 • 时间

撰写评论 ...