crazyhzm 265

Dubbo源码解析（十）远程通信——Exchange层

java dubbo 阅读约 110 分钟

远程通讯——Exchange层

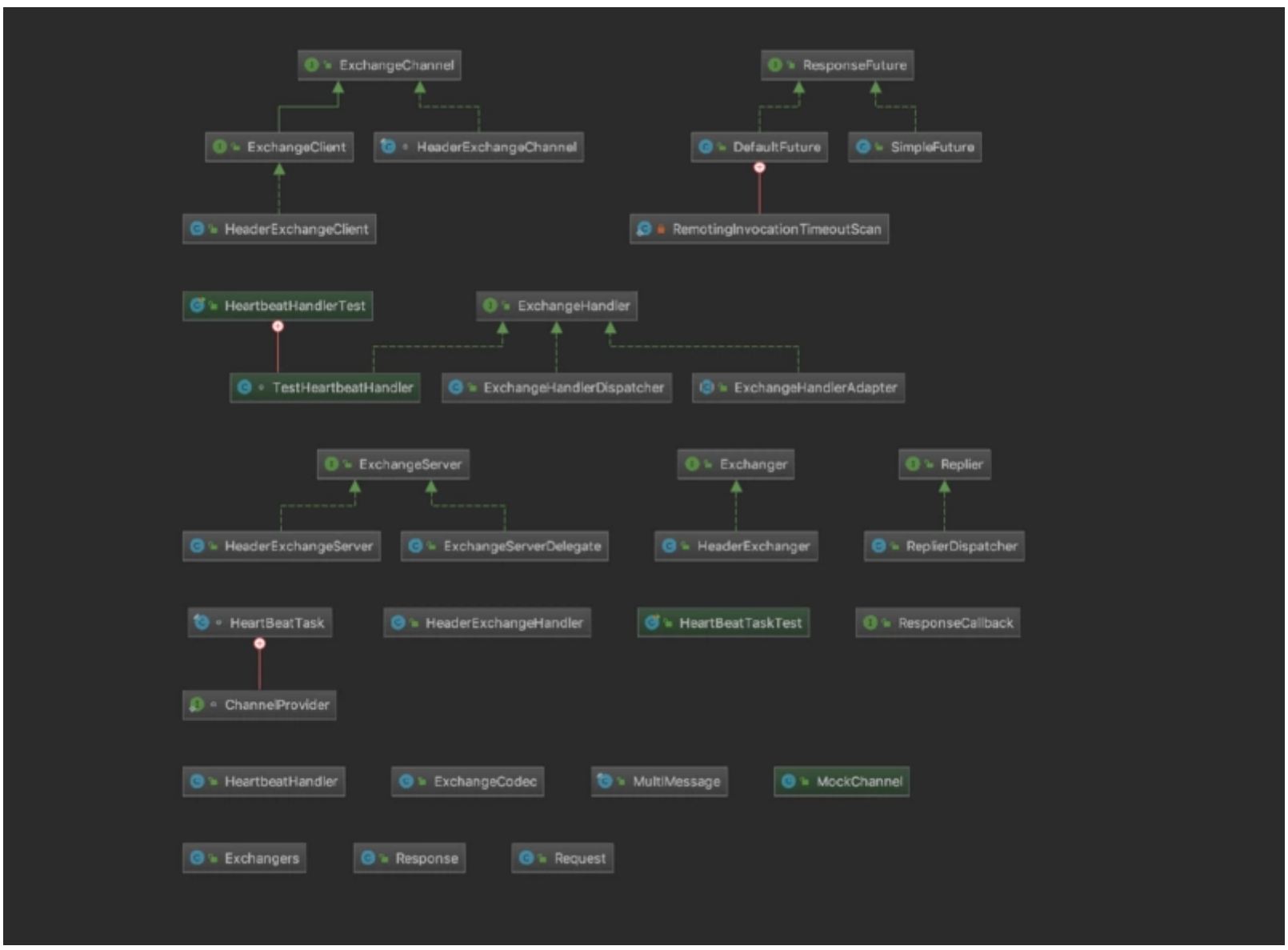
> 目标：介绍Exchange层的相关设计和逻辑、介绍dubbo-remoting-api中的exchange包内的源码解析。

前言

上一篇文章我讲的是dubbo框架设计中Transport层，这篇文章我要讲的是它的上一层Exchange层，也就是信息交换层。官方文档对这一层的解释是封装请求响应模式，同步转异步，以 Request, Response为中心，扩展接口为 Exchanger, ExchangeChannel, ExchangeClient, ExchangeServer。

这一层的设计意图是什么？它应该算是在信息传输层上又做了部分装饰，为了适应rpc调用的一些需求，比如rpc调用中一次请求只关心它所对应的响应，这个时候只是一个message消息传输过来，是无法区分这是新的请求还是上一个请求的响应，这种类似于幂等性的问题以及rpc异步处理返回结果、内置事件等特性都是在Transport层无法解决满足的，所有在Exchange层讲message分成了request和response两种类型，并且在这两个模型上增加一些系统字段来处理问题。具体我会在下面讲到。而dubbo把一条消息分为了协议头和内容两部分：协议头包括系统字段，例如编号等，内容包括具体请求的参数和响应的结果等。在exchange层中大量逻辑都是基于协议头的。

现在对这一层的设计意图大致应该有所了解了吧，现在来看看exchange的类图：



我讲解的顺序还是按照类图从上而下，分块讲解，忽略绿色的test类。

源码解析

（一）ExchangeChannel

```
public interface ExchangeChannel extends Channel {

    ResponseFuture request(Object request) throws RemotingException;

    ResponseFuture request(Object request, int timeout) throws RemotingException;

    ExchangeHandler getExchangeHandler();

    @Override
    void close(int timeout);

}
```

该接口是信息交换通道接口，有四个方法，前两个是发送请求消息，区别就是第二个发送请求有超时的参数，getExchangeHandler方法就是返回一个信息交换处理器，第四个是需要覆写父类的方法。

（二）HeaderExchangeChannel

该类实现了ExchangeChannel，是基于协议头的信息交换通道。

1.属性

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

/**
 * 通道的key值
 */
private static final String CHANNEL_KEY = HeaderExchangeChannel.class.getName() + ".CHANNEL";

/**
 * 通道
 */
private final Channel channel;

/**
 * 是否关闭
 */
private volatile boolean closed = false;
```

上述属性比较简单，还是放一下这个类的属性是因为该类中有channel属性，也就是说HeaderExchangeChannel是Channel的装饰器，每个实现方法都会调用channel的方法。

2.静态方法

```
static HeaderExchangeChannel getOrAddChannel(Channel ch) {
    if (ch == null) {
        return null;
    }
    // 获得通道中的HeaderExchangeChannel
    HeaderExchangeChannel ret = (HeaderExchangeChannel) ch.getAttribute(CHANNEL_KEY);
    if (ret == null) {
        // 创建一个HeaderExchangeChannel实例
        ret = new HeaderExchangeChannel(ch);
        // 如果通道连接
        if (ch.isConnected()) {
            // 加入属性值
            ch.setAttribute(CHANNEL_KEY, ret);
        }
    }
    return ret;
}

static void removeChannelIfDisconnected(Channel ch) {
    // 如果通道断开连接
    if (ch != null && !ch.isConnected()) {
        // 移除属性值
        ch.removeAttribute(CHANNEL_KEY);
    }
}
```

该静态方法做了HeaderExchangeChannel的创建和销毁，并且生命周期随channel销毁而销毁。

3.send

```
@Override
public void send(Object message) throws RemotingException {
    send(message, getUrl().getParameter(Constants.SENT_KEY, false));
}

@Override
public void send(Object message, boolean sent) throws RemotingException {
    // 如果通道关闭，抛出异常
    if (closed) {
        throw new RemotingException(this.getLocalAddress(), null, "Failed to send message " + message + ", cause:
The channel " + this + " is closed!");
    }
    // 判断消息的类型
    if (message instanceof Request
            || message instanceof Response
            || message instanceof String) {
        // 发送消息
        channel.send(message, sent);
    } else {
        // 新建一个request实例
        Request request = new Request();
        // 设置信息的版本
        request.setVersion(Version.getProtocolVersion());
        // 该请求不需要响应
        request.setTwoWay(false);
        // 把消息传入
```

该方法是在channel的send方法上加上了request和response模型，最后再调用channel.send，起到了装饰器的作用。

4.request

```
@Override
public ResponseFuture request(Object request) throws RemotingException {
    return request(request, channel.getUrl().getPositiveParameter(Constants.TIMEOUT_KEY,
Constants.DEFAULT_TIMEOUT));
}

@Override
public ResponseFuture request(Object request, int timeout) throws RemotingException {
    // 如果通道关闭，则抛出异常
    if (closed) {
        throw new RemotingException(this.getLocalAddress(), null, "Failed to send request " + request + ", cause:
The channel " + this + " is closed!");
    }
    // create request.创建请求
    Request req = new Request();
    // 设置版本号
    req.setVersion(Version.getProtocolVersion());
    // 设置需要响应
    req.setTwoWay(true);
    // 把请求数据传入
    req.setData(request);
    // 创建DefaultFuture对象，可以从future中主动获得请求对应的响应信息
    DefaultFuture future = new DefaultFuture(channel, req, timeout);
    try {
        // 发送请求消息
        channel.send(req);
```

该方法是请求方法，用Request模型把请求内容装饰起来，然后发送一个Request类型的消息，并且返回DefaultFuture实例，DefaultFuture我会在后面讲到。

cloes方法也重写了，我就不再多说，因为比较简单，没有重点，其他方法都是直接调用channel属性的方法。

（三）ExchangeClient

该接口继承了Client和ExchangeChannel，是信息交换客户端接口，其中没有定义多余的方法。

（四）HeaderExchangeClient

该类实现了ExchangeClient接口，是基于协议头的信息交互客户端类，同样它是Client、Channel的适配器。在该类的源码中可以看到所有的实现方法都是调用了client和channel属性的方法。该类主要的作用就是增加了心跳功能，为什么要增加心跳功能呢，对于长连接，一些拔网线等物理层的断开，会导致TCP的FIN消息来不及发送，对方收不到断开事件，那么就需要用到发送心跳包来检测连接是否断开。consumer和provider断开，处理措施不一样，会分别做出重连和关闭通道的操作。

1.属性

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

/**
 * 定时器线程池
 */
private static final ScheduledThreadPoolExecutor scheduled = new ScheduledThreadPoolExecutor(2, new
NamedThreadFactory("dubbo-remoting-client-heartbeat", true));
/**
 * 客户端
 */
private final Client client;
/**
 * 信息交换通道
 */
private final ExchangeChannel channel;
// heartbeat timer
/**
 * 心跳定时器
 */
private ScheduledFuture<?> heartbeatTimer;
// heartbeat(ms), default value is 0 , won't execute a heartbeat.
/**
 * 心跳周期，间隔多久发送心跳消息检测一次
 */
private int heartbeat;
/**
```

该类的属性除了需要适配的属性外，其他都是跟心跳相关属性。

2.构造函数

```
public HeaderExchangeClient(Client client, boolean needHeartbeat) {
    if (client == null) {
        throw new IllegalArgumentException("client == null");
    }
    this.client = client;
    // 创建信息交换通道
    this.channel = new HeaderExchangeChannel(client);
    // 获得dubbo版本
    String dubbo = client.getUrl().getParameter(Constants.DUBBO_VERSION_KEY);
    //获得心跳周期配置，如果没有配置，并且dubbo是1.0版本的，则这只为1分钟，否则设置为0
    this.heartbeat = client.getUrl().getParameter(Constants.HEARTBEAT_KEY, dubbo != null && dubbo.startsWith("1.0.") ?
Constants.DEFAULT_HEARTBEAT : 0);
    // 获得心跳超时配置，默认是心跳周期的三倍
    this.heartbeatTimeout = client.getUrl().getParameter(Constants.HEARTBEAT_TIMEOUT_KEY, heartbeat * 3);
    // 如果心跳超时时间小于心跳周期的两倍，则抛出异常
    if (heartbeatTimeout < heartbeat * 2) {
        throw new IllegalStateException("heartbeatTimeout < heartbeatInterval * 2");
    }
    if (needHeartbeat) {
        // 开启心跳
        startHeartbeatTimer();
    }
}
```

构造函数就是对一些属性初始化设置，优先从url中获取。心跳超时时间小于心跳周期的两倍就抛出异常，意思就是至少重试两次心跳检测。

3.startHeartbeatTimer

```
private void startHeartbeatTimer() {
    // 停止现有的心跳线程
    stopHeartbeatTimer();
    // 如果需要心跳
    if (heartbeat > 0) {
        // 创建心跳定时器
        heartbeatTimer = scheduled.scheduleWithFixedDelay(
                // 新建一个心跳线程
                new HeartBeatTask(new HeartBeatTask.ChannelProvider() {
                    @Override
                    public Collection<Channel> getChannels() {
                        // 返回一个只包含HeaderExchangeClient对象的不可变列表
                        return Collections.<Channel>singletonList(HeaderExchangeClient.this);
                    }
                }, heartbeat, heartbeatTimeout),
                heartbeat, heartbeat, TimeUnit.MILLISECONDS);
    }
}
```

该方法就是开启心跳。利用心跳定时器来做到定时检测心跳。因为这是信息交换客户端类，所有这里的只是返回包含HeaderExchangeClient对象的不可变列表，因为客户端跟channel是一一对应的，只有这一个该客户端本身的channel需要心跳。

4.stopHeartbeatTimer

```
private void stopHeartbeatTimer() {
    if (heartbeatTimer != null && !heartbeatTimer.isCancelled()) {
        try {
            // 取消定时器
            heartbeatTimer.cancel(true);
            // 取消大量已排队任务，用于回收空间
            scheduled.purge();
        } catch (Throwable e) {
            if (logger.isWarnEnabled()) {
                logger.warn(e.getMessage(), e);
            }
        }
    }
    heartbeatTimer = null;
}
```

该方法是停止现有心跳，也就是停止定时器，释放空间。

其他方法都是调用channel和client属性的方法。

（五）HeartBeatTask

该类实现了Runnable接口，实现的是心跳任务，里面包含了核心的心跳策略。

1.属性

```
/**
 * 通道管理
 */
private ChannelProvider channelProvider;

/**
 * 心跳间隔 单位：ms
 */
private int heartbeat;

/**
 * 心跳超时时间 单位：ms
 */
private int heartbeatTimeout;
```

后两个属性跟HeaderExchangeClient中的属性含义一样，第一个是该类自己内部的一个接口：

```
interface ChannelProvider {
    // 获得所有的通道集合，需要心跳的通道数组
    Collection<Channel> getChannels();
}
```

该接口就定义了一个方法，获得需要心跳的通道集合。可想而知，会对集合内的通道都做心跳检测。

2.run

```
@Override
public void run() {
    try {
        long now = System.currentTimeMillis();
        // 遍历所有通道
        for (Channel channel : channelProvider.getChannels()) {
            // 如果通道关闭了，则跳过
            if (channel.isClosed()) {
                continue;
            }
            try {
                // 最后一次接收到消息的时间戳
                Long lastRead = (Long) channel.getAttribute(
                        HeaderExchangeHandler.KEY_READ_TIMESTAMP);
                // 最后一次发送消息的时间戳
                Long lastWrite = (Long) channel.getAttribute(
                        HeaderExchangeHandler.KEY_WRITE_TIMESTAMP);
                // 如果最后一次接收或者发送消息到时间到现在的时间间隔超过了心跳间隔时间
                if ((lastRead != null && now - lastRead > heartbeat)
                        || (lastWrite != null && now - lastWrite > heartbeat)) {
                    // 创建一个request
                    Request req = new Request();
                    // 设置版本号
                    req.setVersion(Version.getProtocolVersion());
                    // 设置需要得到响应
                    req.setTwoWay(true);
```

该方法中是心跳机制的核心逻辑。注意以下几个点：

1. 如果需要心跳的通道本身如果关闭了，那么跳过，不添加心跳机制。
2. 无论是接收消息还是发送消息，只要超过了设置的心跳间隔，就发送心跳消息来测试是否断开
3. 如果最后一次接收到消息到到现在已经超过了心跳超时时间，那就认定对方的确断开，分两种情况来处理对方断开的情况。分别是服务端断开，客户端重连以及客户端断开，服务端断开这个客户端的连接。，这里要好好品味一下谁是发送方，谁在等谁的响应，苦苦没有等到。

（六）ResponseFuture

```
public interface ResponseFuture {

    Object get() throws RemotingException;

    Object get(int timeoutInMillis) throws RemotingException;

    void setCallback(ResponseCallback callback);

    boolean isDone();

}
```

该接口是响应future接口，该接口的设计意图跟java.util.concurrent.Future很类似。发送出去的消息，泼出去的水，只有等到对方主动响应才能得到结果，但是请求方需要去主动回去该请求的结果，就显得有些艰难，所有产生了这样一个接口，它能够获取任务执行结果、可以核对请求消息是否被响应，还能设置回调来支持异步。

（七）DefaultFuture

该类实现了ResponseFuture接口，其中封装了处理响应的逻辑。你可以把DefaultFuture看成是一个中介，买房和卖房都通过这个中介进行沟通，中介拥有着买房者的信息request和卖房者的信息response，并且促成他们之间的买卖。

1.属性

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

/**
 * 通道集合
 */
private static final Map<Long, Channel> CHANNELS = new ConcurrentHashMap<Long, Channel>();

/**
 * Future集合，key为请求编号
 */
private static final Map<Long, DefaultFuture> FUTURES = new ConcurrentHashMap<Long, DefaultFuture>();

// invoke id.
/**
 * 请求编号
 */
private final long id;
/**
 * 通道
 */
private final Channel channel;
/**
 * 请求
 */
private final Request request;
/**
```

可以看到，该类的属性包含了request、response、channel三个实例，在该类中，把请求和响应通过唯一的id一一对应起来。做到异步处理返回结果时能给准确的返回给对应的请求。可以看到属性中有两个集合，分别是通道集合和future集合，也就是该类本身也是所有 DefaultFuture 的管理容器。

2.构造函数

```
public DefaultFuture(Channel channel, Request request, int timeout) {
    this.channel = channel;
    this.request = request;
    // 设置请求编号
    this.id = request.getId();
    this.timeout = timeout > 0 ? timeout : channel.getUrl().getPositiveParameter(Constants.TIMEOUT_KEY,
Constants.DEFAULT_TIMEOUT);
    // put into waiting map.，加入到等待集合中
    FUTURES.put(id, this);
    CHANNELS.put(id, channel);
}
```

构造函数比较简单，每一个DefaultFuture实例都跟每一个请求一一对应，被存入到集合中管理起来。

3.closeChannel

```
public static void closeChannel(Channel channel) {
    // 遍历通道集合
    for (long id : CHANNELS.keySet()) {
        if (channel.equals(CHANNELS.get(id))) {
            // 通过请求id获得future
            DefaultFuture future = getFuture(id);
            if (future != null && !future.isDone()) {
                // 创建一个关闭通道的响应
                Response disconnectResponse = new Response(future.getId());
                disconnectResponse.setStatus(Response.CHANNEL_INACTIVE);
                disconnectResponse.setErrorMessage("Channel " +
                        channel +
                        " is inactive. Directly return the unFinished request : " +
                        future.getRequest());
                // 接收该关闭通道并且请求未完成的响应
                DefaultFuture.received(channel, disconnectResponse);
            }
        }
    }
}
```

该方法是关闭不活跃的通道，并且返回请求未完成。也就是关闭指定channel的请求，返回的是请求未完成。

4.received

```
public static void received(Channel channel, Response response) {
    try {
        // future集合中移除该请求的future，（响应id和请求id一一对应的）
        DefaultFuture future = FUTURES.remove(response.getId());
        if (future != null) {
            // 接收响应结果
            future.doReceived(response);
        } else {
            logger.warn("The timeout response finally returned at "
                    + (new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS").format(new Date()))
                    + ", response " + response
                    + (channel == null ? "" : ", channel: " + channel.getLocalAddress()
                    + " -> " + channel.getRemoteAddress()));
        }
    } finally {
        // 通道集合移除该请求对应的通道，代表着这一次请求结束
        CHANNELS.remove(response.getId());
    }
}
```

该方法是接收响应，也就是某个请求得到了响应，那么代表这次请求任务完成，所有需要把future从集合中移除。具体的接收响应结果在doReceived方法中实现。

5.doReceived

```
private void doReceived(Response res) {
    // 获得锁
    lock.lock();
    try {
        // 设置响应
        response = res;
        if (done != null) {
            // 唤醒等待
            done.signal();
        }
    } finally {
        // 释放锁
        lock.unlock();
    }
    if (callback != null) {
        // 执行回调
        invokeCallback(callback);
    }
}
```

可以看到，当接收到响应后，会把等待的线程唤醒，然后执行回调来处理该响应结果。

6.invokeCallback

```
private void invokeCallback(ResponseCallback c) {
    ResponseCallback callbackCopy = c;
    if (callbackCopy == null) {
        throw new NullPointerException("callback cannot be null.");
    }
    c = null;
    Response res = response;
    if (res == null) {
        throw new IllegalStateException("response cannot be null. url:" + channel.getUrl());
    }

    // 如果响应成功，返回码是20
    if (res.getStatus() == Response.OK) {
        try {
            // 使用响应结果执行 完成 后的逻辑
            callbackCopy.done(res.getResult());
        } catch (Exception e) {
            logger.error("callback invoke error .reasult:" + res.getResult() + ",url:" + channel.getUrl(), e);
        }
        //超时，回调处理成超时异常
    } else if (res.getStatus() == Response.CLIENT_TIMEOUT || res.getStatus() == Response.SERVER_TIMEOUT) {
        try {
            TimeoutException te = new TimeoutException(res.getStatus() == Response.SERVER_TIMEOUT, channel,
res.getErrorMessage());
            // 回调处理异常
            callbackCopy.caught(te);
```

该方法是执行回调来处理响应结果。分为了三种情况：

1. 响应成功，那么执行完成后的逻辑。
2. 超时，会按照超时异常来处理
3. 其他，按照RuntimeException异常来处理

具体的处理都在ResponseCallback接口的实现类里执行，后面我会讲到。

7.get

```
@Override
public Object get() throws RemotingException {
    return get(timeout);
}

@Override
public Object get(int timeout) throws RemotingException {
    // 超时时间默认为1s
    if (timeout <= 0) {
        timeout = Constants.DEFAULT_TIMEOUT;
    }
    // 如果请求没有完成，也就是还没有响应返回
    if (!isDone()) {
        long start = System.currentTimeMillis();
        // 获得锁
        lock.lock();
        try {
            // 轮询 等待请求是否完成
            while (!isDone()) {
                // 线程阻塞等待
                done.await(timeout, TimeUnit.MILLISECONDS);
                // 如果请求完成或者超时，则结束
                if (isDone() || System.currentTimeMillis() - start > timeout) {
                    break;
                }
            }
```

该方法是实现了ResponseFuture定义的方法，是获得该future对应的请求对应的响应结果，其实future、请求、响应都是一一对应的。其中如果还没得到响应，则会线程阻塞等待，等到有响应结果或者超时，才返回。返回的逻辑在returnFromResponse中实现。

8.returnFromResponse

```
private Object returnFromResponse() throws RemotingException {
    Response res = response;
    if (res == null) {
        throw new IllegalStateException("response cannot be null");
    }
    // 如果正常返回，则返回响应结果
    if (res.getStatus() == Response.OK) {
        return res.getResult();
    }
    // 如果超时，则抛出超时异常
    if (res.getStatus() == Response.CLIENT_TIMEOUT || res.getStatus() == Response.SERVER_TIMEOUT) {
        throw new TimeoutException(res.getStatus() == Response.SERVER_TIMEOUT, channel, res.getErrorMessage());
    }
    // 其他 抛出RemotingException异常
    throw new RemotingException(channel, res.getErrorMessage());
}
```

这代码跟invokeCallback方法中差不多，都是把响应分了三种情况。

9.cancel

```
public void cancel() {
    // 创建一个取消请求的响应
    Response errorResult = new Response(id);
    errorResult.setErrorMessage("request future has been canceled.");
    response = errorResult;
    // 从集合中删除该请求
    FUTURES.remove(id);
    CHANNELS.remove(id);
}
```

该方法是取消一个请求，可以直接关闭一个请求，也就是值创建一个响应来回应该请求，把response值设置到该请求对于到future中，做到了中断请求的作用。该方法跟closeChannel的区别是closeChannel中对response的状态设置了CHANNEL_INACTIVE，而cancel方法是中途被主动取消的，虽然有response值，但是并没有一个响应状态。

10.RemotingInvocationTimeoutScan

```
private static class RemotingInvocationTimeoutScan implements Runnable {

    @Override
    public void run() {
        while (true) {
            try {
                for (DefaultFuture future : FUTURES.values()) {
                    // 已经完成，跳过扫描
                    if (future == null || future.isDone()) {
                        continue;
                    }
                    // 超时
                    if (System.currentTimeMillis() - future.getStartTimestamp() > future.getTimeout()) {
                        // create exception response.，创建一个超时的响应
                        Response timeoutResponse = new Response(future.getId());
                        // set timeout status.，设置超时状态，是服务端侧超时还是客户端侧超时
                        timeoutResponse.setStatus(future.isSent() ? Response.SERVER_TIMEOUT :
Response.CLIENT_TIMEOUT);
                        // 设置错误信息
                        timeoutResponse.setErrorMessage(future.getTimeoutMessage(true));
                        // handle response.，接收创建的超时响应
                        DefaultFuture.received(future.getChannel(), timeoutResponse);
                    }
                }
                // 睡眠
                Thread.sleep(30);
```

该方法是扫描调用超时任务的线程，每次都会遍历future集合，检测请求是否超时了，如果超时则创建一个超时响应来回应该请求。

```
static {
    // 开启一个后台扫描调用超时任务
    Thread th = new Thread(new RemotingInvocationTimeoutScan(), "DubboResponseTimeoutScanTimer");
    th.setDaemon(true);
    th.start();
}
```

开启一个后台线程进行扫描的逻辑写在了静态代码块里面，只开启一次。

（八）SimpleFuture

该类实现了ResponseFuture，目前没有用到，很简单的实现，我就不多说了。

（九）ExchangeHandler

该接口继承了ChannelHandler, TelnetHandler接口，是信息交换处理器接口。

```
public interface ExchangeHandler extends ChannelHandler, TelnetHandler {
    /**
     * reply.
     * 回复请求结果
     * @param channel
     * @param request
     * @return response
     * @throws RemotingException
     */
    Object reply(ExchangeChannel channel, Object request) throws RemotingException;

}
```

该接口只定义了一个回复请求结果的方法，返回的是请求结果。

（十）ExchangeHandlerDispatcher

该类实现了ExchangeHandler接口， 是信息交换处理器调度器类，也就是对应不同的事件，选择不同的处理器去处理。该类中有三个属性，分别对应了三种事件：

```
/**
 * 回复者调度器
 */
private final ReplierDispatcher replierDispatcher;

/**
 * 通道处理器调度器
 */
private final ChannelHandlerDispatcher handlerDispatcher;

/**
 * Telnet 命令处理器
 */
private final TelnetHandler telnetHandler;
```

如果事件是跟通道处理器有关的，就调用通道处理器来处理，比如：

```
@Override
@SuppressWarnings({"unchecked", "rawtypes"})
public Object reply(ExchangeChannel channel, Object request) throws RemotingException {
    return ((Replier) replierDispatcher).reply(channel, request);
}

@Override
public void connected(Channel channel) {
    handlerDispatcher.connected(channel);
}
@Override
public String telnet(Channel channel, String message) throws RemotingException {
    return telnetHandler.telnet(channel, message);
}
```

可以看到以上三种事件，回复请求结果需要回复者调度器来处理，连接需要通道处理器调度器来处理，telnet消息需要Telnet命令处理器来处理。

（十一）ExchangeHandlerAdapter

该类继承了TelnetHandlerAdapter，实现了ExchangeHandler，是信息交换处理器的适配器类。

```
public abstract class ExchangeHandlerAdapter extends TelnetHandlerAdapter implements ExchangeHandler {

    @Override
    public Object reply(ExchangeChannel channel, Object msg) throws RemotingException {
        // 直接返回null
        return null;
    }

}
```

该类直接让ExchangeHandler定义的方法reply返回null，交由它的子类选择性的去实现具体的回复请求结果。

（十二）ExchangeServer

该接口继承了Server接口，定义了两个方法：

```
public interface ExchangeServer extends Server {

    /**
     * get channels.
     * 获得通道集合
     * @return channels
     */
    Collection<ExchangeChannel> getExchangeChannels();

    /**
     * get channel.
     * 根据远程地址获得对应的信息通道
     * @param remoteAddress
     * @return channel
     */
    ExchangeChannel getExchangeChannel(InetSocketAddress remoteAddress);

}
```

该接口比较好理解，并且在Server接口基础上新定义了两个方法。直接来看看它的实现类吧。

（十三）HeaderExchangeServer

该类实现了ExchangeServer接口，是基于协议头的信息交换服务器实现类，HeaderExchangeServer是Server的装饰器，每个实现方法都会调用server的方法。

1.属性

```
protected final Logger logger = LoggerFactory.getLogger(getClass());

/**
 * 线程池
 */
private final ScheduledExecutorService scheduled = Executors.newScheduledThreadPool(1,
        new NamedThreadFactory(
                "dubbo-remoting-server-heartbeat",
                true));
/**
 * 服务器
 */
private final Server server;
// heartbeat timer
/**
 * 心跳定时器
 */
private ScheduledFuture<?> heartbeatTimer;
// heartbeat timeout (ms), default value is 0 , won't execute a heartbeat.
/**
 * 心跳周期
 */
private int heartbeat;
/**
 * 心跳超时时间
 */
```

该类里面的很多实现跟HeaderExchangeClient差不多，包括心跳检测等逻辑。看得懂上述我讲的HeaderExchangeClient的属性，想必这里的属性应该也很简单了。

2.构造函数

```
public HeaderExchangeServer(Server server) {
    if (server == null) {
        throw new IllegalArgumentException("server == null");
    }
    this.server = server;
    //获得心跳周期配置，如果没有配置，默认设置为0
    this.heartbeat = server.getUrl().getParameter(Constants.HEARTBEAT_KEY, 0);
    // 获得心跳超时配置，默认是心跳周期的三倍
    this.heartbeatTimeout = server.getUrl().getParameter(Constants.HEARTBEAT_TIMEOUT_KEY, heartbeat * 3);
    // 如果心跳超时时间小于心跳周期的两倍，则抛出异常
    if (heartbeatTimeout < heartbeat * 2) {
        throw new IllegalStateException("heartbeatTimeout < heartbeatInterval * 2");
    }
    // 开始心跳
    startHeartbeatTimer();
}

public Server getServer() {
    return server;
}
```

构造函数就是对属性的设置，心跳的机制以及默认值都跟HeaderExchangeClient中的一模一样。

3.isRunning

```
private boolean isRunning() {
    Collection<Channel> channels = getChannels();
    // 遍历所有连接该服务器的通道
    for (Channel channel : channels) {

        /**
         *  If there are any client connections,
         *  our server should be running.
         */

        // 只要有任何一个客户端连接，则服务器还运行着
        if (channel.isConnected()) {
            return true;
        }
    }
    return false;
}
```

该方法是检测服务器是否还运行，只要有一个客户端连接着，就算服务器运行着。

4.close

```
@Override
public void close() {
    // 关闭线程池和心跳检测
    doClose();
    // 关闭服务器
    server.close();
}

@Override
public void close(final int timeout) {
    // 开始关闭
    startClose();
    if (timeout > 0) {
        final long max = (long) timeout;
        final long start = System.currentTimeMillis();
        if (getUrl().getParameter(Constants.CHANNEL_SEND_READONLYEVENT_KEY, true)) {
            // 发送 READONLY_EVENT事件给所有连接该服务器的客户端，表示 Server 不可读了。
            sendChannelReadOnlyEvent();
        }
        // 当服务器还在运行，并且没有超时，睡眠，也就是等待timeout左右时间在进行关闭
        while (HeaderExchangeServer.this.isRunning()
                && System.currentTimeMillis() - start < max) {
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                logger.warn(e.getMessage(), e);
```

两个close方法，第二个close方法是优雅的关闭，有一定的延时来让一些响应或者操作做完。关闭分两个步骤，第一个就是关闭信息交换服务器中的线程池和心跳检测，然后才是关闭服务器。

5.sendChannelReadOnlyEvent

```
private void sendChannelReadOnlyEvent() {
    // 创建一个READONLY_EVENT事件的请求
    Request request = new Request();
    request.setEvent(Request.READONLY_EVENT);
    // 不需要响应
    request.setTwoWay(false);
    // 设置版本
    request.setVersion(Version.getProtocolVersion());

    Collection<Channel> channels = getChannels();
    // 遍历连接的通道，进行通知
    for (Channel channel : channels) {
        try {
            // 通过通道还连接着，则发送通知
            if (channel.isConnected())
                channel.send(request, getUrl().getParameter(Constants.CHANNEL_READONLYEVENT_SENT_KEY, true));
        } catch (RemotingException e) {
            logger.warn("send cannot write message error.", e);
        }
    }
}
```

在关闭服务器中有一个操作就是发送事件READONLY_EVENT，告诉客户端该服务器不可读了，就是该方法实现的，逐个通知连接的客户端该事件。

6.doClose

```
private void doClose() {
    if (!closed.compareAndSet(false, true)) {
        return;
    }
    // 停止心跳检测
    stopHeartbeatTimer();
    try {
        // 关闭线程池
        scheduled.shutdown();
    } catch (Throwable t) {
        logger.warn(t.getMessage(), t);
    }
}
```

该方法就是close方法调用到的停止心跳检测和关闭线程池。

7.getExchangeChannels

```
@Override
public Collection<ExchangeChannel> getExchangeChannels() {
    Collection<ExchangeChannel> exchangeChannels = new ArrayList<ExchangeChannel>();
    // 获得连接该服务器通道集合
    Collection<Channel> channels = server.getChannels();
    if (channels != null && !channels.isEmpty()) {
        // 遍历通道集合，为每个通道都创建信息交换通道，并且加入信息交换通道集合
        for (Channel channel : channels) {
            exchangeChannels.add(HeaderExchangeChannel.getOrAddChannel(channel));
        }
    }
    return exchangeChannels;
}
```

该方法是返回连接该服务器信息交换通道集合。逻辑就是先获得通道集合，在根据通道来创建信息交换通道，然后返回信息通道集合。

8.reset

```
@Override
public void reset(URL url) {
    // 重置属性
    server.reset(url);
    try {
        // 重置的逻辑跟构造函数一样设置
        if (url.hasParameter(Constants.HEARTBEAT_KEY)
                || url.hasParameter(Constants.HEARTBEAT_TIMEOUT_KEY)) {
            int h = url.getParameter(Constants.HEARTBEAT_KEY, heartbeat);
            int t = url.getParameter(Constants.HEARTBEAT_TIMEOUT_KEY, h * 3);
            if (t < h * 2) {
                throw new IllegalStateException("heartbeatTimeout < heartbeatInterval * 2");
            }
            if (h != heartbeat || t != heartbeatTimeout) {
                heartbeat = h;
                heartbeatTimeout = t;
                // 重新开始心跳
                startHeartbeatTimer();
            }
        }
    } catch (Throwable t) {
        logger.error(t.getMessage(), t);
    }
}
```

该方法就是重置属性，重置后，重新开始心跳，设置心跳属性的机制跟构造函数一样。

9.startHeartbeatTimer

```
private void startHeartbeatTimer() {
    // 先停止现有的心跳检测
    stopHeartbeatTimer();
    if (heartbeat > 0) {
        // 创建心跳定时器
        heartbeatTimer = scheduled.scheduleWithFixedDelay(
                new HeartBeatTask(new HeartBeatTask.ChannelProvider() {
                    @Override
                    public Collection<Channel> getChannels() {
                        // 返回一个不可修改的连接该服务器的信息交换通道集合
                        return Collections.unmodifiableCollection(
                                HeaderExchangeServer.this.getChannels());
                    }
                }, heartbeat, heartbeatTimeout),
                heartbeat, heartbeat, TimeUnit.MILLISECONDS);
    }
}
```

该方法是开始心跳，跟HeaderExchangeClient类中的开始心跳方法唯一区别是获得的通道不一样，客户端跟通道是一一对应的，所有只要对一个通道进行心跳检测，而服务端跟通道是一对多的关系，所有需要对该服务器连接的所有通道进行心跳检测。

10.stopHeartbeatTimer

```
private void stopHeartbeatTimer() {
    if (heartbeatTimer != null && !heartbeatTimer.isCancelled()) {
        try {
            // 取消定时器
            heartbeatTimer.cancel(true);
            // 取消大量已排队任务，用于回收空间
            scheduled.purge();
        } catch (Throwable e) {
            if (logger.isWarnEnabled()) {
                logger.warn(e.getMessage(), e);
            }
        }
    }
    heartbeatTimer = null;
}
```

该方法是停止当前的心跳检测。

（十四）ExchangeServerDelegate

该类实现了ExchangeServer接口，是信息交换服务器装饰者，是ExchangeServer的装饰器。该类就一个属性ExchangeServer server，所有实现方法都调用了server属性的方法。目前只有在p2p中被用到，代码为就不贴了，很简单。

（十五）Exchanger

```
@SPI(HeaderExchanger.NAME)
public interface Exchanger {

    /**
     * bind.
     * 绑定一个服务器
     * @param url 服务器url
     * @param handler 数据交换处理器
     * @return message server 数据交换服务器
     */
    @Adaptive({Constants.EXCHANGER_KEY})
    ExchangeServer bind(URL url, ExchangeHandler handler) throws RemotingException;

    /**
     * connect.
     * 连接一个服务器，也就是创建一个客户端
     * @param url 服务器url
     * @param handler 数据交换处理器
     * @return message channel 返回数据交换客户端
     */
    @Adaptive({Constants.EXCHANGER_KEY})
    ExchangeClient connect(URL url, ExchangeHandler handler) throws RemotingException;

}
```

该接口是数据交换者接口，该接口是一个可扩展接口默认实现的是HeaderExchanger类，并且用到了dubbo SPI的Adaptive机制，优先实现url携带的配置。如果不了解dubbo SPI机制的可以看《dubbo源码解析（二）Dubbo扩展机制SPI》。那么回到该接口定义的方法，定义了绑定和连接两个方法，分别返回信息交互服务器和客户端实例。

（十六）HeaderExchanger

```
public class HeaderExchanger implements Exchanger {

    public static final String NAME = "header";

    @Override
    public ExchangeClient connect(URL url, ExchangeHandler handler) throws RemotingException {
        // 用传输层连接返回的client 创建对应的信息交换客户端，默认开启心跳检测
        return new HeaderExchangeClient(Transporters.connect(url, new DecodeHandler(new
HeaderExchangeHandler(handler))), true);
    }

    @Override
    public ExchangeServer bind(URL url, ExchangeHandler handler) throws RemotingException {
        // 用传输层绑定返回的server 创建对应的信息交换服务端
        return new HeaderExchangeServer(Transporters.bind(url, new DecodeHandler(new
HeaderExchangeHandler(handler))));
    }

}
```

该类继承了Exchanger接口，是Exchanger接口的默认实现，实现了Exchanger接口定义的两个方法，分别调用的是Transporters的连接和绑定方法，再利用这这两个方法返回的客户端和服务端实例来创建信息交换的客户端和服务端。

（十七）Replier

我们知道Request对应的是ExchangeHandler接口实现对象来处理，但有些时候我们需要不同数据类型对应不同的处理器，该类就是为了支持这一需求所设计的。

```
public interface Replier<T> {

    /**
     * reply.
     * 回复请求结果
     * @param channel
     * @param request
     * @return response
     * @throws RemotingException
     */
    Object reply(ExchangeChannel channel, T request) throws RemotingException;

}
```

可以看到该接口跟ExchangeHandler定义的方法也一一，只有请求的类型改为了范型。

（十八）ReplierDispatcher

该类实现了Replier接口，是回复者调度器实现类。

```
/**
 * 默认回复者
 */
private final Replier<?> defaultReplier;

/**
 * 回复者集合
 */
private final Map<Class<?>, Replier<?>> repliers = new ConcurrentHashMap<Class<?>, Replier<?>>();
```

这是该类的两个属性，缓存了回复者集合和默认的回复者。

```
/**
 * 从回复者集合中找到该类型的回复者，并且返回
 * @param type
 * @return
 */
private Replier<?> getReplier(Class<?> type) {
    for (Map.Entry<Class<?>, Replier<?>> entry : repliers.entrySet()) {
        if (entry.getKey().isAssignableFrom(type)) {
            return entry.getValue();
        }
    }
    if (defaultReplier != null) {
        return defaultReplier;
    }
    throw new IllegalStateException("Replier not found, Unsupported message object: " + type);
}

/**
 * 回复请求
 * @param channel
 * @param request
 * @return
 * @throws RemotingException
 */
@Override
@SuppressWarnings({"unchecked", "rawtypes"})
```

上述是该类中关键的两个方法，reply还是调用实现类的reply。根据请求的数据类型来使用指定的回复者进行回复。

（十九）MultiMessage

该类实现了实现 Iterable 接口，是多消息的封装，我们直接看它的属性：

```
/**
 * 消息集合
 */
private final List messages = new ArrayList();
```

该类要和《dubbo源码解析（九）远程通信——Transport层》的（八）MultiMessageHandler联合着看。

（二十）HeartbeatHandler

该类继承了AbstractChannelHandlerDelegate类，是心跳处理器。是用来处理心跳事件的，也接收消息上增加了对心跳消息的处理。该类是

```
@Override
public void received(Channel channel, Object message) throws RemotingException {
    // 设置接收时间的时间戳属性值
    setReadTimestamp(channel);
    // 如果是心跳请求
    if (isHeartbeatRequest(message)) {
        Request req = (Request) message;
        // 如果需要响应
        if (req.isTwoWay()) {
            // 创建一个响应
            Response res = new Response(req.getId(), req.getVersion());
            // 设置为心跳事件的响应
            res.setEvent(Response.HEARTBEAT_EVENT);
            // 发送消息，也就是返回响应
            channel.send(res);
            if (logger.isInfoEnabled()) {
                int heartbeat = channel.getUrl().getParameter(Constants.HEARTBEAT_KEY, 0);
                if (logger.isDebugEnabled()) {
                    logger.debug("Received heartbeat from remote channel " + channel.getRemoteAddress()
                            + ", cause: The channel has no data-transmission exceeds a heartbeat period"
                            + (heartbeat > 0 ? ": " + heartbeat + "ms" : ""));
                }
            }
        }
        return;
    }
```

该方法是就是在handler处理消息上增加了处理心跳消息的功能，做到了功能增强。

（二十一）Exchangers

该类跟Transporters的设计意图是一样的，Transporters我在《dubbo源码解析（八）远程通信——开篇》的（十）Transporters已经讲到了。Exchangers也用到了外观模式。代码为就不贴了，可以对照着Transporters来看，很简单。

（二十二）Request

请求模型类，最重要的肯定是模型的属性，我们来看看属性：

```
/**
 * 心跳事件
 */
public static final String HEARTBEAT_EVENT = null;

/**
 * 只读事件
 */
public static final String READONLY_EVENT = "R";

/**
 * 请求编号自增序列
 */
private static final AtomicLong INVOKE_ID = new AtomicLong(0);

/**
 * 请求编号
 */
private final long mId;

/**
 * dubbo版本
 */
private String mVersion;

/**
```

1. 由于心跳事件比较常用，所有设置为null。
2. 请求编号使用INVOKE_ID生成，是JVM 进程内唯一的。
3. 其他属性比较简单

（二十三）Response

响应模型，来看看它的属性：

```
/**
 * 心跳事件
 */
public static final String HEARTBEAT_EVENT = null;

/**
 * 只读事件
 */
public static final String READONLY_EVENT = "R";

/**
 * ok.
 * 成功状态码
 */
public static final byte OK = 20;

/**
 * clien side timeout.
 * 客户端侧的超时状态码
 */
public static final byte CLIENT_TIMEOUT = 30;

/**
 * server side timeout.
 * 服务端侧超时的状态码
 */
```

很多属性跟Request模型的属性一样，并且含义也一样，不过该模型多了很多的状态码。关键的是id跟请求一一对应。

（二十四）ResponseCallback

```
public interface ResponseCallback {

    /**
     * done.
     * 处理请求
     * @param response
     */
    void done(Object response);

    /**
     * caught exception.
     * 处理异常
     * @param exception
     */
    void caught(Throwable exception);

}
```

该接口是回调的接口，定义了两个方法，分别是处理正常的响应结果和处理异常。

（二十五）ExchangeCodec

该类继承了TelnetCodec，是信息交换编解码器。在本文的开头，我就写到，dubbo将一条消息分成了协议头和协议体，用来解决粘包拆包问题，但是头跟体在编解码上有区别，我们先来看看dubbo 的协议头的配置：

上图是官方文档的图片，能够清晰的看出协议中各个数据所占的位数：

1. 0-7位和8-15位：Magic High和Magic Low，类似java字节码文件里的魔数，用来判断是不是dubbo协议的数据包，就是一个固定的数字
2. 16位：Req/Res：请求还是响应标识。
3. 17位：2way：单向还是双向
4. 18位：Event：是否是事件
5. 19-23位：Serialization 编号
6. 24-31位：status状态
7. 32-95位：id编号
8. 96-127位：body数据
9. 128-...位：上图表格内的数据

可以看到一个该协议中前65位是协议头，后面的都是协议体数据。那么在编解码中，协议头是通过 Codec 编解码，而body部分是用Serialization序列化和反序列化的。下面我们就来看看该类对协议头的编解码。

1.属性

```
// header length.
/**
 * 协议头长度：16字节 = 128Bits
 */
protected static final int HEADER_LENGTH = 16;
// magic header.
/**
 * MAGIC二进制：1101101010111011，十进制：55995
 */
protected static final short MAGIC = (short) 0xdabb;
/**
 * Magic High，也就是0-7位：11011010
 */
protected static final byte MAGIC_HIGH = Bytes.short2bytes(MAGIC)[0];
/**
 * Magic Low  8-15位 ：10111011
 */
protected static final byte MAGIC_LOW = Bytes.short2bytes(MAGIC)[1];
// message flag.
/**
 * 128 二进制：10000000
 */
protected static final byte FLAG_REQUEST = (byte) 0x80;
/**
 * 64 二进制：1000000
 */
```

可以看到 MAGIC是个固定的值，用来判断是不是dubbo协议的数据包，并且MAGIC_LOW和MAGIC_HIGH分别是MAGIC的低位和高位。其他的属性用来干嘛后面会讲到。

2.encode

```
@Override
public void encode(Channel channel, ChannelBuffer buffer, Object msg) throws IOException {
    if (msg instanceof Request) {
        // 如果消息是Request类型，对请求消息编码
        encodeRequest(channel, buffer, (Request) msg);
    } else if (msg instanceof Response) {
        // 如果消息是Response类型，对响应消息编码
        encodeResponse(channel, buffer, (Response) msg);
    } else {
        // 直接让父类( Telnet ) 处理，目前是 Telnet 命令的结果。
        super.encode(channel, buffer, msg);
    }
}
```

该方法是根据消息的类型来分别进行编码，分为三种情况：Request类型、Response类型以及其他

3.encodeRequest

```
protected void encodeRequest(Channel channel, ChannelBuffer buffer, Request req) throws IOException {
    Serialization serialization = getSerialization(channel);
    // header.
    // 创建16字节的字节数组
    byte[] header = new byte[HEADER_LENGTH];
    // set magic number.
    // 设置前16位数据，也就是设置header[0]和header[1]的数据为Magic High和Magic Low
    Bytes.short2bytes(MAGIC, header);

    // set request and serialization flag.
    // 16-23位为serialization编号，用到或运算10000000|serialization编号，例如serialization编号为11111，则为00011111
    header[2] = (byte) (FLAG_REQUEST | serialization.getContentTypeId());

    // 继续上面的例子，00011111|1000000 = 01011111
    if (req.isTwoWay()) header[2] |= FLAG_TWOWAY;
    // 继续上面的例子，01011111|100000 = 011 11111 可以看到011代表请求标记、双向、是事件，这样就设置了16、17、18位，后面
19-23位是Serialization 编号
    if (req.isEvent()) header[2] |= FLAG_EVENT;

    // set request id.
    // 设置32-95位请求id
    Bytes.long2bytes(req.getId(), header, 4);

    // encode request data.
    // // 编码 `Request.data` 到 Body ，并写入到 Buffer
    int savedWriteIndex = buffer.writerIndex();
```

该方法是对Request类型的消息进行编码，仔细阅读上述我写的注解，结合协议头各个位数的含义，好好品味我举的例子。享受二进制位运算带来的快乐，也可以看到前半部分逻辑是对协议头的编码，后面还有对body值的序列化。

4.encodeResponse

```
protected void encodeRequest(Channel channel, ChannelBuffer buffer, Request req) throws IOException {
    Serialization serialization = getSerialization(channel);
    // header.
    // 创建16字节的字节数组
    byte[] header = new byte[HEADER_LENGTH];
    // set magic number.
    // 设置前16位数据，也就是设置header[0]和header[1]的数据为Magic High和Magic Low
    Bytes.short2bytes(MAGIC, header);

    // set request and serialization flag.
    // 16-23位为serialization编号，用到或运算10000000|serialization编号，例如serialization编号为11111，则为00011111
    header[2] = (byte) (FLAG_REQUEST | serialization.getContentTypeId());

    // 继续上面的例子，00011111|1000000 = 01011111
    if (req.isTwoWay()) header[2] |= FLAG_TWOWAY;
    // 继续上面的例子，01011111|100000 = 011 11111 可以看到011代表请求标记、双向、是事件，这样就设置了16、17、18位，后面
19-23位是Serialization 编号
    if (req.isEvent()) header[2] |= FLAG_EVENT;

    // set request id.
    // 设置32-95位请求id
    Bytes.long2bytes(req.getId(), header, 4);

    // encode request data.
    // // 编码 `Request.data` 到 Body ，并写入到 Buffer
    int savedWriteIndex = buffer.writerIndex();
```

该方法是对Response类型的消息进行编码，该方法里面我没有举例子演示如何进行编码，不过过程跟encodeRequest类似。

5.decode

```
@Override
public Object decode(Channel channel, ChannelBuffer buffer) throws IOException {
    int readable = buffer.readableBytes();
    // 读取前16字节的协议头数据，如果数据不满16字节，则读取全部
    byte[] header = new byte[Math.min(readable, HEADER_LENGTH)];
    buffer.readBytes(header);
    // 解码
    return decode(channel, buffer, readable, header);
}

@Override
protected Object decode(Channel channel, ChannelBuffer buffer, int readable, byte[] header) throws IOException {
    // check magic number.
    // 核对魔数（该数字固定）
    if (readable > 0 && header[0] != MAGIC_HIGH
            || readable > 1 && header[1] != MAGIC_LOW) {
        int length = header.length;
        // 将 buffer 完全复制到 `header` 数组中
        if (header.length < readable) {
            header = Bytes.copyOf(header, readable);
            buffer.readBytes(header, length, readable - length);
        }
        for (int i = 1; i < header.length - 1; i++) {
            if (header[i] == MAGIC_HIGH && header[i + 1] == MAGIC_LOW) {
                buffer.readerIndex(buffer.readerIndex() - header.length + i);
                header = Bytes.copyOf(header, i);
```

该方法就是解码前的一些核对过程，包括检测是否为dubbo协议，是否有拆包现象等，具体的解码在decodeBody方法。

6.decodeBody

```
protected Object decodeBody(Channel channel, InputStream is, byte[] header) throws IOException {
    // 用并运算符
    byte flag = header[2], proto = (byte) (flag & SERIALIZATION_MASK);
    // get request id.
    // 获得请求id
    long id = Bytes.bytes2long(header, 4);
    // 如果第16位为0，则说明是响应
    if ((flag & FLAG_REQUEST) == 0) {
        // decode response.
        Response res = new Response(id);
        // 如果第18位不是0，则说明是心跳事件
        if ((flag & FLAG_EVENT) != 0) {
            res.setEvent(Response.HEARTBEAT_EVENT);
        }
        // get status.
        byte status = header[3];
        res.setStatus(status);
        try {
            ObjectInput in = CodecSupport.deserialize(channel.getUrl(), is, proto);
            // 如果响应是成功的
            if (status == Response.OK) {
                Object data;
                if (res.isHeartbeat()) {
                    // 如果是心跳事件，则心跳事件的解码
                    data = decodeHeartbeatData(channel, in);
                } else if (res.isEvent()) {
```

该方法就是解码的过程，并且对协议头和协议体分开解码，协议头编码是做或运算，而解码则是做并运算，协议体用反序列化的方式解码，同样也是分为了Request类型、Response类型进行解码。

后记

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